ETSI TS 103 973 V1.1.1 (2024-10)

Coded Multisource Media Format (CMMF) for Content Distribution and Delivery

EBU

Reference
DTS/JTC-112

Keywords
broadband, broadcast, CDN, container,
distribution, multimedia, multi-path, multi-source,
network coding, robustness

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 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

List of figures

List of tables

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are 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 (https://ipr.etsi.org/).

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, 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.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™** and **LTE™** are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This Technical Specification (TS) 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: 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

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Introduction

The Coded Multisource Media Format (CMMF) is an extensible container format designed to facilitate the management and interchange of audio-visual media and metadata in one or more coded representations (e.g. encoded with application-layer, linear, network, or channel codes).

The coded media representations supported by CMMF enable the efficient use of multisource, multipath and multi-access connectivity for network-delivered media applications.

Additionally, CMMF supports signalling and encapsulations of:

- Coding type.
- Media-related information and metadata.
- Payload integrity information.
- Encoder UUID.

This bitstream format provides a standard to the industry so that a single container format can be used for a wide variety of content and use cases.

Motivation

CMMF provides a generic container format that supports multimedia (e.g. video and audio streaming, broadcast, XR, video conferencing, and online gaming) delivery through coding the underlying content. This format supports multiple types of codes (currently xCD-1, RaptorQ, and Reed-Solomon) and can be optimized for a range of networks and use cases. Specifically, CMMF supports efficient decentralized multi-source and multi-path content delivery for use cases such as audio and video streaming that require high availability/robustness but also have strict latency and bandwidth constraints. CMMF is designed to operate with existing and future streaming source (e.g. HLS, MPEG-DASH, CMAF, etc.) and network protocols (e.g. HTTP, TCP, UDP, WebRTC, etc.) protocols, while remaining protocol-agnostic. A multisource media encoder is envisioned to take an existing packaged media format as a source and generate CMMF bitstreams for delivery over networks to clients for rendering.

CMMF provides a flexible and extensible framework for managing the delivery of encoded multimedia content. Standardizing the container format rather than a code type enables cooperation within the industry by creating a common interchange format for the distribution and delivery of encoded content. This allows Service Providers (e.g. Mobile Network Operators or media platforms) to distribute media in an encoded format that can be interpreted and decoded by their partners.

Document structure

Clause 4 gives an overview of the functionality provided by the CMMF bitstream format.

Clause 5 provides the definition of the CMMF bitstream format, with clause 5.1 defining the semantics, and clause 5.2 the syntax.

Clause 6 provides a description of the bitstream elements.

Clause 7 gives guidance for implementations of the CMMF container format.

Annex A provides the definition of the xCD-1 coding format.

Annex B provides a high-level example of a Media Service Architecture with CMMF.

Annex C provides application examples of the CMMF container format, with different use cases and coding formats.

Annex D provides a mapping of CMMF to IETF RFC 5052 [15] principles, including application examples.

1 Scope

The present document specifies a Coded Multisource Media Format (CMMF) container. This format is used to support storage and delivery of linear, network or channel coded multisource audio/video media over networks. It also specifies the xCD-1 linear coding type. The present document also includes examples.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

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

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] IETF RFC 9562: "Universally Unique IDentifiers (UUIDs)".

[2] EIDR: "EIDR ID Format", V1.51, October 5, 2017.

[3] IETF RFC 8107: "Advertising Digital Identifier (Ad-ID) URN Namespace Definition".

[4] ISO 639: "Code for individual languages and language groups".

[5] IETF RFC 6381: "The 'Codecs' and 'Profiles' Parameters for "Bucket" Media Types".

[6] IETF RFC 4646: "Tags for Identifying Languages".

[7] ISO/IEC 646:1991: "Information technology -- ISO 7-bit coded character set for information interchange".

[8] Mersenne Twister Home Page: "A very fast random number generator Of period 2^{19937}-1".

[9] FIPS PUB 180-4: "Secure Hash Standard (SHS)", August 2015.

[10] ISO/IEC 23091-2: "Information technology -- Coding-independent code point -- Part 2: Video".

[11] IETF RFC 3629: "UTF-8, a transformation format of ISO 10646".

[12] IETF RFC 4648: "The Base16, Base32, and Base64 Data Encodings".

[13] IETF RFC 6330: "RaptorQ Forward Error Correction Scheme for Object Delivery".

[14] IETF RFC 5510: "Reed-Solomon Forward Error Correction (FEC) Schemes".

[15] IETF RFC 5052: "Forward Error Correction (FEC) Building Block".

[16] IETF RFC 5053: "Raptor Forward Error Correction Scheme for Object Delivery".

[17] IETF RFC 5775: "Asynchronous Layered Coding (ALC) Protocol Instantiation".

[18] IETF RFC 6726: " FLUTE - File Delivery over Unidirectional Transport".

[19] ETSI TS 126 346: "Universal Mobile Telecommunications System (UMTS); LTE; 5G; Multimedia Broadcast/Multicast Service (MBMS); Protocols and codecs (3GPP TS 26.346)".

[20] Recommendation ITU-R BS.2051-3: "Advanced sound system for programme production".

[21] IETF RFC 1321: "The MD5 Message-Digest Algorithm".

[22] IETF RFC 4337: "MIME Type Registration for MPEG-4".

[23] IETF RFC 2046: "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] IETF RFC 8446: "The Transport Layer Security (TLS) Protocol Version 1.3".

[i.2] ISO/IEC 23009-1: "Information technology -- Dynamic adaptive streaming over HTTP (DASH) -- Part 1: Media presentation description and segment formats".

[i.3] ISO/IEC 23000-19: "Information technology -- Multimedia application format (MPEG-A) -- Part 19: Common media application format (CMAF) for segmented media".

[i.4] IETF RFC 2616: "Hypertext Transfer Protocol -- HTTP/1.1".

[i.5] IETF RFC 9110: "HTTP Semantics".

3 Definition of terms, symbols and abbreviations

3.1 Terms

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

bitstream: sequence of bits

block: unit of data on which a block-based code is applied

NOTE: All coding operations (encoding and decoding) on one block are independent of all coding operations on another block.

block coding: coding technique where the input is first be segmented into a sequence of blocks, or chunks; then encoding and decoding are performed independently on a per-block basis

block size: number of bytes that a block of data contains

block symbol count: number of source symbols belonging to a block

byte: 8 bits

channel: generic term for any type of communication technology

EXAMPLE: An Ethernet link, a Wi-Fi® network, or a full path between two nodes within a network.

code rate: ratio between the number of source symbols, and the number of source plus coded or repair symbols

NOTE: The code rate is greater than zero, and less or equal to one, where a code rate close to one indicates that a small number of coded or repair symbols have been produced during the encoding process, while a code rate close to zero indicates a large number of coded or repair symbols.

code type: type of coding (linear, network, or channel coding scheme) used to create coded symbols

coded symbol, encoded symbol, or **repair symbol:** unit of data that is the result of a coding operation

coding or **encoding:** operation that takes source symbols as input and produces coded symbols as output

coding coefficient: coefficient chosen from the same finite field encoding and decoding operations are performed over

NOTE: Methods of choosing this coefficient may include: randomly (e.g. LT codes), in a predefined table (e.g. Reed-Solomon, etc.), or using a predefined algorithm plus a seed (e.g. LDPC, RaptorQ, etc.).

coding matrix, generator matrix, or coefficient matrix: matrix (G) that transforms the set of input symbols (X) into a set of coded or repair symbols (Y): $Y = X \bullet G$

NOTE: Defining a generator matrix is typical with block codes. The set of input symbols X can consist only of source symbols.

coding vector or **coefficient vector:** set of coding coefficients used to generate a certain coded or repair symbol through linear coding

NOTE: The number of nonzero coefficients in the Coding Vector defines its density.

decoding: operation that takes coded symbols as input and produces source symbols as output

encoding: operation that takes source symbols as input and produces coded symbols as output

encoding block: See block.

encoding symbol: See coded symbol.

encoding window or **coding window:** set of source symbols used as input to the coding operations

NOTE: The set of symbols will typically change over time, as the coding window slides over the input flow.

encoding window size or **coding window size:** number of source symbols in the current encoding window

NOTE: This size may change over the time.

erasure: drop or loss of information along a communication path

erasure channel: communication path where information is either dropped or received without any error

finite field, **galois field**, or **coding field:** finite fields, used in linear codes, have the desired property of having all elements (except zero) invertible for the + and × operators, and all operations over any elements do not result in an overflow or underflow

finite field size or **coding field size:** number of elements in a finite field

EXAMPLE: The binary extension field $\{0..2^m-1\}$ has size $q = 2^m$.

flow or stream: stream of information (or packets) that are logically grouped

input or source symbol: unit of data that is an input to an encoding operation or an output of a decoding operation

linear coding: process in which a linear combination of a set of source symbols is generated using a given set of coefficients and resulting in a coded symbol or repair symbol

multipath coding: coding that enables transmission over multiple routes that have multiple (at least partially) disjoint paths from the source to each given destination

multisource coding: coding that enables transmission from multiple sources over (at least partially) disjoint paths to each given destination

network: interconnected set of nodes that communicate over a collection of links or channels

node: point of connection in a communication network

object: ordered sequence of data and associated metadata delivered as part of a flow. an object can be a source object, or a coded/repair object

output symbol: See coded symbol.

packet: unit of data that is sent over a network

rank or encoding rank: number of linearly independent linearly encoded symbols, or equivalently the number of linearly independent equations of the linear system

repair flow: flow containing repair packets after FEC encoding

single-path coding: coding over a route that has a single path from the source to each destination(s). in case of multicast or broadcast traffic, this route is a tree

sliding window coding: coding technique that generates coded or repair symbol(s) on the fly, from the set of source symbols present in the sliding encoding window at that time

NOTE: The sliding window may be either of fixed size (Fixed Sliding Window), or of variable size over time (Elastic Sliding Window).

sliding window size, **encoding window size**, or **coding window size:** number of symbols in the current window

NOTE: This size may change over the time.

source coding: process of removing redundant and/or (perceptually) irrelevant information from an information source, i.e. compression of data or media (audio, video)

source data, source file, or original data: unit of data that may be partitioned into blocks where each block is an input to an encoding operation

source flow: flow of source information to which coding is to be applied, potentially along with other source flows

source node: node that generates one or more source flows

source symbol, **information symbol**, **systematic symbol:** unit of data originating from the source that is used as input to encoding operations

symbol: unit of data that is manipulated during encoding and decoding operations

symbol size: size of each symbol on which encode and decode operations are performed

systematic coding: coding technique where source symbols are part of the output flow generated by an encoder

3.2 Symbols

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

^ Exponentiation: a^b is equivalent to a^b

3.3 Abbreviations

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

Ad-ID	Advertising digital Identifier
AES	Advanced Encryption Standard
ASCII	American Standard Code for Information Interchange
CBC	Cipher Block Chaining
CDN	Content Delivery Network
CDP	Content Delivery Protocol
CFB	Cipher FeedBack
CI	Configuration Information
CMMF	Coded Multisource Media Format
CRC	Cyclic Redundancy Check

CTR	CounTeR
DASH	Dynamic Adaptive Streaming over HTTP
ECB	Electronic Code Book
EFD	Extended FDT Description
EFDT	Extended File Delivery Table
EIDR	Entertainment IDentifier Registry
ESI	Encoding Symbol ID
FDT	File Delivery Table
FEC	Forward Erasure Correction
HAS	HTTP Adaptive Streaming
HDR	High Dynamic Range
HLS	HTTP Live Streaming
HTTP	Hyper-Text Transfer Protocol
IV	Initialization Vector
KMS	Key Management System
LDPC	Low-Density Parity-Check
LT code	Luby Transform code
MIME	Multipurpose Internet Mail Extensions
MT	Mersenne Twister
OFB	Output FeedBack
OTI	Object Transmission Information
PGP	Pretty Good Privacy
PRNG	PseudoRandom Number Generator
SBN	Source Block Number
SHA-1	Secure Hash Algorithm 1
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TOI	Transport Object Identifier
TSI	Transport Session Identifier
UDP	User Datagram Protocol
UTF	Unicode Transformation Format
UUID	Universally Unique IDentifier
VoD	Video-on-Demand
xCD-1	Multisource Media Coding (Experience Coding and Delivery)

4 Overview

4.0 Introduction

Media delivery over the internet today relies on using a single representation for coded media across storage, distribution, and delivery systems. Within a server-client single-path streaming model, traditional source coding techniques are applied to the media and segment-based media formatting is utilized for storage and delivery. This allows clients to download appropriate bit rates and/or resolutions of media during streaming sessions, aiming to provide a high-quality experience across varying network conditions.

Content Delivery Networks (CDNs) are commonly used to copy media across multiple Points of Presence (PoPs) to help improve Quality of Service (QoS) and Quality of Experience (QoE) by distributing and caching media closer to viewers. Client traffic from a CDN is commonly delivered through a single network path and relies on a single representation of the media. The CDN server fulfils client requests during the streaming session until the client (or related steering mechanism) requests switching to a different CDN due to network conditions, performance metrics, or other metrics that impact QoE. Switching among available CDNs is not always seamless and can result in clients experiencing delays and/or periods of degraded QoE.

Content services using multiple CDNs replicate identical media representations across each of them. This redundancy limits the efficiency in settings where, the downloading of media over a multiple network paths to a client device from two or more CDNs, content origins, network storage locations, or in multi-access settings, is necessary to ensure the highest degree of network QoS is continuously available.

The Coded Multisource Media Format (CMMF) provides a framework to enable efficient use of multiple information sources and network pathways to improve network QoS and client QoE in cloud-based or network-powered settings. By redesigning the media format from using a single representation, single encoded version of the source media to using multiple representations of the source media (multisource), CMMF enables seamless load balancing among the most performant network pathways and information sources (e.g. CDN servers, storage servers, etc.) and limits or eliminates the amount of redundant information stored and transmitted since each CMMF source contains a unique/useful coded version of the source media. This method efficiently enhances network performance (e.g. increased throughput/rendered bit rate, lower rebuffering and start times) and resilience without having to rely on centralized orchestration or complex download scheduling.

CMMF is designed to carry multisource-coded representations of media, where multiple CMMF bitstreams can be generated with each containing unique/useful coded versions of the source content. Multisource representations (or CMMF encoded versions) are generated using various linear, network, or channel coding methods. A single CMMF bitstream may contain all of the necessary information needed to reconstruct the original source information; or in some use cases, multiple CMMF bitstreams may be required to fully reconstruct the original source content.

Different coding methods are supported in CMMF. Some coding methods are typically employed in Forward Error Correction (FEC) applications, such as FEC codes/schemes that follow IETF RFC 5052 [15]. The coded data generated by these codes can be mapped to and carried within a CMMF bitstream. As such, similar concepts for partitioning of media data, ancillary data to aid in transmission and delivery, are re-used and expanded upon for the context of creating multisource representations (or simply to leverage a common bitstream format for carrying a single source representation). While the terms "repair" or "FEC" appear in the present document, this is done to provide a foundation for comparison with terminology used for these existing codes/schemes and is not necessarily meant to imply that CMMF is solely used to repair some form of loss or corruption.

CMMF has been designed in accordance with standard networking principles as an additional, independent layer within the media delivery stack. Moreover, CMMF is agnostic to how the underlying source content has been prepared (i.e. encoded and packaged) and is also agnostic to the underlying methods used to store and transport content within a network.

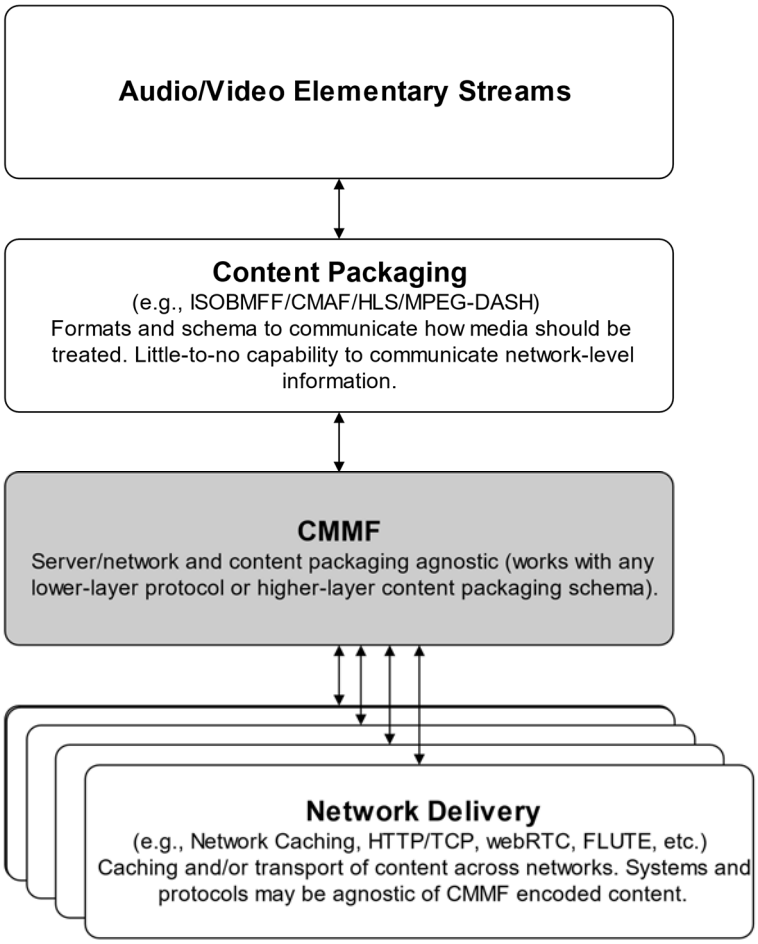

Figure 1: CMMF layer within multimedia transport stack

A system utilizing CMMF is envisaged to:

- Take elementary media streams in an already packaged media format as the source data.
- Encode the source media using an appropriate CMMF coding method.
- Generate multiple CMMF bitstreams according to the syntax and semantics described in clause 5 and clause 6.
- Deliver the CMMF bitstreams (either on-the-fly or taken from storage) and possibly ancillary configuration data over networks using an appropriate transport protocol(s) to clients.
- Decode the received CMMF bitstreams.
- Client renders the CMMF-decoded media.

Annex C provides different examples illustrating how CMMF can be used to provide multisource media delivery.

4.1 Source data

Source data is used to create multiple CMMF encoded versions. CMMF can be applied to any media object type, such as a WAV audio clip, an MPEG-4 video clip, an MPEG Dynamic Adaptive Streaming over HTTP (DASH) [i.2] video segment, or an MPEG Common Media Application Format (CMAF) [i.3] addressable resource.

In a common practice, a CMMF encoder will take an existing packaged media format (e.g. ISO BMFF/CMAF). CMMF is not a replacement for these formats. Rather, CMMF supplements these formats by enabling more efficient and flexible strategies to deploy and manage these existing formats within the network. Staying agnostic to the packaging format allows CMMF to operate in a decentralized manner by avoiding the requirement of establishing and maintaining manifests. For any given asset, new CMMF bitstreams can be dynamically generated and/or destroyed without the need to touch existing CMMF bitstreams for that asset or updating manifests in all the places they are located.

The source data encoded within a CMMF bitstream may have additional assigned metadata associated with it that may be used by an application or a network delivery protocol to properly process the data or deliver it across a network. Examples may include manifests, URLs to the source data (as referred to by an application), the content type, timing information, etc. It is not intended for CMMF to replace this metadata. Depending on the application, use case, network architecture, etc., this metadata may or may not be communicated using CMMF.

4.2 CMMF bitstream creation

4.2.1 Encoding CMMF

Generating a CMMF encoded version can be either a real-time process, or a file-based process depending on the application. In this process, the application of linear, network or channel coding to source coded and packaged audio-visual media (i.e. the source data) typically follows these basic steps:

1) Partitioning of the source data into `block_count` number of source blocks. The value of `block_count` may be small (e.g. 1) or large depending on the size of the source data and the code type (the specific linear, network, or channel coding scheme) that is applied.

2) Partition each of the source blocks into `block_num_symbols` worth of source symbols, x_i, where *i* is the index of the original source symbol in the source block. Each symbol is of size `block_symbol_size` (in bytes), padding the final symbol with zero-valued bytes if necessary. The value of `block_symbol_size` is typically chosen by the application, for example taking into account the size of the data to be encoded. In a similar way, the value of `block_num_symbols` is typically chosen based on the specific application and the code type used.

3) Based on the code type used, encode the source symbols to create at least `block_num_symbols` worth of `coded_symbol` instances (depending on code type), y_i. The number of coded symbols generated is typically a function of the code type, use case, and network. Each code type will take in certain coding parameters (e.g. `block_count`, `block_num_symbols`) and generate additional coding parameters (e.g. `block_symbol_size`).

A graphical representation of these steps is illustrated by the example in Figure 2. In this example, the original source (media) data file or stream is partitioned into three source blocks. Each source block is further partitioned into eight source symbols $x_{n,i}$, where n is the block index (generally n=0, 1, …, `block_count` - 1) and i = 0, 1, …, `block_num_symbols` - 1. These source symbols are then encoded by one of the supported CMMF code types to produce the resulting coded symbols $y_{n,j}$, $i \leq j$.

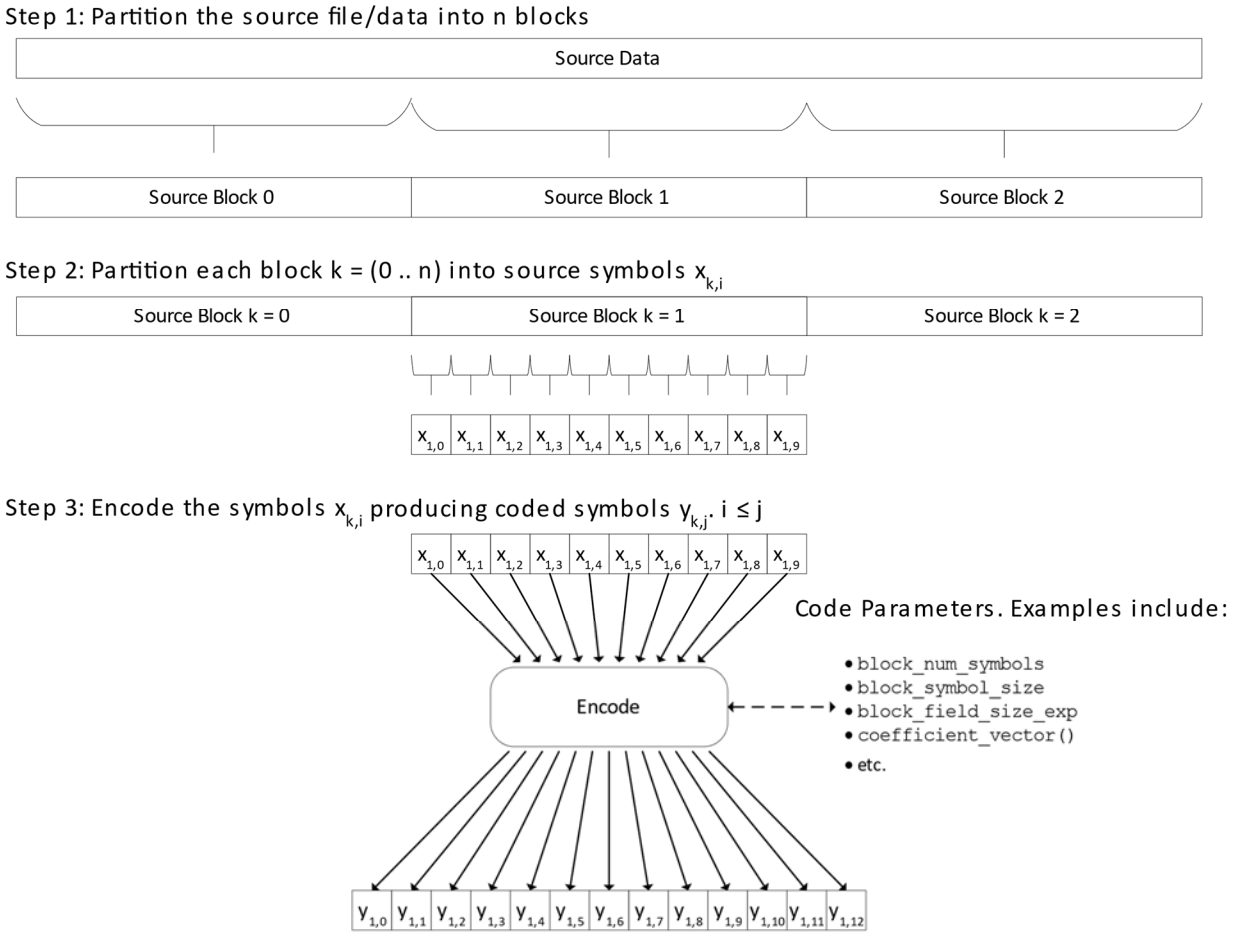

Figure 2: CMMF encode example

The resulting coded symbols and associated coding parameters are formatted per the CMMF syntax as described in clause 5 and clause 6, for storage, interchange, or delivery. The coding parameters are necessary to ensure that a compliant CMMF decoder can reconstruct the original (media) source symbols and the source file. Depending on the type of code utilized, the coding parameters can include an Encoding Symbol Index (ESI), a block index or Source Block Number (SBN), coefficient vector, Pseudo-Random Number Generator (PRNG) seed, etc. along with the `block_num_symbols`, `block_symbol_size`, etc. An implementor may choose the code type and how the source data is partitioned into blocks and symbols. In some cases (for example if the code type is an FEC code/scheme based on IETF RFC 5052 [15]) an implementor may follow the block partitioning algorithm specified in section 9 of IETF RFC 5052 [15] or the partitioning defined in a specific FEC scheme.

CMMF supports multiple methods of data partitioning and coding. In addition to block coding, where the source data and source blocks are available in their entirety at the time of CMMF bitstream generation, CMMF also supports sliding-window constructions for use cases that may have rigid latency requirements and the availability of the source data is variable in both time and size.

Depending on the application, encoding of a CMMF bitstream may or may not be an active process (i.e. real-time). A CMMF bitstream may be generated at the time of a request by an application/client, within a CMMF-aware network delivery protocol, or the generation can occur offline and bitstreams stored for future delivery and/or processing.

4.2.2 Symbol groups in CMMF

Coded/repair symbols (including systematic or source symbols) can be grouped and packaged in different ways within the CMMF bitstream. The symbol grouping can be chosen to best fit the needs of a given application. Example symbol groups include:

- Individual symbol group (i.e. not grouped): Each coded/repair symbol is packaged individually within the bitstream. See also clause 6.1.6.
- Single block symbol group: Coded/repair symbols from the same block can be grouped and packaged collectively within the bitstream. See also clause 6.1.18.
- Multiple block symbol group: Coded/repair symbols from multiple blocks can be grouped and packaged collectively within the bitstream. See also clause 6.1.22.

Example symbol groups are shown in Figure 3.

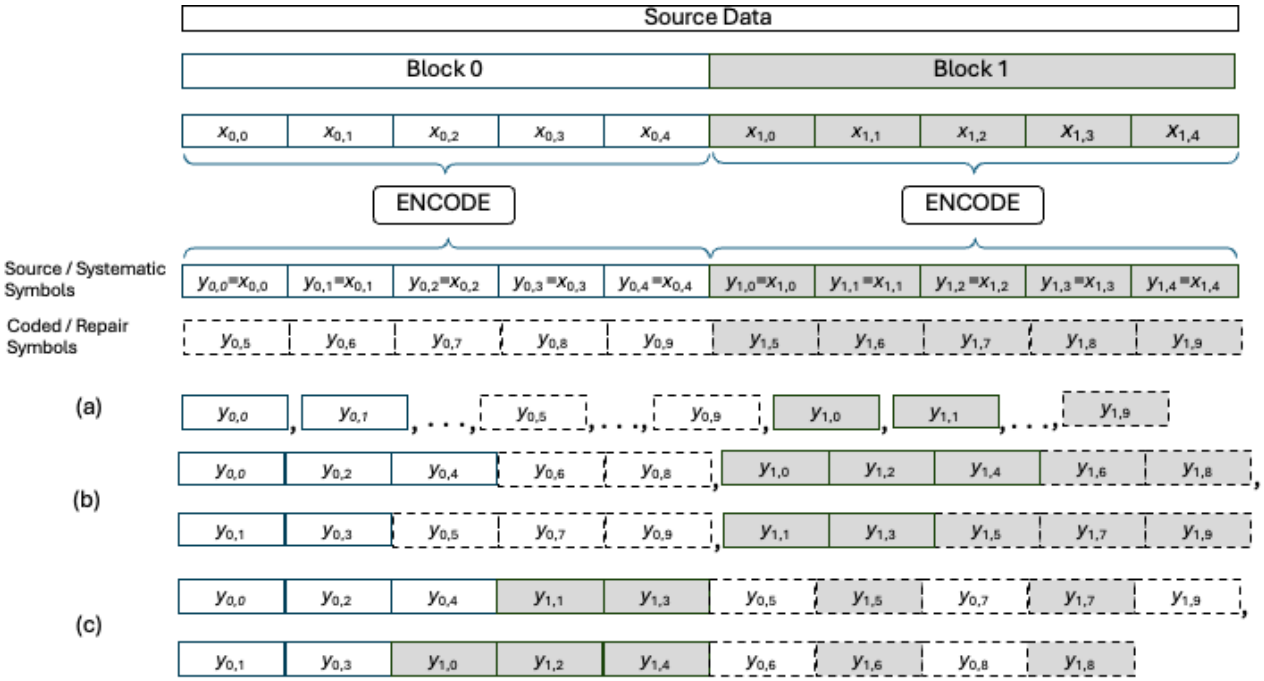

Figure 3: Example symbol groups

In Figure 3, a file is split into two blocks, each partitioned into five symbols, and each encoded to ten coded/repair symbols (including the five source/systematic symbols):

a) Shows each coded/repair symbol as its own individual symbol group.

b) Shows examples of four single block symbol groups.

c) Shows examples of two multiple block symbol groups.

The arrangement of symbols within a symbol group may follow a specific pattern. Symbol groups may be distinct, i.e. each unique coded/repair symbol is added exactly once to exactly one *symbol group*. Symbol groups may also have a property of *completeness*, i.e. the source object can be recovered from all symbols in the symbol group.

4.2.3 Decoding CMMF

The CMMF bitstream specifies and enables carriage and signalling of essential coding parameters required for reconstruction of the original source symbols in a decoder.

The basic steps for decoding include:

1) For each block, decode the received coded symbols $y_{n,j}$ recreating the source symbols $x_{n,i}$.

2) Concatenate each source symbol $x_{n,i}$ and removing any zero-padding from the last symbol if necessary to reform the original source block.

3) Concatenate each of the reassembled source blocks to reform the original source coded media file/data.

These steps are illustrated in Figure 4.

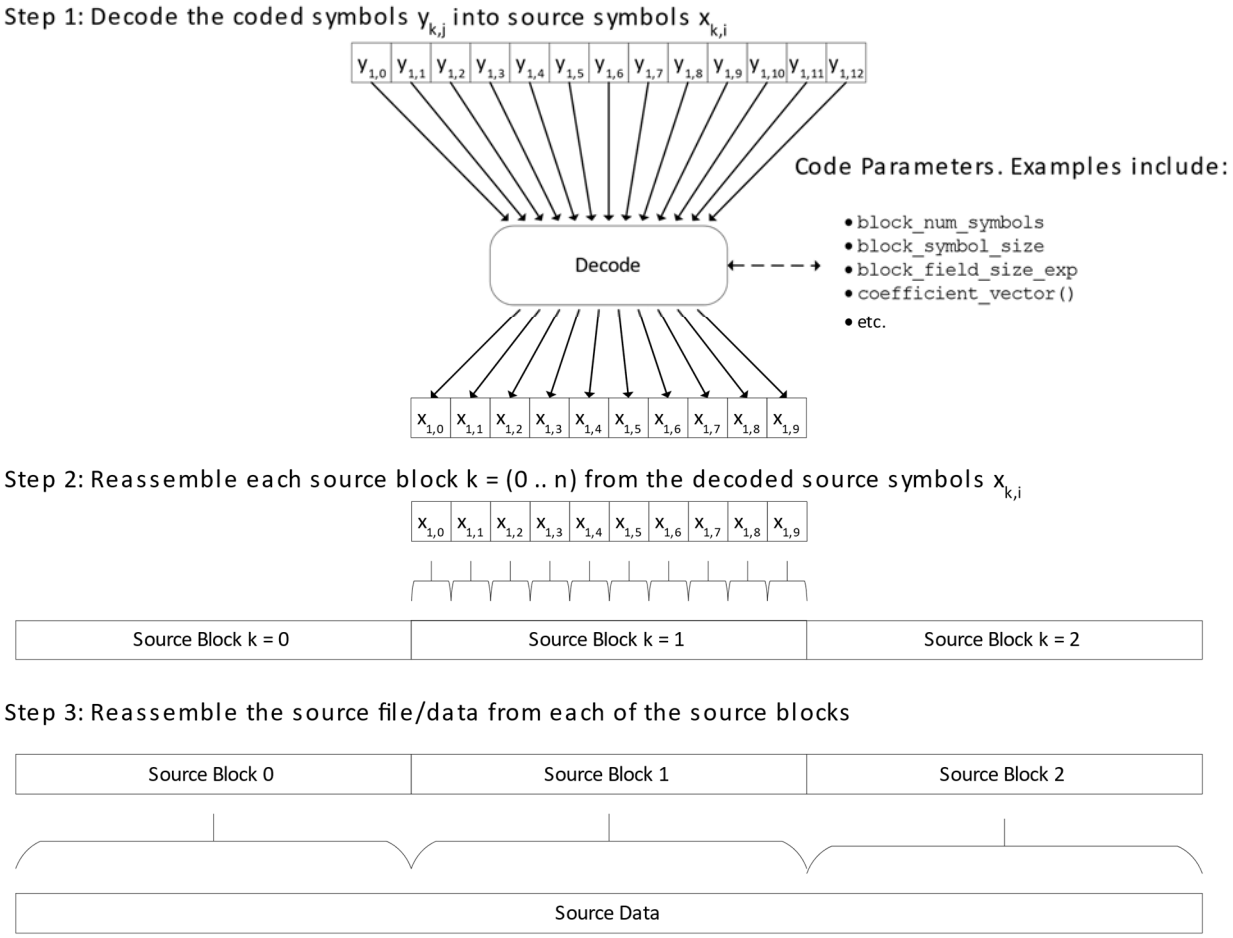

Figure 4: CMMF decode example

4.2.4 Mapping to/from CMMF

The coding CMMF employs is essentially a mapping of information from one form to another to enable additional methods and approaches to communicate or store that information across a network. CMMF supports multiple code types to help ensure applicability in different use cases. A CMMF encoder and decoder may include native support for one or more code types. For some code types, existing and/or standardized encoders, decoders, and bitstream formats may already exist. To improve interoperability and make use of CMMF as an interchange and container format, in some applications a CMMF encoder may contain a bitstream mapper used to translate the coding parameter(s) and coded symbol(s) information produced by existing/standardized linear, network or channel encoders to the equivalent bitfield information and format(s) in the CMMF bitstream as defined in clause 6. Conversely, a receiver, upon parsing a CMMF bitstream, may be required to implement a de-mapping function to translate the relevant CMMF bitfield information back into the original bitfield syntax or format to use with existing/standardized decoder components. High-level representative functional components for generating and parsing a CMMF bitstream are shown in Figure 5 and Figure 6.

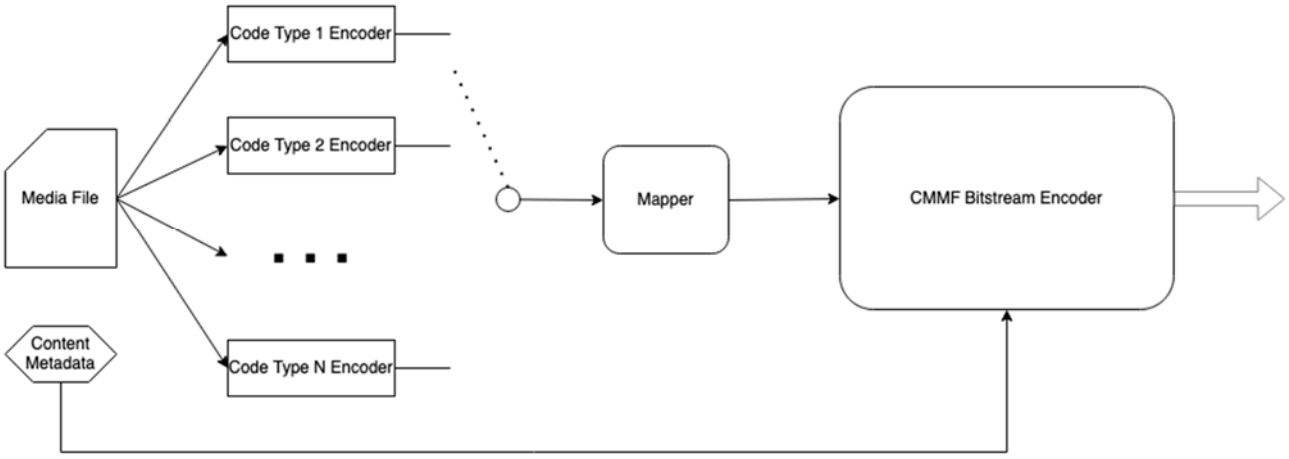

Figure 5: CMMF bitstream encoding/packaging

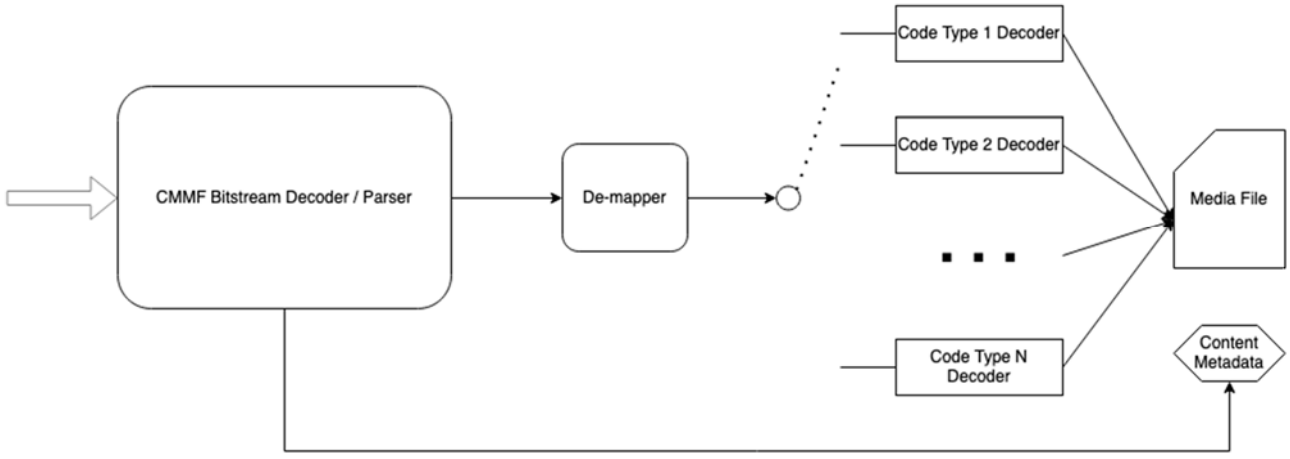

Figure 6: CMMF bitstream decoding

Although shown as a single processing block in the figures above, the mapper and de-mapper functions may be specific for each code type. The code type specific encoders and decoders may be either tightly integrated or only loosely integrated depending on the availability of existing components, it is left for the integrator to determine the level of integration.

As an example of the mapping process when using a Reed-Solomon code type, some of the coding parameters may include:

- Galois Field Size.
- Source Block Number (for each coded symbol).

- Encoding Symbol ID (for each coded symbol).

Table 1 shows a sample mapping to CMMF for these two coding parameters.

Table 1: Example mapping

Original coding parameter	CMMF coding parameter	Mapping function	Relevant clause
Galois Field Size {where size = 2^^m}		`if (m==1) {block_mask = BBBBB0Bb} else {block_field_size_exp = m}`	6.1.5.16, 6.1.5.19
Source Block Number (SBN)	packet_block_index	`packet_block_index = SBN`	6.1.6.1
Encoding Symbol ID (ESI)	packet_symbol_index	`packet_symbol_index = ESI`	6.1.7.4

4.3 Media delivery using CMMF

4.3.1 Overview

CMMF supplements existing methods to stream/deliver media across networks by enabling content to be delivered from multiple sources and/or across multiple network paths. In general, CMMF can be integrated into existing workflows through the addition of a minimal set of new components. All that is required is a CMMF bitstream generator/encoder located at the entry point into or within the network and a CMMF receiver located near or on a media client. CMMF bitstreams created by one or more CMMF bitstream encoders can then be cached and/or delivered from multiple network sources or across multiple network paths using existing network protocols (e.g. HTTP/TCP, webRTC, FLUTE, etc.). Figure 7 shows generic example of some possible delivery/transport scenarios that might occur in a many-to-one multisource application using CMMF. There are six scenarios shown where the creation, storage/caching, and delivery of CMMF bitstreams containing encoded representations of the original source media differ.

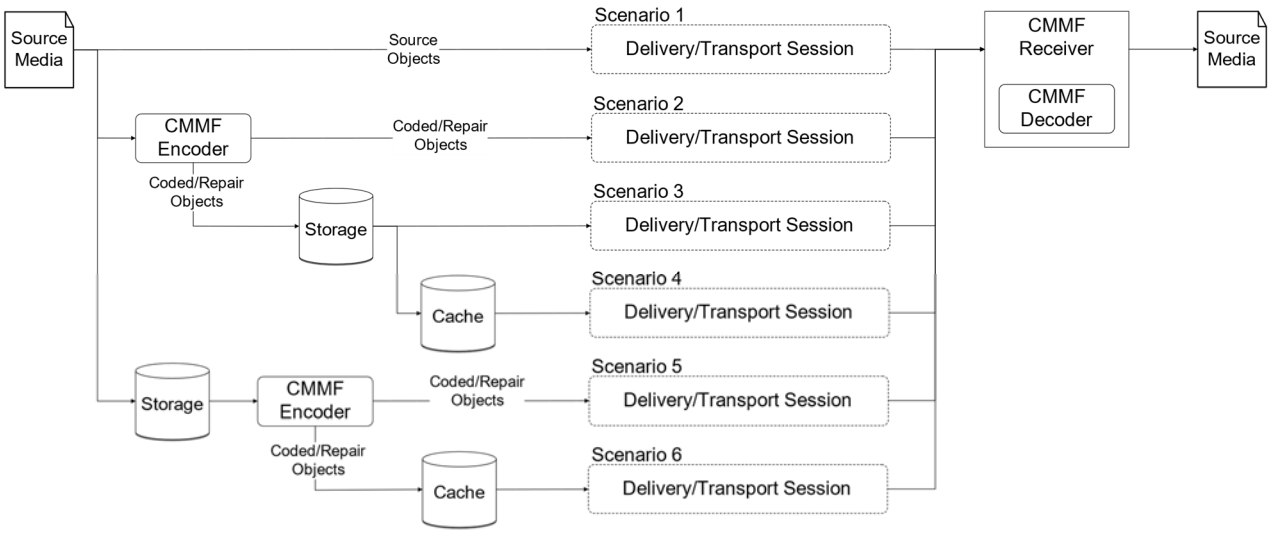

Figure 7: Generic Example CMMF delivery/transport sessions

In Figure 7, objects are delivered, or flow, from the original source media to a CMMF receiver. An object can be a source object (which is essentially the source data), or a coded/repair object (i.e. a CMMF bitstream) generated by a CMMF encoder. In the first scenario, the source object is delivered to the CMMF receiver as would occur in an existing "legacy" application. While not shown, this source object may be stored and/or cached within the network prior to delivery to the CMMF receiver. In the second and fifth scenarios, a real-time CMMF encoder creates a coded representation (i.e. a CMMF bitstream) from the source data which is then transmitted immediately to the CMMF receiver. The third, fourth, and sixth scenarios show a CMMF encoder creating another coded representation of the source, but the output of this CMMF encoder is stored and/or cached where it sits until accessed by a CMMF receiver. Other scenarios and integrations are possible that are not depicted within the diagram, including scenarios where the CMMF encoder and an existing network component (e.g. storage, cache, network protocol) are tightly integrated.

Any single scenario, multiple of a single scenario, or combination of scenarios can be used in the delivery of a source object using CMMF. Objects, whether source and/or coded/repair objects, are collected by the CMMF receiver where they are processed and the decoded yielding the original source media. Further details regarding the delivery/transport of CMMF based on existing use cases and network protocols are provided in annex A, annex B, and annex C.

4.3.2 CMMF transport objects and transport sessions

Objects to be delivered using CMMF in the context of the present document may be organized using transport objects where transport objects are organized in a delivery session composed of one or more transport sessions:

- A transport session consists of one or multiple transport objects. Each transport object can be uniquely identified within the session, for example, by a Transport Object Identifier (TOI).
- Each transport object is assigned to exactly one transport session within the delivery session. Transport sessions can be uniquely identified by a Transport Session Identifier (TSI).
- Transport objects within a transport session are uniquely identified by a TOI and may be ordered.
- Transport objects within a transport session may share common metadata and other properties.
- Transport objects within a transport session may have associated timing, for example media time.
- Transport objects in different transport sessions may contain source and/or coded/repair objects generated from the same source data.

An overview of a CMMF session model is provided in Figure 8. A delivery session consists of *S* transport sessions, each identified by a TSI = 0, …, *S*-1, and each transport session consists of *N*[TSI] transport objects, each identified by a TOI = 0, …, *N*[TSI]-1. Note that this model applies if only a single transport session with a single transport object is created.

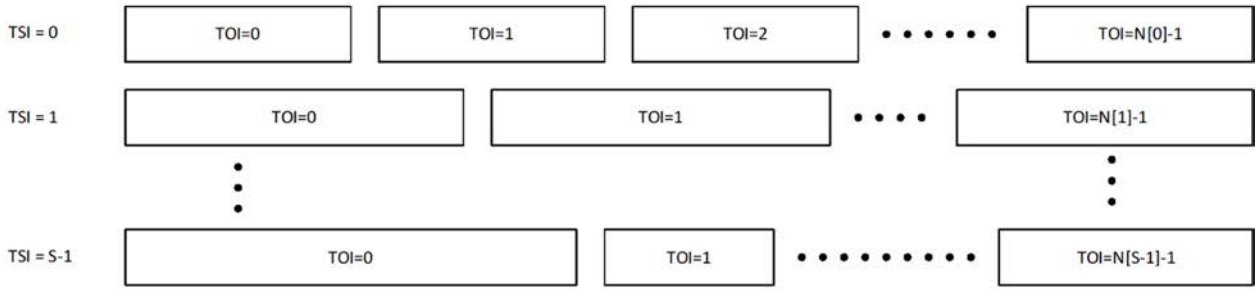

Figure 8: Delivery session model for CMMF

The method used to form transport objects and map them to TSI/TOI combinations within a delivery session is application and use case dependent. Object formation and mapping may be dependent on numerous factors including the formatting of the underlying source data, application requirements, network protocols, content delivery protocols, etc.

4.3.3 CMMF delivery architecture reference points

CMMF provides a general framework for multisource-coded media that can be utilized across many different media delivery architectures and use cases. While many of these architectures and scenarios differ in one way or another, many common concepts and approaches are often shared. Figure 9 provides a notional reference architecture comprising of common components within media delivery architectures and illustrates how CMMF can be integrated within them.

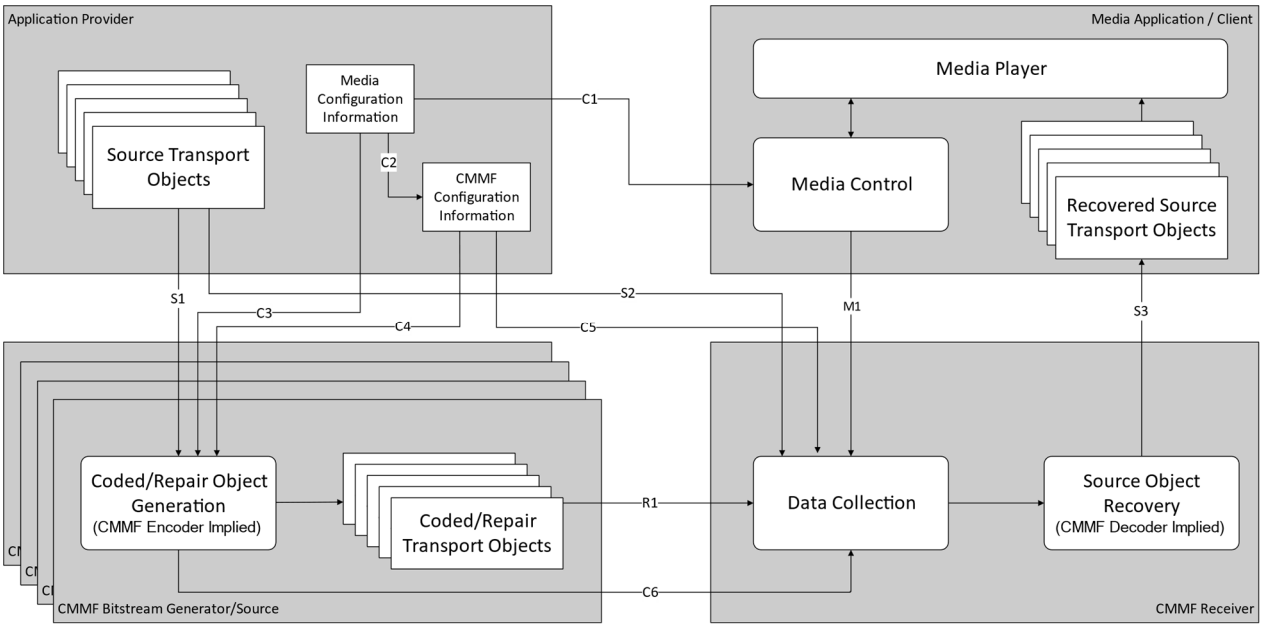

Figure 9: CMMF delivery architecture reference points

Figure 9 shows four major components:

- Application Provider: The application provider is responsible for producing source media and configuring their service. Sub-components within the Application Provider include:

  - Source Transport Objects: Source Transport Objects consist of original source media intended to be delivered to a Media Application/Client. Examples may include CMAF/MPEG-DASH/HLS video/audio segments, etc.

  - Media Information: Depending on the method used to prepare the source media for delivery or the use case, information may be defined that describes the characteristics of that media. Examples may include MPEG-DASH/HLS manifests, etc. This information may include details on how the source transport objects are structured within a delivery session.

  - CMMF Configuration Information: Depending on the use case and delivery architecture, global configuration information necessary for the delivery of source media using CMMF may be defined. This information may include details about how to configure either the CMMF encoder or decoder, provide information about where a CMMF receiver can access CMMF bitstreams, etc. In some use cases, the CMMF Configuration Information may fully include the Media Information or in other cases the CMMF Configuration Information may be used as a supplement to the Media Information. While the reference architecture shows that the Application Provider is aware of CMMF's use and defines the CMMF Configuration Information, use cases and delivery architectures exists where the Application Provider is agnostic to the CMMF's use. In these use cases and delivery architectures, the CMMF Configuration Information may be defined by a network provider or other entity responsible for delivering the Application Providers' media. The present document defines one instantiation of CMMF Configuration Information based on an Extended File Delivery Table (EFDT).

- CMMF Bitstream Generator/Source: CMMF Bitstream Generators/Sources are responsible for generation of CMMF bitstreams containing coded representations of Source Transport Objects. Depending on the use case and/or network architecture, one or more CMMF Bitstream Generators/Sources may be used. Their location(s) may be centralized or distributed across a network. In some deployments, a single CMMF Bitstream Generator/Source may be collocated with the Application Provider and create every CMMF bitstream used in delivery of the required media. In other deployments, CMMF Bitstream Generators/Sources may be located at each PoP within the network where content is cached and only responsible for creating CMMF bitstreams intended to be cached only on that specific PoP. These generators/sources may also create Coded/Repair Objects as part of an active process (i.e. in real-time) or off-line as the use case/delivery architecture dictates.

- CMMF Receiver: CMMF Receivers are responsible for obtaining one or more CMMF bitstreams created by one or more CMMF Bitstream Generators/Sources, decoding those CMMF bitstreams, and recovering the original source media. A CMMF receiver does not necessarily require any one CMMF bitstream in its entirety. Rather, it requires enough information from all received bitstreams to be able to decode, at which point delivery of the remainder of any outstanding CMMF bitstream(s) may be terminated. The CMMF receiver may also use parts of source objects together with CMMF bitstreams to recover the entire source objects. The CMMF Receiver may be collocated with the Media Application/Client or located elsewhere in the network at a point where multisource/multipath delivery is no longer desired. The CMMF Receiver may require Media Information, CMMF Configuration Information, and/or information from the Media Application/Client to obtain the required CMMF bitstreams and recover the associated source media. The CMMF Receiver outputs recovered Source Transport Objects. Depending on the implementation of the interface between the CMMF Receiver and Media Application/Client, these recovered Source Transport Objects may be cached within the CMMF Receiver until the media application requires them or they may be immediately delivered to the media application as they are recovered.

- Media Application/Client: The Media Application/Client is responsible for playback of the recovered source media. In some use cases, the Media Application/Client may control which source media is selected (using the Media Configuration Information) for playback. In these cases, the Media Application/Client may provide input via a Media Control process to the CMMF Receiver.

Figure 9 also shows reference points between the components listed above. Depending on the use case and delivery architecture, these reference points may or may not be utilized:

- C1: This reference point is included as reference only and is not intended to replace or replicate the process other protocols (e.g. MPEG-DASH, HLS, etc.) use to transfer media information between an Application Provider and Media Application/Client. It describes the Media Information that is sent from the Application Provider to the Media Application/Client. This information generally describes the source media; and an example may include a MPEG-DASH or HLS manifest.

- C2: This reference point describes the Media Information used in defining the CMMF Configuration Information. For example, this information may be used to define source blocks within CMMF bitstreams based the structure defined in the Media Information.

- C3: This reference point describes the Media Information that is sent to and used by the CMMF Bitstream Generator/Source to produce coded/repair transport objects.

- C4: This reference point describes the CMMF Configuration Information that is sent to and used by the CMMF Bitstream Generator/Source. This information may be used to configure the CMMF Bitstream Generator/Source and/or used in the production of coded/repair transport objects.

- C5: This reference point describes the CMMF Configuration Information that is sent to and used by the CMMF Receiver. This information may contain details not otherwise specified within clause 5 and clause 6. For example, it may contain information that is used by the CMMF Receiver for purposes such as transport object discovery, how Coded/Repair Transport Objects are formed, system operation, etc.

- C6: This reference point describes the Media and/or CMMF Configuration Information that is sent by the CMMF Bitstream Generator/Source to the CMMF receiver. Unlike the other interfaces, this interface may contain configuration information that is specific to a given transport session and/or network protocol. An example may include the delivery of an EFDT.

- S1: This reference point provides Source Transport Objects between the Application Provider and one or more CMMF Bitstream Generators/Sources. These objects are unmodified from the original media/data. These objects may be used in the creation of Coded/Repair Transport Objects encapsulated within one or more CMMF Bitstreams.

- S2: This reference point provides Source Transport Objects between the Application Provider and the CMMF Receiver. These objects are unmodified from the original media/data.

- S3: This reference point provides Source Transport Objects between the CMMF Receiver and the Media Application/Client. These objects may have been obtained by the CMMF Receiver from reference point S2 or recovered from Coded/Repair Transport Objects received over reference point R1.

- R1: This reference point provides Coded/Repair Transport Objects between one or more CMMF Bitstream Generators/Sources and the CMMF Receiver. Details on the formats used to communicate these transport objects are provided in clause 5 and clause 6.
- M1: This reference point provides control and/or configuration information from a Media Application/Client to the CMMF Receiver. For example, this control/information may include details regarding which Source and/or Coded/Repair Transport Objects the CMMF Receiver should obtain.

Examples of deployment options along with other details on the CMMF sender and CMMF receiver architectures are provided in annex D.

4.3.4 CMMF delivery procedure

The procedure for delivering source media using CMMF is highly dependent on the use case, delivery architecture, and network protocols utilized. As a result, defining one generic procedure is not possible. However, Figure 10 provides an example delivery procedure for HTTP-based adaptive streaming scenarios.

NOTE: Other delivery procedures are also possible for the same scenario.

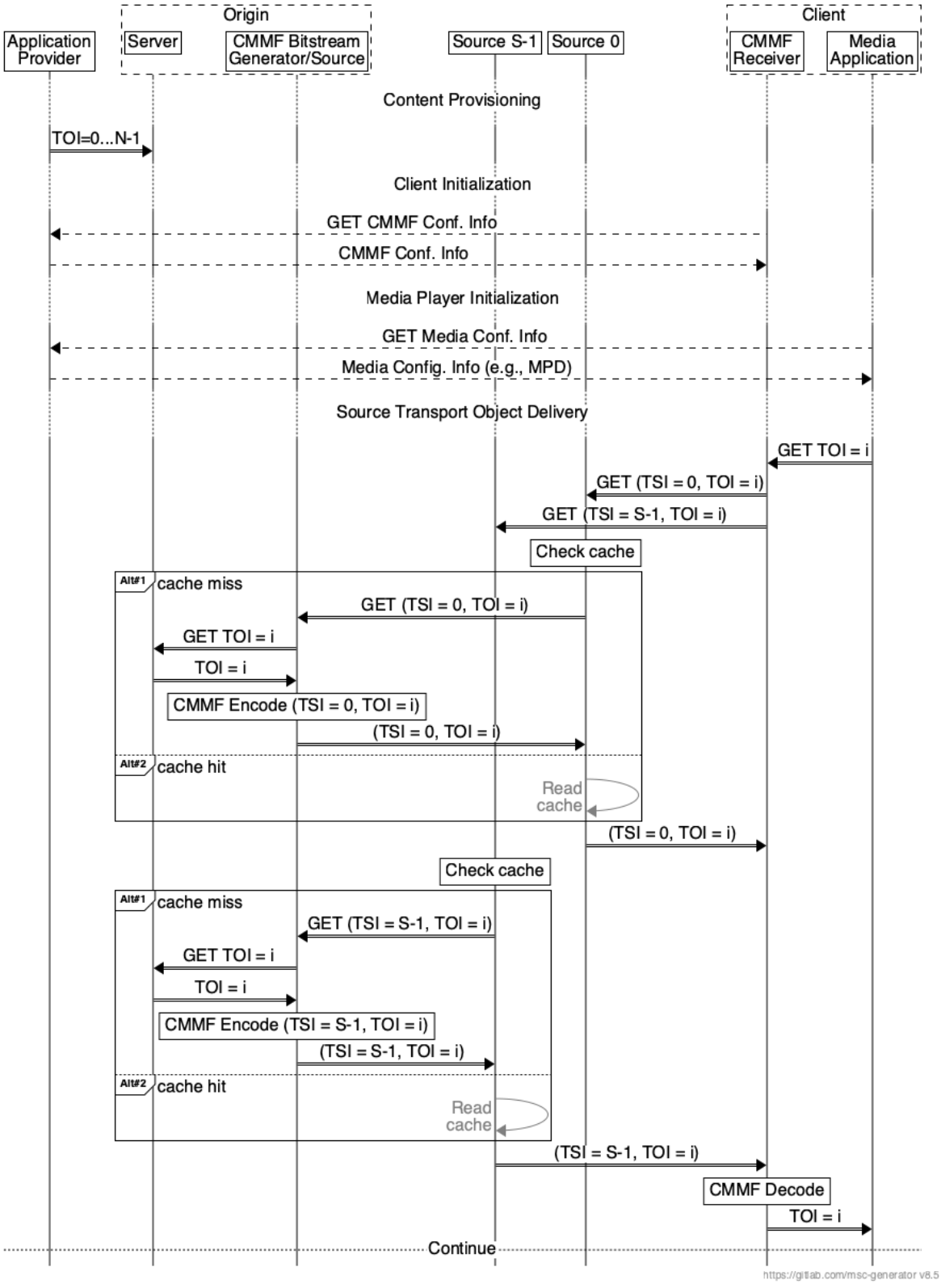

Figure 10: HTTP-Based Adaptive Streaming CMMF Delivery Procedure Example

Within this example, an Application Provider encodes, prepares, and packages source media into an MPEG-DASH or HLS stream. This results in multiple segmented representations of the source media at different bit rates and/or resolutions along with a MPEG-DASH or HLS manifest. Each segment is assumed to be a unique source transport object with a unique TOI. This source media is streamed to clients using the following procedure:

Content Provisioning:

- The Application Provider in this example provisions an origin server with the source transport objects thereby making the source media available to Media Applications/Clients.
- The Application Provider publishes the Media Configuration Information (e.g. MPEG-DASH or HLS manifest).
- The Application Provider establishes global settings CMMF (e.g. CMMF code type, coded/repair transport object to source transport object mappings, source locations, etc.) for how to encode and distribute CMMF bitstreams to clients. These settings are stored within the CMMF Configuration Information.

Media Application/Client Initialization:

- During initialization, a Media Application/Client will obtain all or a subset of the CMMF Configuration Information from the Application Provider. This CMMF Configuration Information, at a minimum, provides the Media Application/Client information about where to obtain source and/or coded/repair transport objects once a media selection has been made.

Media Player Initialization:

- Upon selection of the media to be played, the media player is initialized. As part of this process, the media player obtains the Media Configuration Information (e.g. MPEG-DASH or HLS manifest) from the Application Provider.

Source Transport Object Delivery:

- Once the Media Application/Client has obtained the Media Configuration Information, it selects the source transport objects it wants to fetch and play. These source transport objects are requested from the CMMF Receiver. In the case of this example, source transport object TOI = i is requested.
- The CMMF Receiver replicates the request for TOI = i across S transport sessions where each transport session provides a unique source and/or coded/repair transport objects created from source transport object TOI = i. Each transport session is terminated at one of S sources (e.g. cache, CDN PoP, etc.).
- Each source s determines whether the transport object TOI = i for transport session TSI = s is cached locally. If so (i.e. a cache hit), it replies to the CMMF Receiver with the necessary transport object. If not, it obtains the transport object for the transport session from the CMMF Bitstream Generator/Source.
- In this example, the CMMF Bitstream Generator/Source is collocated with an origin server that stores the original source transport object TOI = i. Upon a request from source s (i.e. a cache miss), the CMMF Bitstream Generator/Source obtains the source transport object TOI = i from the origin server, encodes the transport object associated with TSI = s and TOI = i, and responds to the request from source s.
- Once the CMMF receiver obtains enough information from the combination of source and/or coded/repair transport objects from the different transport sessions, it decodes and recovers the original source transport object TOI = i and responds to the Media Application/Client.
- This process repeats for each source transport object requested by the Media Application/Client.

4.3.5 CMMF Configuration Information

Additional information may need to be communicated between components of the reference CMMF delivery architecture shown in clause 4.3.3 to enable multi-source delivery using CMMF. In general, this information is specific to the operation of CMMF and intended to supplement existing information defined for the media delivery protocol (e.g. MPEG-DASH, HLS, etc.) in use. The method used to communicate this information is dependent on the use case and/or scenario. In some implementations, this information can be transmitted via a configuration file or supplied via an API. In other implementations, this information can be provided in the form of an EFDT as described in clause D.2.3. Examples of the type of information that may be contained with the CMMF Configuration Information include:

- code type the CMMF encoder should use;
- host URLs of the locations for each source where CMMF encoded content is available;
- arrangement of symbols that are included in each of the transport objects;
- etc.

Specific examples of different versions of CMMF Configuration Information are provided in annex B, annex C and annex D.

4.3.6 CMMF as a Content Delivery Protocol

As described in clause 4.3.1, CMMF provides a means to generate coded/repair transport objects for transport/delivery sessions. In one instantiation, the CMMF coded/repair framework can be based on the Forward Error Correction (FEC) Building Block as defined in IETF RFC 5052 [15], CMMF can be considered as a Content Delivery Protocol (CDP) specification as defined in clause 8 of IETF RFC 5052 [15]. When operating as a CDP, additional requirements on CMMF apply in order to support applications requiring reliable delivery according to IETF RFC 5052 [15].

The CMMF CDP Framework, example CDP CMMF instantiations, and alignment to the File Delivery over Unidirectional Transport (FLUTE) protocol [18] are further described in clause D.2.

4.4 Overview of the Specification

The remainder of the present document defines a bitstream syntax for communicating multi-source encoded media and provides examples of delivery architectures that utilize CMMF for multi-source delivery:

- Clause 5 and clause 6 provide the bitstream's syntax and semantics respectively. These clauses define the syntax used on interface R1 of Figure 9.
- Clause 7 provides additional information that should be considered in the design of CMMF encoders, parsers, and decoders.
- Annex A provides the normative definition of the xCD-1 code type that can be used within CMMF to encode media for multi-source delivery; and
- Annex D provides a normative mapping between CMMF and the principles defined in IETF RFC 5052 [15]. It also provides an instantiation for a Configuration information, namely an Extended file delivery table, aligned with IETF RFC 6726 [18].
- Finally, annex B, annex C and clause D.2.5 provide examples of possible implementations of CMMF within existing content delivery systems.

5 Bitstream syntax

5.0 Bitstream organization

The syntax specification presented in clause 5.2 shows how the CMMF bitstream (`cmmf_bitstream()`) is the top-level structure of a bitstream. Every bitstream begins with a `sync()` structure, and contains one or more subatom (`subatom()`) structures.

A subatom is a generic container that can be populated with different types of data. Because a subatom can carry only one type of data, the type of data carried within defines a subatom type, and the associated data structure. For example, a bitstream header subatom is an instance of the `subatom()` syntactical structure carrying a `bitstream_header()` syntactical structure. In the present document the subatom type may be referred to by a name (e.g. bitstream header subatom) or its associated syntactical structure (`bitstream_header()`).

Figure 11 shows the high-level hierarchical organization for a CMMF bitstream with N subatoms.

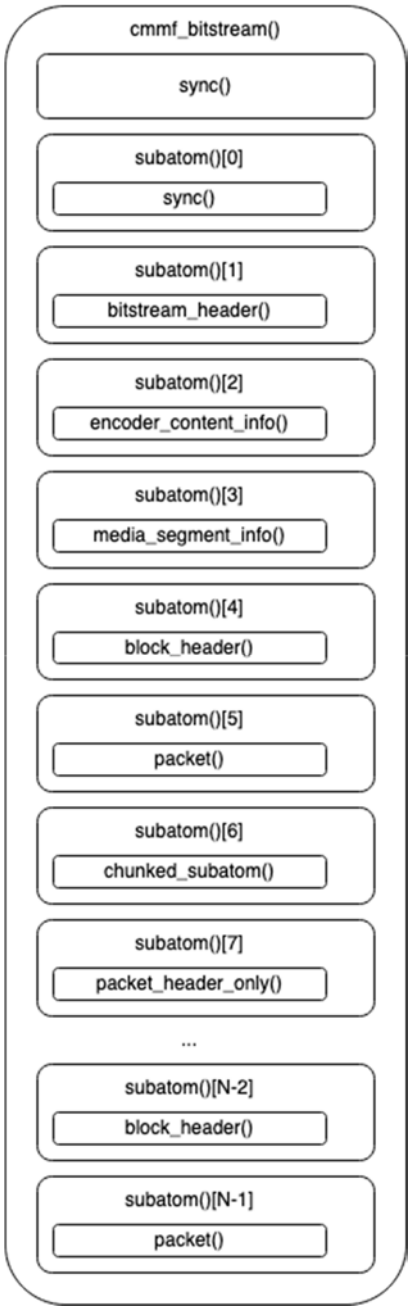

Figure 11: High-level CMMF bitstream organization

There are several types of subatom data types defined for the CMMF bitstream, including:

- Sync (`sync()`)
- Bitstream header (`bitstream_header()`)
- Encoder and content information (`encoder_content_info()`)
- Media segment information (`media_segment_info()`)
- Block header (`block_header()`)
- Packet (`packet()`)
- Chunked subatom (`chunked_subatom()`)
- Block group directory (`block_group_directory()`)
- Packet header only (`packet_header_only()`)
- Packet group (`packet_group()`)
- Multiple block packet group (`multi_block_packet_group()`)

While a bitstream containing a single subatom would be technically valid, a decoder typically needs to receive multiple subatoms, for example receipt of a bitstream header, block header(s), and packet(s) subatoms would allow for successful decode a CMMF bitstream and reconstruction the original source data.

The syntax permits additional subatom data types to be defined for future applications.

5.1 Semantics of syntax specification

5.1.1 Pseudocode syntax

The following pseudocode within syntax boxes describes the order of arrival of information within the CMMF bitstream. This pseudocode is roughly based on C language syntax, but simplified for ease of reading. For bitstream elements that are larger than one bit, the order of the bits in the serial bitstream is either most-significant-bit-first (for numerical values), or left-bit-first (for bit-field values). Fields or elements contained in the bitstream are indicated with **bold** type. Syntactic elements are typographically distinguished by the use of a different font (e.g. `block_count_minus1`).

5.1.2 Bitstream variable syntax

The variable syntax is described in the format of a three-column table in the present document.

In these tables, first column entries in **bold** signify the occurrence of the corresponding variable in the bitstream; the second column specifies the bit-field length and way of encoding (according to the notation in clause 5.1.9); the third column provides (where applicable) a reference to a clause containing more information about the variable.

Table 2 provides an example of the bitstream variable syntax format for the **block_count_minus1** metadata field.

Table 2: Example of bitstream variable

Syntax	Encoding	Clause
`code_type`	u(4)	6.1.4.5

5.1.3 Bitstream structure syntax

Bitstream structures (also referred to as elements) are defined using the C procedure syntax.

The substructure is expanded by the C call syntax name (`<arguments>`).

Table 3 shows an example of the bitstream structure syntax format.

Table 3: Example of bitstream structure

Syntax	Encoding	Clause

`name(<optional arguments>)`
`{`
`  .`
`  .`
`}`		

5.1.4 Iteration and conditional operators

Iteration is expressed using the usual C statements `for`, `while`, and `do...while`. Conditional expansion uses these constructs:

- if (…) ...
- if (…) ... else ...
- if (…) ... else if () ...

These examples demonstrate the procedure of reading a flag from the stream and performing an `<action>` if the flag is `set` (binary 1).

Table 4 provides an example of a shorthand `if` syntax format.

Table 4: Example of shorthand If syntax format

Syntax	Encoding	Clause
`if (`**`flag`**`) <action>`	b(1)	

Table 5 provides an example of an expanded if syntax format, which is the expansion of the preceding shorthand `if` syntax format.

Table 5: Example of expanded If syntax format

Syntax	Encoding	Clause

`flag`
`if (flag) {`
`  <action>`
`}`
	b(1)	

5.1.5 Boolean operations

The Boolean operators `&&` and `||` follow normal C conventions.

For example, for an `&&` operation, the right-hand side is not evaluated if the left-hand side is FALSE, as listed in Table 6.

Table 6: Example of Boolean operation syntax format

Syntax	Encoding	Clause

`if (paramA &&` **`paramB`**`) {`
`  <action>`
`}`
	b(1)	

5.1.6 Labels and comments

Labels are flush against the left margin and are indicated as comments in the code.

Table 7 shows an example label:

Table 7: Example of bitstream label

Syntax	Encoding	Clause
`/* Label */`		

Labels are regarded as markers within the expanded bitstream, not as textual markers within the syntax description.

Comments are indented and aligned with associated syntax.

Table 8 shows an example comment:

Table 8: Example of bitstream comment

Syntax	Encoding	Clause

`if (flag)`
`{`
`  /* Comment */`
`  **field**`
`}`		

5.1.7 Operational variables not in the bitstream

Some operational variables that are not present in the bitstream but are useful for performing certain operations (such as parsing the bitstream) are presented in not bold, as shown in the example in Table 9.

Table 9: Example of operational variable format

Syntax	Encoding	Clause
`block_count = block_count_minus1 + 1`		

5.1.8 Arrays

Some bitstream elements naturally form arrays. Even if bitstreams are naturally included in arrays, this syntax specification treats all bitstream elements individually. Where appropriate, arrays are described as multiple elements (for example, `block_header_subatom_offset[blk]`) within control structures, such as for loops, that are employed to increment the index (`blk` for block in this example).

5.1.9 Bit field encoding

The encoded data can be thought of as a serial bitstream containing items of data that occupy fields of varying length (for example, three bits to specify a number in the range [0…7]). The bits occupying each field are presented with the most-significant bit first.

Because fields can be of variable length, they can generally straddle word boundaries. However, the complete CMMF bitstream and some of its substructures are aligned with 8-bit word (or byte) boundaries.

Values are encoded in signed or unsigned bit fields and are referred to in the syntax description using the notation listed in Table 10.

Table 10: Bit field descriptors

Bit field encoding	Description
b(n)	Boolean, n bit field: `1` is `TRUE`, `0` is `FALSE`.
v(n)	Bit field taking n bits, with arbitrary representation.
s(n)	Signed integer taking n bits (n ≥ 1), represented in two's complement.
u(n)	Unsigned integer taking n bits (n ≥ 0, u(0) = 0).
pad(n)	Pad with n zero bits.
pad	Pad with an arbitrary number of zero bits. The number is determined by a specified variable.

If a Boolean bitstream element is defined to signal a *whether statement* then the *statement* is true if the Boolean is set (TRUE), i.e. the corresponding bit is set to `1`. A subsequent *otherwise statement* is true if the Boolean is cleared (FALSE), i.e. the corresponding bit is set to `0`.

Bit fields indicated as `reserved` are encoded as v(n) with all bits set to `0` and shall be ignored by bitstream parsers that conform to this version of CMMF.

When present in the syntax, the `byte_align` field uses the `pad` bitfield encoding with 0 to 7 zero bits. The field ensures that there exists a whole number of bytes relative to a reference starting point.

5.2 Syntax specification

5.2.1 cmmf_bitstream()

Table 11: Syntax of cmmf_bitstream()

Syntax	Encoding	Clause

`cmmf_bitstream()`
`{`
`  sync()`
`  while (true)`
`  {`
`  subatom()`
`  }`
`}`
		6.1.1
		
		5.2.3
		
		5.2.2

5.2.2 subatom()

Table 12: Syntax of subatom()

Syntax	Encoding	Clause

`subatom()`
`{`
`/* subatom header start */`
`  **subatom_id**`
`  if (subatom_id == 0xF){`
`  **subatom_id_ext**`
`  subatom_id = subatom_id + subatom_id_ext`
`  }`
 	
`  **b_bitstream_id_present**`
`  **reserved**`
 	
`  sas_bits = num_bits_code()`
 	
`  if (b_bitstream_id_present){`
`  **bitstream_id**`
	
	u(4)
	
	u(8)
	
	b(1)
	v(1)
	
	v(16)
		6.1.2
		
		6.1.2.1
		
		6.1.2.1
		
		6.1.2.2
		
		6.1.2.3,
		5.2.23
		
		6.1.2.4

Syntax	Encoding	Clause
}		
subatom_size	u(sas_bits)	6.1.2.5
/* subatom header end */		
/* subatom data start */		
switch(subatom_id)		
{		
case SUBATOM_ID_SYNC:		
sync()		5.2.3
break		
case SUBATOM_ID_BITSTREAM_HEADER:		
bitstream_header()		5.2.4
break		
case SUBATOM_ID_ENCODER_CONTENT_INFO:		
encoder_content_info()		5.2.10
break		
case SUBATOM_ID_MEDIA_SEGMENT_INFO:		
media_segment_info()		5.2.11
break		
case SUBATOM_ID_BLOCK_HEADER:		
block_header()		5.2.5
break		
case SUBATOM_ID_PACKET:		
packet()		5.2.8
break		
case SUBATOM_ID_CHUNKED_SUBATOM:		
chunked_subatom()		5.2.13
break		
case SUBATOM_ID_BLOCK_GROUP_DIRECTORY:		
block_group_directory()		5.2.14
break		
case SUBATOM_ID_PACKET_HEADER_ONLY:		
packet_header_only()		5.2.19
break		
case SUBATOM_ID_PACKET_GROUP:		
packet_group()		5.2.21
break		
case SUBATOM_ID_MULTI_BLOCK_PACKET_GROUP:		
multi_block_packet_group()		5.2.25
break		
default:		
UNKNOWN	v(subatom_size × 8)	
}		
padding	pad	
/* subatom data end */		
}		

5.2.3 sync()

Table 13: Syntax of sync()

Syntax	Encoding	Clause
sync()		6.1.3
{		
syncword	v(64)	6.1.3.1
version	v(4)	6.1.3.2
b_content_encode_uuid	b(1)	6.1.3.3
reserved	v(3)	
if (b_content_encode_uuid) {		
content_encode_uuid	v(128)	6.1.3.4
}		
}		

5.2.4 bitstream_header()

Table 14: Syntax of bitstream_header()

Syntax	Encoding	Clause
bitstream_header() {		6.1.4
content_source_size	u(64)	6.1.4.1
content_source_type	v(3)	6.1.4.2
reserved	v(1)	
b_content_source_split	b(1)	6.1.4.3
if (b_content_source_split){		
content_source_split_start	u(64)	6.1.4.4
content_source_split_end	u(64)	6.1.4.4
}		
code_type	u(4)	6.1.4.5
if (code_type == 0xF){		
code_type_ext	u(8)	6.1.4.5
code_type = code_type + code_type_ext		
}		
b_rfc5052	b(1)	6.1.4.6
if (b_rfc5052){		
rfc5052_information()		5.2.20
}		
block_count_minus1 = block_index_or_count_value()		6.1.4.7, 5.2.24
block_count = block_count_minus1 + 1		6.1.4.7
b_content_block_separate_sources	b(1)	6.1.4.8
if (b_content_block_separate_sources){		
num_content_block_sources_minus1	u(8)	6.1.4.9
}		
b_profile_information_present	b(1)	6.1.4.10
if (b_profile_information_present){		
profile_type_size	u(4)	6.1.4.11
profile_type	v(profile_type_size × 8)	6.1.4.11
profile_description	v(32)	6.1.4.12
}		
b_block_cc_encrypted	b(1)	6.1.4.13
if (b_block_cc_encrypted){		
bitstream_encryption_key_id_size_exp	u(4)	6.1.4.14
bseki_bits = 2^ bitstream_encryption_key_id_size_exp		
bitstream_encryption_key_id	v(bseki_bits)	6.1.4.15
}		
}		

5.2.5 block_header()

Table 15: Syntax of block_header()

Syntax	Encoding	Clause
block_header()		6.1.5
{		
block_index = block_index_or_count_value()		6.1.5.1, 5.2.24
block_size	u(32)	6.1.5.2
block_symbol_size	u(32)	6.1.5.3
bns_bits = num_bits_code()		6.1.5.4, 5.2.23
block_num_symbols	u(bns_bits)	6.1.5.5
b_block_max_symbol_index_present	b(1)	6.1.5.6
if (b_block_max_symbol_index_present)		
{		
bmsi_bits = num_bits_code()		6.1.5.7, 5.2.23
block_max_symbol_index	u(bmsi_bits)	6.1.5.8

Syntax	Encoding	Clause
}		
b_block_content_source_index_present	b(1)	6.1.5.9
if (b_block_content_source_index_present)		
{		
block_content_source_index	u(8)	6.1.5.10
}		
b_block_composite_sources	b(1)	6.1.5.11
if (b_block_composite_sources)		
{		
block_num_composite_sources_minus1	u(8)	6.1.5.12
blk_num_cmp_sources = block_num_composite_sources_minus1 + 1		
for (cmpsrc=0; cmpsrc < blk_num_cmp_sources; cmpsrc++) {		
bcss_bits = num_bits_code()		6.1.5.13, 5.2.23
block_composite_source_size	u(bcss_bits)	6.1.5.14
}		
}		
b_addl_block_coding_info_present	b(1)	6.1.5.15
if (b_addl_block_coding_info_present)		
{		
addl_block_coding_mask	v(3)	6.1.5.16
if (addl_block_coding_mask & 0x1) {		
b_addl_window_info_present	b(1)	6.1.5.17
if (b_addl_window_info_present) {		
addl_window_info() = extension(4)		
}		
}		
if (addl_block_coding_mask & 0x2) {		
b_reserved_block_coding_params_present	b(1)	6.1.5.18
if (b_reserved_block_coding_params_present) {		
reserved_block_coding_params() = extension(4)		
}		
}		
if (addl_block_coding_mask & 0x4) {		
reserved_block_coding_info() = extension(4)		
}		
}		
block_mask	v(6)	6.1.5.19
if (block_mask & 0x1)		
{		
b_sufficient_symbols_present	b(1)	6.1.5.20
if (!b_sufficient_symbols_present)		
{		
bsp_bits = num_bits_code()		6.1.5.21, 5.2.23
block_symbols_present	u(bsp_bits)	6.1.5.22
}		
}		
else		
{		
/* Symbols present count not available */		
}		
if (block_mask & 0x2)		
{		
block_field_size_exp	v(3)	6.1.5.23
}		
else		
{		
/* GF{2^1}, block_field_size_exp_val=1 */		
}		
if (block_mask & 0x4)		

Syntax	Encoding	Clause

{
/* Coding coeff info in packet header subatoms */
}
if (block_mask & 0x8)
{		

block_cc_encryption_info_size_bits_code **byte_align** bcceis_bits = (block_cc_encryption_info_size_bits_code + 1) × 8 **block_cc_encryption_info_size**	v(1) pad u(bcceis_bits)	6.1.5.24.1 6.1.5.24.2 6.1.5.24.3

/* block encryption info start */
reserved
block_cc_encryption_algorithm
block_cc_encryption_mode
block_cce_key_size_exp
bck_bits = 2^block_cce_key_size_exp
block_cce_key
b_addl_block_cce_params_present
if (b_addl_block_cce_params_present)
{
addl_cce_parameters()
}
padding
/* block encryption info end */
	
	v(1)
	v(3)
	v(4)
	u(4)
	
	v(bck_bits)
	b(1)
	
	pad
		
		6.1.5.24.4
		6.1.5.24.5
		6.1.5.24.6
		
		6.1.5.24.6
		6.1.5.24.7
		
		5.2.6

}
if (block_mask & 0x10)
{
prng_parameters()
}
		
		5.2.7

if (block_mask & 0x20)
{
block_integrity() = fb_integrity()
}
}
		
		5.2.15

5.2.6 addl_cce_parameters()

Table 16: Syntax of addl_cce_parameters()

Syntax	Encoding	Clause
addl_cce_parameters() {		6.1.5.24.8

num_addl_cce_params_minus1
for (eprm = 0; eprm < (num_addl_cce_params_minus1 + 1); eprm++)
{
	u(2)	6.1.5.24.1

cce_parameter_type **cce_parameter_size_exp** ccep_bits = 2 ^ cce_parameter_size_exp **cce_parameter**	v(3) u(4) v(ccep_bits)	6.1.5.24.8.2 6.1.5.24.8.3 6.1.5.24.8.3
} }		

5.2.7 prng_parameters()

Table 17: Syntax of prng_parameters()

Syntax	Encoding	Clause
prng_parameters()		6.1.5.25
{		
prng_type	v(3)	6.1.5.25.1
prng_seed_bits_code	v(2)	6.1.5.25.2
ps_bits = 2 ^ (prng_seed_bits_code + 4)		
prng_seed	u(ps_bits)	6.1.5.25.3
prng_density_percentage	u(7)	6.1.5.25.4
}		

5.2.8 packet()

Table 18: Syntax of packet()

Syntax	Encoding	Clause
packet()		6.1.6
{		
if (block_count > 1)		
{		
packet_block_index = block_index_or_count_value()		6.1.6.1, 5.2.24
}		
else		
{		
packet_block_index = 0		6.1.6.1
}		
packet_header()		5.2.9
coded_symbol	v(css_bits)	6.1.6.2
}		

5.2.9 packet_header()

Table 19: Syntax of packet_header()

Syntax	Encoding	Clause
packet_header()		6.1.7
{		
b_systematic_symbol	b(1)	6.1.7.1
packet_mask	v(7)	6.1.7.2
if (packet_mask & 0x1)		
{		
psi_bits = num_bits_code()		6.1.7.3, 5.2.23
packet_symbol_index	u(psi_bits)	6.1.7.4
}		
if (packet_mask & 0x2)		
{		
if (b_systematic_symbol) {		
b_systematic_symbol_encrypted	b(1)	6.1.7.5.1
} else {		
reserved	v(1)	
}		
b_addl_packet_cce_params_present	b(1)	6.1.7.5.2
if (b_addl_packet_cce_params_present) {		
addl_cce_parameters()		5.2.6
}		
}		
if (block_mask & 0x4) {		

Syntax	Encoding	Clause
window_start_index	u(16)	6.1.7.6
window_stop_index	u(16)	6.1.7.6
window_size = (window_stop_index - window_start_index + 1) % 65536		
} else {		
window_size = block_num_symbols		
}		
if (packet_mask & 0x8) {		
prng_parameters()		5.2.7
}		
if (packet_mask & 0x10) {		
coefficient_vector(window_size, block_field_size_exp_val)		5.2.17
}		
if (packet_mask & 0x20) {		
packet_integrity()		5.2.16
}		
if (packet_mask & 0x40){		
packet_header_extension() = extension(5)		5.2.18
}		
byte_align	pad	6.1.7.7
}		

5.2.10 encoder_content_info()

Table 20: Syntax of encoder_content_info()

Syntax	Encoding	Clause
encoder_content_info()		6.1.8
{		
b_encoder_id_present	b(1)	6.1.8.1
if (b_encoder_id_present)		
{		
encoder_uuid	v(128)	6.1.8.2
}		
b_content_id_present	b(1)	6.1.8.4
if (b_content_id_present)		
{		
content_id_type	v(4)	6.1.8.4.1
content_id_size_minus1	u(8)	6.1.8.4.2
content_id	v((content_id_size_minus1+1) × 8)	6.1.8.4.3
}		
b_content_location_present	b(1)	6.1.8.5
if (b_content_location_present)		
{		
content_location_size_minus1	u(11)	6.1.8.6
content_location	v((content_location_size_minus1+1) ×8)	6.1.8.6
}		
b_content_type_present	b(1)	6.1.8.7
if (b_content_type_present)		
{		
content_type_size	u(8)	6.1.8.8
content_type	v(content_type_size×8)	6.1.8.8
}		
b_content_header_present	b(1)	6.1.8.9
if (b_content_header_present)		
{		
content_header_size	u(7)	6.1.8.10
content_header	v(content_header_size×8)	6.1.8.10
}		
b_file_integrity_present	b(1)	6.1.8.11
if (b_file_integrity_present)		
{		
file_integrity() = fb_integrity()		5.2.15
}		

Syntax	Encoding	Clause
b_media_preso_dur_present	b(1)	6.1.8.12
if (b_media_preso_dur_present)		
{		
media_presentation_duration() = cmmf_time()		5.2.12
}		
reserved	v(4)	
}		

5.2.11 media_segment_info()

Table 21: Syntax of media_segment_info()

Syntax	Encoding	Clause
media_segment_info()		6.1.9
{		
media_segment_block_index = block_index_or_count_value()		6.1.9.1, 5.2.24
reserved	v(4)	
b_composite_source_index_present	b(1)	6.1.9.3
if (b_composite_source_index_present)		
{		
media_segment_composite_source_index	u(8)	6.1.9.4
}		
media_segment_index	u(2)	6.1.9.2
if (media_segment_index == 0x3)		
{		
media_segment_index_ext	u(6)	6.1.9.2
media_segment_index = media_segment_index + media_segment_index_ext		
}		
if (**b_asset_name_present**)	b(1)	6.1.9.5
{		
asset_name_size	u(8)	6.1.9.6
for (i = 0; i < asset_name_size; i++)		
{		
asset_name[i]	v(8)	6.1.9.6
}		
}		
segment_tag_mask	v(4)	6.1.9.7
if (segment_tag_mask & 0x1) {		
segment_duration() = cmmf_time()		5.2.12
}		
if (segment_tag_mask & 0x2) {		
segment_start_time() = cmmf_time()		5.2.12
}		
if (segment_tag_mask & 0x4) {		
segidx_bits = num_bits_code()		6.1.9.8, 5.2.23
segment_index	u(segidx_bits)	6.1.9.9
}		
if (segment_tag_mask & 0x8) {		
segcnt_bits = num_bits_code()		6.1.9.8, 5.2.23
segment_count	u(segcnt_bits)	6.1.9.10
}		
if (**b_media_mime_type_present**) {	b(1)	6.1.9.11
media_mime_type_size	u(6)	6.1.9.12
for (i = 0; i < media_mime_type_size; i++) {		
media_mime_type[i]	v(8)	6.1.9.12
}		
}		
if (**b_media_codec_present**) {	b(1)	6.1.9.13
media_codec_size	u(6)	6.1.9.14
for (i = 0; i < media_codec_size; i++) {		
media_codec[i]	v(8)	6.1.9.14

Syntax	Encoding	Clause
} }		
if (**b_bit_rate_present**) {	b(1)	6.1.9.15
bit_rate_bits_code	v(1)	6.1.9.16
bps_bits = (bit_rate_bits_code + 3) × 8		
bit_rate	u(bps_bits)	6.1.9.17
}		
if (**b_ms_content_type_present**) {	b(1)	6.1.9.18
ms_content_type	v(3)	6.1.9.19
b_ms_content_type_info_present	b(1)	6.1.9.20
}		
else		
{		
/* b_ms_content_type_info_present = 0b */		
}		
if (b_ms_content_type_info_present) {		
switch (ms_content_type) {		
case 0x0:		
if (**b_aspect_ratio_present**) {	b(1)	6.1.9.21
sample_aspect_ratio	u(8)	6.1.9.22
if (sample_aspect_ratio == 255) {		
sar_width	u(8)	6.1.9.23
sar_height	u(8)	6.1.9.23
}		
}		
b_dynamic_resolution_video	b(1)	6.1.9.24
if (**b_resolution_present**) {	b(1)	6.1.9.25
resolution_width	u(16)	6.1.9.26
resolution_height	u(16)	6.1.9.26
}		
if (**b_frame_rate_present**) {	b(1)	6.1.9.27
frame_rate	v(5)	6.1.9.28
}		
reserved	v(4)	
if (**b_hdr_info_present**) {	b(1)	6.1.9.29
hdr_compatibility_mask	v(16)	6.1.9.30
if (**b_addl_hdr_info_present**) {	b(1)	6.1.9.31
hdr_compat_mask_index	u(4)	6.1.9.32
hdr_profile	u(8)	6.1.9.33
hdr_level	u(8)	6.1.9.34
hdr_compatibility_id	u(8)	6.1.9.35
}		
}		
if (**b_addl_video_info_present**) {	b(1)	6.1.9.36
addl_video_info = extension(6)		5.2.18
}		
break		
case 0x1:		
if (**b_sampling_freq_present**) {	b(1)	6.1.9.37
b_sampling_freq_is_48k	b(1)	6.1.9.38
if (!b_audio_fs_is_48k) {		
sampling_frequency	v(4)	6.1.9.39
}		
}		
if (**b_audio_config_present**) {	b(1)	6.1.9.40
audio_channel_config	v(24)	6.1.9.41
}		
if (**b_audio_props_present**) {	b(1)	6.1.9.42
reserved	v(1)	
b_virtualized_bin	b(1)	6.1.9.43
reserved	v(2)	
b_object_audio	b(1)	6.1.9.44
if (**b_complexity_index_present**) {	b(1)	6.1.9.45
complexity_index	u(8)	6.1.9.46
}		
}		

Syntax	Encoding	Clause
`if (b_addl_audio_info_present) {`	b(1)	6.1.9.47
`addl_audio_info = extension(6)`		5.2.18
`}`		
`break`		
`default:`		
`if (b_addl_ms_content_type_info_present) {`	b(1)	6.1.9.48
`addl_ms_content_type_info() = extension(6)`		5.2.18
`}`		
`}`		
`}`		
accessibility_mask	v(4)	6.1.9.49
`if (accessibility_mask & 0x1) {`		
language_size	u(6)	6.1.9.50
`for (i = 0; i < language_size; i++) {`		
language[i]	v(8)	6.1.9.48
`}`		
`}`		
`if (accessibility_mask & 0x2) {`		
reserved	v(3)	
`}`		
`if (accessibility_mask & 0x4) {`		
reserved	v(4)	
`}`		
`if (accessibility_mask & 0x8) {`		
`addl_accessibility_info() = extension(4)`		5.2.18
`}`		
`}`		

5.2.12 cmmf_time()

Table 22: Syntax of cmmf_time()

Syntax	Encoding	Clause
`cmmf_time()`		6.1.10
`{`		
b_ddhhmmss_format	b(1)	6.1.10.1
`if (b_ddhhmmss_format) {`		
`if (b_dd_present) (`	b(1)	6.1.10.2
days	u(5)	6.1.10.2
`}`		
`if (b_hh_present) (`	b(1)	6.1.10.2
hours	u(5)	6.1.10.2
`}`		
`if (b_mm_present) (`	b(1)	6.1.10.2
minutes	u(6)	6.1.10.2
`}`		
`if (b_ss_present) (`	b(1)	6.1.10.2
seconds	u(6)	6.1.10.2
`}`		
`} else {`		
int_seconds_bits_code	v(2)	
`is_bits = (int_seconds_bits_code) × 8`		
int_seconds	u(is_bits)	6.1.10.4
`}`		
b_fract_seconds_present	b(1)	6.1.10.5
`if (b_fract_seconds_present) {`		
fract_seconds_bits_code	v(2)	
`fs_bits = 10 × 2^ fract_seconds_bits_code`		
fract_seconds	u(fs_bits)	6.1.10.7
`}`		
`}`		

5.2.13 chunked_subatom()

Table 23: Syntax of chunked_subatom()

Syntax	Encoding	Clause
chunked_subatom()		6.1.11
{		
/* chunk subatom info start */		
chunk_segment_id	u(16)	6.1.11.1
chunk_segment_index	u(24)	6.1.11.2
if (chunk_segment_index == 0)		
{		
num_chunk_segments	u(24)	6.1.11.3
original_subatom_id	u(4)	6.1.11.4
if (original_subatom_id == 0xF)		
{		
original_subatom_id_ext	u(8)	6.1.11.4
original_subatom_id = original_subatom_id + original_subatom_id_ext		
}		
oss_bits = num_bits_code()		6.1.11.5, 5.2.23
original_subatom_size	u(oss_bits)	6.1.11.6
}		
byte_align	pad	6.1.11.7
/* chunk subatom info stop */		
chunk_subatom_segment_data	u(cssd_bits)	6.1.11.8
}		

5.2.14 block_group_directory()

Table 24: Syntax of block_group_directory()

Syntax	Encoding	Clause
block_group_directory()		6.1.12
{		
block_group_dir_mask	v(8)	6.1.12.1
block_count_minus1 = block_index_or_count_value()		6.1.4.7, 5.2.24
if (block_group_dir_mask & 0x1)		
{		
for (block = 0; block < (block_count_minus1 + 1); block++) {		
block_header_subatom_offset[block]	u(64)	6.1.12.2
}		
}		
if (block_group_dir_mask & 0x2)		
{		
for (block = 0; block < (block_count_minus1 + 1); block++) {		
num_packet_groups[block]	u(8)	6.1.12.3
for (pg = 0; pg < num_packet_groups[block]; pg++) {		
if (block_group_dir_mask & 0x4)		
{		
packet_group()		5.2.21
}		
else		
{		
packet_group_index[block][pg]	u(8)	6.1.12.4
}		
}		
}		
for (block = 0; block < (block_count_minus1 + 1); block++) {		
for (pg = 0; pg < num_packet_groups[block]; pg++) {		
packet_group_subatom_offset[block][pg]	u(64)	6.1.12.5
}		
}		
}		

Syntax	Encoding	Clause
`if (block_group_dir_mask & 0x8)`		
`{`		
`num_multi_block_packet_groups`	u(16)	6.1.12.6
`for (mbpg = 0; mbpg <; mbpg++) {`		
`multi_block_packet_group_subatom_offset[mbpg]`	u(64)	6.1.12.7
`}`		
`}`		
`/* directory end */`		
`}`		

5.2.15 fb_integrity()

Table 25: Syntax of fb_integrity()

Syntax	Encoding	Clause
`fb_integrity()`		6.1.13
`{`		
`fb_hash_type`	v(2)	6.1.13.1
`fb_hash_algorithm`	v(3)	6.1.13.2
`fb_hash_size`	v(4)	6.1.13.3
`reserved`	v(6)	
`b_fb_integrity_ext`	b(1)	6.1.13.4
`if (b_fb_integrity_ext)`		
`{`		
`fb_integrity_extension() = extension(3)`		5.2.18
`}`		
`fb_hash`	v(fb_hash_bits)	6.1.13.5
`}`		

5.2.16 packet_integrity()

Table 26: Syntax of packet_integrity()

Syntax	Encoding	Clause
`packet_integrity()`		6.1.14
`{`		
`packet_hash_algorithm`	v(3)	6.1.14.1
`packet_hash_size`	v(3)	6.1.14.2
`reserved`	v(1)	
`b_packet_integrity_ext`	b(1)	6.1.14.3
`if (b_packet_integrity_ext)`		
`{`		
`packet_integrity_extension() = extension(3)`		5.2.18
`}`		
`packet_hash`	v(pkt_hash_bits)	6.1.14.4
`}`		

5.2.17 coefficient_vector()

Table 27: Syntax of coefficient_vector()

Syntax	Encoding	Clause
`coefficient_vector(window_size, block_field_size_exp_val)`		6.1.15
`{`		
`for (index = 0; index < window_size; index++)`		
`{`		
`coded_symbol_coeff[index]`	u(block_field_size_exp_val)	6.1.15.1
`}`		
`}`		

5.2.18 extension()

Table 28: Syntax of extension()

Syntax	Encoding	Clause
extension(num_bits)		6.1.16
{		
extension_byte_size	u(num_bits)	6.1.16.1
for (i = 0; i < extension_byte_size; i++)		
{		
reserved	v(8)	
}		
}		

5.2.19 packet_header_only()

Table 29: Syntax of packet_header_only()

Syntax	Encoding	Clause
packet_header_only()		6.1.17
{		
if (block_count > 1)		
{		
packet_block_index = block_index_or_count_value()		6.1.6.1, 5.2.24
}		
else		
{		
packet_block_index = 0		
}		
packet_header()		5.2.9
}		

5.2.20 rfc5052_information()

Table 30: Syntax of rfc5052_information()

Syntax	Encoding	Clause
rfc5052_information() {		6.1.4.6
encoding_symbol_length	u(32)	6.1.4.6
if (code_type == 5)		
{		
maximum_source_block_length	u(32)	6.1.4.6
max_number_of_encoding_symbols	u(32)	6.1.4.6
}		
b_addl_rfc5052_information_present	b(1)	
if (b_rfc5052_information_present)		
{		
addl_rfc5052_information() = extension(7)		
}		
}		

5.2.21 packet_group()

Table 31: Syntax of packet_group()

Syntax	Encoding	Clause
packet_group()		6.1.18
{		
if (block_count > 1)		
{		
packet_group_block_index = block_index_or_count_value()		6.1.18.1, 5.2.24
}		
else		
{		
packet_group_block_index = 0		6.1.18.1
}		
packet_group_index	u(8)	6.1.18.2
reserved	v(4)	
pgns_bits = num_bits_code()		6.1.18.3, 5.2.23
packet_group_num_symbols	u(pgns_bits)	6.1.18.4
packet_group_type	v(2)	6.1.18.5
if ((packet_group_type == 0x0) || (packet_group_type == 0x1))		
(		
packet_group_header()		5.2.22
}		
if ((packet_group_type == 0x0) || (packet_group_type == 0x2))		
(		
/* start hash */		
for (index=0; index < packet_group_num_symbols; index++)		
{		
coded_symbol	u(block_symbol_size * 8)	6.1.18.6
}		
/* end hash */		
}		
}		

5.2.22 packet_group_header()

Table 32: Syntax of packet_group_header()

Syntax	Encoding	Clause
packet_group_header()		6.1.19
{		
packet_group_symbol_arrangement	v(4)	6.1.19.1
packet_group_mask	v(6)	6.1.19.2
if (packet_group_mask & 0x1)		
{		
reserved	v(32)	
}		
if (packet_group_mask & 0x2)		
{		
switch (packet_group_symbol_arrangement)		
{		
case 0:		
case 1:		
reserved	v(4)	
pgsi_bits = num_bits_code()		6.1.19.3, 5.2.23
for (index=0; index < packet_group_num_symbols; index++)		
{		
packet_group_symbol_index	u(pgsi_bits)	6.1.19.4
}		
break		

Syntax	Encoding	Clause
case 2:		
reserved	v(4)	
pgfsi_bits = num_bits_code()		5.2.23
packet_group_first_symbol_index	u(pgfsi_bits)	
packet_group_index_difference	u(16)	
break		
default:		
reserved	v(2)	
pgsii_bits = num_bits_code()		
packet_group_symbol_index_info = extension(pgsii)		5.2.18
break		
}		
}		
if (packet_group_mask & 0x4)		
{		
b_addl_packet_group_cce_params_present	b(1)	6.1.19.1
if (b_addl_packet_group_cce_params_present)		
{		
addl_cce_parameters()		5.2.6
}		
}		
if (packet_group_mask & 0x8)		
{		
/* start packet_group encryption */		
for (index=0; index < packet_group_num_symbols; index++)		
{		
coefficient_vector(block_num_symbols, block_field_size_exp_val)		5.2.17
}		
/* stop packet_group encryption */		
}		
if (packet_group_mask & 0x10)		
{		
packet_group_integrity() = packet_integrity()		5.2.16
}		
if (packet_group_mask & 0x20)		
{		
packet_group_header_extension = extension(5)		5.2.18
}		
byte_align	pad	
}		

5.2.23 num_bits_code()

Table 33: Syntax of num_bits_code()

Syntax	Encoding	Clause
num_bits_code()		6.1.20
{		
bits_code	u(2)	6.1.20.1
num_bits = (bits_code + 1) × 8		
return num_bits		
}		

5.2.24 block_index_or_count_value()

Table 34: Syntax of block_index_or_count_value()

Syntax	Encoding	Clause
block_index_or_count_value() {		6.1.21
block_index_or_count	u(8)	6.1.21.1
if (block_index_or_count == 0xFF)		
{		
block_index_or_count_ext	u(32)	6.1.21.1
}		
else		
{		
block_index_or_count_ext = 0		6.1.21.1
}		
block_index_or_count_val = block_index_or_count + block_index_or_count_ext		
return block_index_or_count_val		
}		

5.2.25 multi_block_packet_group()

Table 35: Syntax of multi_block_packet_group()

Syntax	Encoding	Clause
multi_block_packet_group() {		6.1.22
mbpg_index	u(16)	6.1.22.1
mbpg_start_block_index = block_index_or_count_value()		5.2.24, 6.1.22.2
mbpg_num_blocks = block_index_or_count_value()		5.2.24, 6.1.22.3
mbpg_num_symbols	u(64)	6.1.22.4
mbpg_header()		5.2.26
/* start hash */		
for (symbol=0; symbol<num_mbpg_symbols; symbol++)		
{		
coded_symbol	u(block_symbol_size × 8)	6.1.22.5
}		
/* end hash */		
}		

5.2.26 mbpg_header()

Table 36: Syntax of mbpg_header()

Syntax	Encoding	Clause
mbpg_header() {		6.1.23
mbpg_symbol_arrangement	v(4)	6.1.23.1
reserved	v(4)	
switch (mbpg_group_symbol_arrangement)		
{		
case 0:		
case 1:		
reserved	v(4)	
mbpgsi_bits = num_bits_code()		5.2.23, 6.1.23.2
for (index=0; index < mbpg_num_symbols; index++)		
{		

Syntax	Encoding	Clause
mbpg_source_block_index = block_index_or_count_value()		5.2.24, 6.1.23.3
mbpg_symbol_index	u(mbpgsi_bits)	6.1.23.3
}		
break		
case 2:		
case 3:		
reserved	v(4)	
mbpg_index_difference	u(16)	6.1.23.6
mbpgfsi_bits = num_bits_code()		5.2.23, 6.1.23.4
mbpg_first_symbol_index	u(mbpgfsi_bits)	6.1.23.7
b_mbpg_is symbol_group_subset	b(1)	
if (b_mbpg_is_symbol_group_subset)		
{		
mbpg_symbol_group_subset_index	u(64)	6.1.23.5
)		
else		
{		
mbpg_symbol_group_subset_index = 0		
}		
break		
default:		
reserved	v(4)	
mbpgsai_bits_code	v(2)	6.1.23.7
mbpgsai_bits = 8 × 2^mbpgsai_bits_code		
mbpg_symbol_arrangement_info = extension(mbpgsai_bits)		5.2.18
break		
}		
b_mbpg_integrity_present	b(1)	6.1.23.9
if (b_mbpg_integrity_present)		
{		
mbpg_integrity() = packet_integrity()		5.2.16
}		
b_mbpg_header_ext_present	b(1)	6.1.23.10
if (b_mbpg_header_ext_present)		
{		
mbpg_header_extension = extension(6)		5.2.18
}		
byte_align	pad	
}		

6 Bitstream description

6.0 Introduction

This clause describes the meaning of the fields in the CMMF bitstream syntax. Bit fields that are similar in purpose and/or meaning have been grouped and described only once.

6.1 Description of bitstream elements

6.1.1 cmmf_bitstream()

The `cmmf_bitstream()` is the top-level structure containing the `sync()` and all `subatom()` instances for the encoded CMMF bitstream.

6.1.2 subatom()

6.1.2.0 Introduction

The subatom (`subatom()`) is a generic structure that represents various types of data present in the CMMF bitstream.

Subatoms inherit values from previously received subatoms with a matching `bitstream_id` (if present the bitstream). For example, the `packet_header()` structure should be able to use `block_field_size_exp` field present in a previously received `block_header()` with the same associated `block_index` / `packet_block_index`.

6.1.2.1 subatom_id, subatom_id_ext

The `subatom_id` field, which is extendable by the `subatom_id_ext` field, identifies the type of data in contained within the subatom. This field indicates which syntactical structure is present in the subatom.

Subatom identification values are provided in Table 37.

Table 37: subatom_id meaning

`subatom_id` value	Associated Constant Identifier	Subatom Data Type
0		Reserved
1	SUBATOM_ID_SYNC	Bitstream synchronization
2	SUBATOM_ID_BITSTREAM_HEADER	Bitstream header
3	SUBATOM_ID_ENCODER_CONTENT_INFO	Encoder and content information
4	SUBATOM_ID_MEDIA_SEGMENT_INFO	Media segment information
5	SUBATOM_ID_BLOCK_HEADER	Block header
6	SUBATOM_ID_PACKET	Packet
7	SUBATOM_ID_CHUNKED_SUBATOM	Chunked subatom
8	SUBATOM_ID_BLOCK_GROUP_DIRECTORY	Block header and packet group directory
9	SUBATOM_ID_PACKET_HEADER_ONLY	Packet header only
10	SUBATOM_ID_PACKET_GROUP	Packet group
11	SUBATOM_ID_MULTI_BLOCK_PACKET_GROUP	Multiple block packet group header
12 - 270		Reserved

If a reserved subatom data type is received, a parser shall skip over the subatom.

While a valid bitstream can contain a single subatom type, decoding requires receipt of multiple subatoms, either in a single bitstream or cumulatively across multiple bitstreams depending on the application.

Although not syntactically hierarchical, subatoms can inherit information from previously received subatom instances. As such, each subatom is not independent, and may require data from a different/previous `subatom()` instance to be parsed and/or decoded correctly. Subatom dependencies are provided in Table 38.

Table 38: Subatom dependencies

Subatom Data Type (`subatom_id`)	Subatom Dependency (`subatom_id`)
Block header (5)	Bitstream header (2)
Packet (6)	Block header (5)
Packet header only (9)	Block header (5)
Packet group (10)	Block header (5)
Multiple block packet group header (11)	Block header (5)

Any additional subatom requirements are described within the definition for each subatom.

6.1.2.2 b_bitstream_id_present

This Boolean indicates whether the `bitstream_id` is present the bitstream. This bit shall be set to the same value in every `subatom()` instance in a `cmmf_bitstream()`.

6.1.2.3 sas_bits

The operational variable `sas_bits` is used in the syntax to specify the number of bits used to encode the subatom data and is represented by the `num_bits_code()` structure.

6.1.2.4 bitstream_id

The `bitstream_id` field identifies different subatoms as belonging to the same bitstream. In applications where simultaneous bitstreams are transmitted, this identification enables a receiver/decoder to determine which subatoms belong to which bitstream.

`bitstream_id`, if present in the bitstream, shall be set to the same value in every `subatom()` instance in the bitstream.

6.1.2.5 subatom_size

The `subatom_size` field describes the size of the subatom data, in bytes. The subatom data is marked by the 'subatom data start' and 'subatom data end' labels. That is, the subatom data type as indicated by the `subatom_id` field, and any additional padding required to ensure that the `subatom_size` is an even number of bytes. Packet subatoms and block directory subatoms shall have no additional padding.

6.1.3 sync()

6.1.3.0 Introduction

The synchronization (`sync()`) structure appears at the beginning of every bitstream and identifies the stream as being a CMMF bitstream. Depending on the application, the `sync()` structure can also be contained within a `subatom()` as needed in order to resynchronize in the middle of a stream.

6.1.3.1 syncword

The `syncword` field is a synchronization word used to identify a CMMF bitstream.

The value of the `syncword` field shall be `0x89780D430044AC31`.

6.1.3.2 version

The `version` field specifies the version of the CMMF bitstream syntax. Encoders conforming to the present document shall use a value of `0` for this field. Bitstream parsers conforming to the present document shall be able to parse version 0. Bitstream parsers conforming to the present document shall reject bitstreams with a version higher than 0.

Bitstream parsers capable of parsing a specific version of the CMMF bitstream shall be able to parse all lesser versions and shall reject all higher versions.

6.1.3.3 b_content_encode_uuid

This Boolean indicates whether the `content_encode_uuid` field is present in the bitstream.

6.1.3.4 content_encode_uuid

The `content_encode_uuid` field identifies a specific encode of a piece of content. This field should carry a version 4, variant 1 universally unique identifier as specified in IETF RFC 9562 [1].

6.1.4 bitstream_header()

6.1.4.0 Introduction

The bitstream header (`bitstream_header()`) structure contains high-level general file information about the encoded data and encoding method.

6.1.4.1 content_source_size

The `content_source_size` field describes the size of the source data, in bytes. If the source data exceeds the size practical for an application, the source data can be split, and multiple bitstreams used.

If the `content_source_size` field is zero, it indicates the source data has a size of 0 bytes, or the size of the source data is not relevant to the application, or the source data is of an undetermined size.

6.1.4.2 content_source_type

The `content_source_type` field describes the source of the encoded data within the bitstream as listed in Table 39.

Table 39: content_source_type meaning

`content_source_type` value	Meaning
`000b`	Not indicated
`001b`	Original data file
`010b`	Reserved
`011b`	Live streaming source
`100b-111b`	Reserved

6.1.4.3 b_content_source_split

This Boolean indicates whether the source data has been split and carried in multiple bitstreams. When True, the `content_source_split_start` and `content_source_split_end` fields are present in the bitstream.

6.1.4.4 content_soure_split_start, content_source_split_end

The `content_source_split_start` and `content_source_split_end` fields describe the start and end byte range of the original source data that is carried in the bitsteam. A parser can use this information to reassemble the original source data.

6.1.4.5 code_type, code_type_ext

The `code_type` field, which is extendable by the `code_type_ext` field, identifies the type of code applied to the data in the bitstream. The different coding types are listed in Table 40. Client rendering devices shall support decoding a minimum of one of the supported code types.

Table 40: code_type meaning

`code _type` value	Meaning
0	xCD-1, as defined in annex A
1	Raptor as defined in IETF RFC 5053 [16]
2	Reserved
3	Reserved
4	Reserved
5	Reed-Solomon (GF{2^8}), as defined in IETF RFC 5510 [14]
6	RaptorQ, as defined in IETF RFC 6330 [13]
7 - 270	Reserved

6.1.4.6 b_rfc5052, rfc5052_information(), b_addl_rfc5052_information_present

This Boolean indicates that CMMF is used as a Content Delivery Protocol (CDP) as defined in IETF RFC 5052 [15]. When True, data shall be partitioned according to the algorithm described in section 9.1 of IETF RFC 5052 [15], and the additional fields contained within the `rfc5052_information()` structure that describe the mandatory, common, and scheme-specific Object Transmission Information (OTI) are present in the bitstream. Some of these additional fields have meaning related to, or identical to, other fields in the bitstream. The OTI fields, their meaning, and related CMMF fields are specified in Table 41.

Table 41: Object Transmission Information Fields

Field	Meaning	Related CMMF Field
`encoding_symbol_length`	See section 6.2.4 of IETF RFC 5052 [15]	`block_symbol_size` (see clause 6.1.5.3)
`maximum_source_block_length`	See section 6.2.4 of IETF RFC 5052 [15]	`block_num_symbols` (see clause 6.1.5.5)
`max_number_of_encoding_symbols`	See section 6.2.4 of IETF RFC 5052 [15]	N/A

When the `b_rfc5052` field is True and the `rfc5052_information()` structure is present, related fields with identical meaning, the `encoding_symbol_length` and `block_symbol_size`, as well as `maximum_source_block_length` and `block_size`, can both be transmitted within the bitstream. In this case, both fields in the shall have the same value.

If the `b_rfc5052` field is True, then blocks shall have identical symbol size (`block_symbol_size`) and the blocks represent the source blocks according to the source blocking of the IETF RFC 5052 [15] FEC scheme.

The mapping from mandatory, common, and scheme-specific OTI parameters to CMMF bitstream elements/fields and values is provided in Table D.2 in clause D.1.4.

The `b_addl_rfc5052_information_present` Boolean indicates whether the `addl_rfc5052_information()` structure is present in the bitstream. The `addl_rfc5052_information()`structure provides a mechanism to transmit additional IETF RFC 5052 [15] information when needed and is an instance of the `extension()` structure.

6.1.4.7 block_count_minus1, block_count

The operational variable `block_count_minus1` which is represented by the `block_index_or_count_value()` structure, describes the total number of blocks present in the bitstream, minus one.

The operational variable `block_count` is used throughout the syntax and is calculated as follows:

```
block_count = block_count_minus1 + 1
```

The equation assumes that the `block_count_minus1` operand has accounted for the `block_count_ext` field when present in the bitstream.

For live streaming applications where the source data stream may continue indefinitely, three options are available. First, the value of `block_index_or_count` in the `block_index_or_count_value()` structure, and thereby the value of `block_count_minus1` may be `0`, meaning there is only a single block present in the bitstream. Other parameters, such as the `window_start_index` and `window_stop_index` fields contained within the packet subatom can be used to differentiate between encoding blocks. Second, the value of `block_index_or_count` in the `block_index_or_count_value()` structure may be `255`. If the `block_index` exceeds 256 blocks, a subsequent bitstream shall be created with the `block_index` reset to `0`. This second option can also be used when an application or code type require more than 256 blocks, e.g. Reed-Solomon coding. Third, the extension `block_index_or_count_ext` in the `block_index_or_count_value()` structure can be used to signal a significant number of additional blocks if necessary.

`block_count_minus1` may appear in multiple subatoms, such as the bitstream header and block directory subatoms. All instances of this operational variable within a bitstream shall be set to the same value.

6.1.4.8 b_content_block_separate_sources

This Boolean indicates whether blocks contain data from separate (but possibly related) sources. For example, block 0 may contain a video segment, while block 1 may contain an audio segment for the same content. When this field is TRUE the `num_content_block_sources_minus1` field is present in the bitstream.

An individual block may be composed of multiple smaller sources (see clause 6.1.5.11 b_block_composite_sources). When a block is composed of multiple smaller sources (i.e. `b_block_composite_sources` is TRUE), for the purposes of setting `b_content_block_separate_sources` (and the related `b_block_content_source_index_present` field in `block_header()`) it shall be considered as having a single source.

For more information on `b_block_content_source_index_present`, see clause 6.1.5.9.

6.1.4.9 num_content_block_sources_minus1

`num_content_block_sources_minus1` describes the number of sources of data that populate the blocks in the bitstream, minus one. The number of sources shall be less than or equal to the number of blocks (indicated by the `block_count` variable) present in the bitstream.

6.1.4.10 b_profile_information_present

This Boolean indicates whether information about the profile that this bitstream conforms to, as indicated by the `profile_type_size`, `profile_type`, and `profile_description` fields, is present in the bitstream.

6.1.4.11 profile_type_size, profile_type

The `profile_type` field is a string of ASCII [7] or UTF-8 [11] characters describing the entity defining the profile. The size (in bytes) of the `profile_type` field is indicated by the `profile_type_size` field.

`profile_type` shall not be set to a value of `0`.

6.1.4.12 profile_description

The `profile_description` field identifies the profile which the bitstream conforms to, as defined by the entity indicated by the `profile_type` field.

A profile may impose constraints on the coding type used, number of blocks, number of symbols or other encoding parameters. Determining whether a bitstream conforms to a specific profile may be useful to applications to quickly determine whether a bitstream is suitable for a specific decoder/parser with restricted or limited capabilities.

6.1.4.13 b_block_cc_encrypted

This Boolean indicates whether the coding coefficient information in one or more blocks is encrypted.

In this case, the `bitstream_encryption_key_id_size_exp` and `bitstream_encryption_key_id` fields are present in the bitstream. The `bitstream_encryption_key_id` can then be used to retrieve the required `bitstream_encryption_key` from a secure/trusted third-party.

6.1.4.14 bitstream_encryption_key_id_size_exp

The `bitstream_encryption_key_id_size_exp` field contains the exponent used to determine the size (in bits) used to transmit the subsequent `bitstream_encryption_key_id`. The size is given by the following equation:

```
bseki_bits = 2^ bitstream_encryption_key_id_size_exp
```

6.1.4.15 bitstream_encryption_key_id

The `bitstream_encryption_key_id` field identifies the key used to symmetrically encrypt certain block encryption parameters.

The identification can be a simple numeric identifier, or for example, a hash of the key.

The encryption method is outside the scope of the present document.

6.1.5 block_header()

6.1.5.0 Introduction

The block header (`block_header()`) structure provides information about how the portion of the data within that block is encoded. It contains fields that describe the coding parameters used for the indicated block.

6.1.5.1 block_index

The operational variable `block_index`, which is represented by the `block_index_or_count_value()` structure, describes which block this `block_header()` instance is associated with.

6.1.5.2 block_size

The `block_size` field describes the number of bytes from the original data the block contains.

A `block_size` value of `0` may be used to indicate that the block size is either not relevant, or the block is empty, or the size is undetermined. A `block_size` value of `0` shall only be used when `block_count` is `1`.

6.1.5.3 block_symbol_size

The `block_symbol_size` field describes the size in bytes of each symbol and coded symbol in the block.

A `block_symbol_size` value of `0` is used to indicate that the coded symbols in the block have a varying size, and the size shall be determined by other means. The `block_symbol_size` shall not have a value of `0` when packet group subatoms are present.

For more information on `coded_symbol`, see clause 6.1.6.2.

6.1.5.4 bns_bits

The operational variable `bns_bits` is used in the syntax to specify the number of bits used to encode the `block_num_symbols` field and is represented by the `num_bits_code()` structure.

6.1.5.5 block_num_symbols

The `block_num_symbols` field describes the number of symbols that the original block of data has been divided into.

A `block_num_symbols` value of `0` may be used to indicate that the block has an unknown number of symbols. A `block_num_symbols` value of `0` shall only be used when `block_count` is `1`.

6.1.5.6 b_block_max_symbol_index_present

This Boolean indicates whether the maximum encoded symbol index (or row index) value for the given block is present in the bitstream. When TRUE, the `block_max_symbol_index` field is present in the bitstream.

6.1.5.7 bmsi_bits

The operational variable `bmsi_bits` is used in the syntax to specify the number of bits used to encode the `block_max_symbol_index` field and is represented by the `num_bits_code()` structure.

6.1.5.8 block_max_symbol_index

The `block_max_symbol_index` field identifies either the value of the maximum symbol index when encoded symbol indexes are used to identify coded symbols, or the value of the maximum row index when a pseudo-random number generator is used to determine the coding coefficients for the block.

For example, if a block contains 20 coded symbols with symbol index values 0-19, then the `block_max_symbol_index` value for the block would be 19.

6.1.5.9 b_block_content_source_index_present

This Boolean indicates whether the given block contains data from a different source than the other blocks present in the bitstream. When TRUE, the `block_content_source_index` field is present in the bitstream.

If the `b_block_content_separate_sources` field in the `bitstream_header()` is TRUE, then `b_block_content_source_index_present` field shall be TRUE.

6.1.5.10 block_content_source_index

The `block_content_source_index` field identifies which source, of the `num_content_block_sources` the data in this block is associated with. The field has a range of [0, `num_content_block_sources` - 1].

6.1.5.11 b_block_composite_sources

This Boolean indicates whether the given block contains composite data from multiple different sources. The `b_block_composite_sources` field indicates whether additional fields that describe the number and size of each composite source are present in the bitstream.

6.1.5.12 block_num_composite_sources_minus1

The `block_num_composite_sources_minus1` field describes the number of composite sources that make up the data contained within the block, minus 1. The field has a range of [1, 256]. For each composite source, there will be an instance of `num_bits_code()` and the `block_composite_source_size` field in the bitstream.

6.1.5.13 bcss_bits

The operational variable `bcss_bits` contains the number of bits the subsequent related field requires and is represented by the `num_bits_code()` structure.

6.1.5.14 block_composite_source_size

The `block_composite_source_size` field describes the size of the composite source data, in bytes, that is present within the block.

6.1.5.15 b_addl_block_coding_info_present

This Boolean indicates whether additional information describing the coding used in the given block follows in the bitstream.

Otherwise, no sliding window is used and the coding information is described by other fields in the bitstream (e.g. `code_type`, `block_mask`).

6.1.5.16 addl_block_coding_mask

The `addl_block_coding_mask` field is a bit mask that describes additional coding information and indicates which subsequent fields are present in the bitstream as specified in Table 42.

Table 42: addl_block_coding_mask meaning

`addl_block_coding_mask` bit	Meaning
0	Sliding window coding used: This bit indicates whether a sliding window is used. The start and stop indexes of the sliding window are present in the packet subatom or the packet header only subatom.
1	Reserved: This bit indicates whether the `b_reserved_block_coding_params_present` element is present in the bitstream. See clause 6.1.5.15.
2	Reserved: This bit indicates whether the `reserved_block_coding_info()` structure is present in the bitstream. This structure is an instance of the `extension()` structure, and shall be ignored by bitstream parsers that conform to this version of CMMF.

6.1.5.17 b_addl_window_info_present

This Boolean indicates whether the `addl_window_info()` element follows. The `addl_window_info()` element is an instance of the `extension()` structure.

6.1.5.18 b_reserved_block_coding_params_present

This Boolean indicates whether the `reserved_block_coding_params()` element follows. The `reserved_block_coding_params()` element is an instance of the `extension()` structure, and shall be ignored by bitstream parsers that conform to this version of CMMF.

6.1.5.19 block_mask

The `block_mask` field is an array of Booleans that indicates which subsequent fields are present and follow in the bitstream as listed in Table 43.

Table 43: block_mask meaning

`block_mask` bit	Meaning
0	Sufficient symbols present: This Boolean indicates whether fields describing the number of encoded symbols contained within the bitstream are present in the bitstream. Otherwise, the number of encoded symbols contained within the bitstream cannot be immediately ascertained without attempting to parse the full bitstream and a system is unable to determine if enough symbols are present to perform a full decode of the given block.
1	Block Field size: This Boolean indicates whether fields specifying the Galois field size used during the encoding process are present in the bitstream. Otherwise, the default Galois field size, GF{2}, `block_field_size_exp_val` = 1, is used for `code_type` values that support different Galois field sizes. `code_type` values where the Galois field size is irrelevant shall set this bit to False.
2	Separated coding coefficients: This Boolean indicates whether the coding coefficient information is carried separately, either in the `packet_header()` structure as part of the packet header only subatoms, or within the `packet_group_header()` structure as part of the packet group header only subatoms. Otherwise, the coding coefficient information is included in the `packet_header()` within the packet subatoms or the `packet_group_header()` within the packet group subatoms. When this Boolean is FALSE, packet header only subatoms and packet group header only subatoms shall not be present in the bitstream.
3	Encrypted coding coefficient information: This Boolean indicates whether the coding coefficient information (and in some cases the coded symbol itself) is encrypted using symmetric encryption described by fields that follow in the bitstream.
4	Block seed: This Boolean indicates whether fields indicating the pseudorandom number generator seed used to generate the coding coefficients for every coded symbol in the block are present in the bitstream. Otherwise, the coding coefficients are transmitted by other means in the bitstream.

block_mask bit	Meaning
5	Block integrity: This Boolean indicates whether the block_integrity() structure is present in the bitstream. This structure provides a mechanism to verify that the received and decoded data in the block matches that of the original source. This structure is an instance of the fb_integrity() structure.

Bit 2 (separated coding coefficients) and bit 4 (block seed) of block_mask are mutually exclusive., and shall not both be set simultaneously.

EXAMPLE: A block_mask value of 100010b, with bit 1 set indicating an alternate block field size, and bit 5 set indicating the presence of the block integrity check.

6.1.5.20 b_sufficient_symbols_present

This Boolean indicates whether a sufficient number of encoded symbols (greater than or equal to block_num_symbols) are present in the bitstream to decode the current block. Otherwise, the block_symbols_present field is present in the bitstream.

6.1.5.21 bsp_bits

The operational variable bsp_bits is used in the syntax to specify the number of bits used to encode the block_symbols_present field and is represented by the num_bits_code() structure.

6.1.5.22 block_symbols_present

The block_symbols_present field describes the number of encoded symbols present within the bitstream associated with the current block.

In a multisource network there may be multiple CMMF bitstreams, with some bitstreams containing a sufficient number of encoded symbols to decode the current block and others with only a partial set of symbols. The block_symbols_present field describes the number of encoded symbols present within the bitstream, associated with the current block. A system may use this information to determine whether it needs to seek alternate sources (i.e. bitstreams) to obtain additional encoded symbols.

6.1.5.23 block_field_size_exp

If block_field_size_exp is present in the bitstream, coding coefficients are used to create encoded symbols. These coefficients come from a finite field set, a Galois field. The size of this Galois field is given by a characteristic prime, 2 in this case, and an exponent. This determines the number of possible coding coefficients in the field set. In addition to the field size, a Galois field set is also defined by the characteristic polynomial.

block_field_size_exp specifies both the field size exponent, and characteristic polynomial (given by *P(x)*) used to define the Galois field used for the block. The field is interpreted as listed in Table 44.

Table 44: block_field_size_exp information

block_field_size_exp value	Field Size Exponent	Characteristic polynomial *P(x)*
000b	2	$x^2 + x + 1$
001b	4	$x^4 + x + 1$
010b	8	$x^8 + x^4 + x^3 + x^2 + 1$
011b	16	$x^{16} + x^{12} + x^3 + x + 1$
100b-111b	Reserved	

For example, if block_field_size_exp has a value of 001b, then the field size exponent is 4 and the resulting Galois field size is GF{2^4} or GF{16}, meaning there are only 16 possible coding coefficients. Also, the characteristic polynomial is given by $P(x) = x^4 + x + 1$.

If the block_field_size_exp field is not present in the bitstream, i.e. if bit 1 of block_mask is set to 0, then the field size exponent takes on a default value of 1.

If the reserved values of `block_field_size_exp` (`100b-111b`) is received, it is considered an error.

The syntax uses an operational variable, `block_field_size_exp_val`. This variable takes on the field size exponent value indicated by the `block_field_size_exp` field or has the default value (`1`) when the `block_field_size_exp` field is not present in the bitstream.

6.1.5.24 Encrypted Coefficients

6.1.5.24.0 Introduction

When bit 3 of the `block_mask` field is set, the coding coefficient information in the bitstream is encrypted. The coding coefficient information can be any of:

- prng_parameters() in block_header()
- coefficient_vector() in packet_header() within the packet subatom or the packet header only subatom
- coefficient_vector() instances in packet_group_header() within the packet group subatom
- prng_parameters() in packet_header()
- coded_symbol in packet() for systematic packets

When the coding coefficient information is encrypted, keys and other encryption parameters are needed in order to decrypt the coefficients and ultimately decode the encoded symbols.

The following fields describe the coding coefficient information encryption keys and parameters.

For more information on generating coding coefficients using a PRNG, see clause 7.1.0.

6.1.5.24.1 block_cc_encryption_info_size_bits_code

The `block_cc_encryption_info_size_bits_code` field describes the number of bits used to describe the size of the block encryption information size (indicated by the `block_cc_encryption_info_size` field). The meaning of this field is specified in Table 45.

Table 45: block_cc_encryption_info_size_bits_code meaning

`block_cc_encryption_info_size_bits_code` value	Meaning
`0b`	The subatom data size is indicated using 8 bits.
`1b`	The subatom data size is indicated using 16 bits.

The variable `bcceis_bits` is used in the syntax to specify the number of bits used to encode the block encryption information and is based on the `block_cc_encryption_info_size_bits_code` field according to the following equation:

$$bcceis\_bits = (block\_cc\_encryption\_info\_size\_bits\_code + 1) \times 8$$

6.1.5.24.2 byte_align

This bit field of size 0 to 7 bits is used for the byte alignment of the associated `block_header()` structure. Byte alignment is defined relative to the start of the enclosing syntactic element.

6.1.5.24.3 block_cc_encryption_info_size

The `block_cc_encryption_info_size` field describes the size of the block encryption information data, in bytes. The data comprises of several fields and is marked by the '*block encryption info start*' and '*block encryption info end*' labels including any additional padding required to ensure that the `block_cc_encrypton_info_size` is an even number of bytes.

NOTE: All the fields between the '*block encryption info start*' and '*block encryption info end*' labels are themselves encrypted using the key identified by the `bitstream_encryption_key_id` field. This blob of data has to be decrypted first, then parsed according to the relevant `block_header()` syntax to obtain the actual bitstream field values.

6.1.5.24.4 block_cc_encryption_algorithm

`block_cc_encryption_algorithm` specifies the algorithm used to encrypt the coding coefficient information. The different algorithm types are listed in Table 46.

Table 46: block_cc_encryption_algorithm meaning

`block_cc_encryption_algorithm` Value	Meaning
000b	AES
001b-111b	Reserved

6.1.5.24.5 block_cc_encryption_mode

`block_cc_encryption_mode` specifies the mode of chaining operation used to encrypt the coding coefficient information. The different algorithm types are listed in Table 47.

Table 47: block_cc_encryption_mode meaning

`block_cc_encryption_mode` Value	Meaning
0000b	Electronic Code Book (ECB)
0001b	Cipher Block Chaining (CBC)
0010b	Cipher Feedback (CFB)
0011b	Output Feedback (OFB)
0100b	Counter (CTR)
0101b	Open Pretty Good Privacy (openPGP)
0110b-1111b	Reserved

6.1.5.24.6 block_cce_key_size_exp, block_cce_key

`block_cce_key` is the symmetric key used to encrypt the coding coefficient information. The key size (in bits) of the `block_cce_key` field is indicated by the `block_cce_key_size_exp` field according to the equation:

```
bck_bits = 2 ^ block_cce_key_size_exp
```

6.1.5.24.7 b_addl_block_cce_params_present

This Boolean indicates whether additional encryption parameters required to decrypt the coding coefficient information, contained in the `addl_cce_parameters()` structure, follow in the bitstream.

6.1.5.24.8 addl_cce_parameters()

6.1.5.24.8.0 Introduction

The `addl_cce_parameters()` structure contains additional encryption parameters. The context of `addl_cce_parameters()` is defined by the containing `block_header()`, `packet_header()`, or `packet_group_header()` structure.

6.1.5.24.8.1 num_addl_block_cce_params_minus1

`num_addl_block_cce_params_minus1` describes the number of additional encryption parameters, minus one. There can be up to 4 additional encryption parameters.

6.1.5.24.8.2 cce_parameter_type

`cce_parameter_type` specifies the type of encryption parameter that follows in the bitstream. The parameter types are listed in Table 48.

Table 48: cce_parameter_type meaning

`cce_parameter_type` Value	Meaning
`000b`	Number once (nonce)
`001b`	Initialization Vector (IV)
`010b`	Initial value
`011b`	Counter start value
`100b`	Counter (CTR)
`101b-111b`	Reserved

6.1.5.24.8.3 cce_parameter_size_exp, cce_parameter

`cce_parameter` is an encryption parameter used to encrypt the coding coefficient information. The type of parameter is described by the `cce_parameter_type` field. The parameter size (in bits) of the `cce_parameter` field is indicated by the `cce_parameter_size_exp` field according to the equation:

```
ccep_bits = 2 ^ cce_parameter_size_exp
```

6.1.5.25 Pseudorandom Noise Generator (PRNG)

6.1.5.25.0 Introduction

A PRNG is used to generate coding coefficients that are applied to source symbols to create encoded symbols. The generated coefficients can be transmitted verbatim in the bitstream, or the state of the PRNG, known as the seed, can be transmitted and a decoder can regenerate the same coding coefficient matrix calculated in an encoder. Using a row index, a decoder can select the appropriate row of the coding coefficient matrix. Transmission of the PRNG seed and row index can be more efficient than transmission of the coding coefficients.

Depending on the `code_type` value, a PRNG seed may be used to determine the coding coefficient(s) for all the coded symbols within a block, or the coded symbol in a packet.

To control the density of coding coefficients, i.e. the percentage of non-zero coefficients, a density value can be transmitted to modify the coefficients output from the PRNG can be transmitted.

The `prng_parameters()` structure contains fields that describe the PRNG seed and density values. The context of `prng_parameters()` is defined by the containing `block_header()`, or `packet_header()` structure.

If the coding coefficient information is encrypted (i.e. bit `3` of the `block_mask` field is set), then the `prng_parameters()` structure is symmetrically encrypted using the `block_cce_key` and associated parameters.

When a PRNG is used to generate a matrix of coding coefficients, a row index into this matrix is required in order to select the appropriate coding coefficients. This row index value is represented by:

- `packet_symbol_index` in the `packet_header()` in a packet subatom
- `packet_group_symbol_index` in the `packet_group_header()` in a packet group subatom
- `mbpg_symbol_index` in the `mbpg_header()` in a multiple block packet group subatom

If a coded symbol has a row index value less than the `block_num_symbols` value in the associated block header subatom, then that coded symbol shall be a systematic symbol, regardless of whether it is indicated as such (for example by the `b_systematic_symbol` flag in `packet_header()`). Further, when the row index is less than `block_num_symbols`, the identified row of coefficients generated by the PRNG shall be discarded and replaced with a coefficient vector consisting of all `0`'s except at the index identified by the row index value (as indicated by either `packet_symbol_index`, or `packet_group_symbol_index`, or `mbpg_symbol_index`), where the coefficient shall be `1`.

For more information on generating coding coefficients using a PRNG, see clause 7.1.0.

6.1.5.25.1 prng_type

`prng_type` specifies the type of number generator to use to generate the coding coefficient(s) for the given block or packet. The different prng types are listed in Table 49.

Table 49: block_prng_type meaning

`block_prng_type` Value	Meaning
`000b`	Mersenne Twister (MT) [8]
`001b-111b`	Reserved

6.1.5.25.2 prng_seed_bits_code

The `prng_seed_bits_code` field describes the number of bits the `prng_seed` field is encoded with as indicated in Table 50.

Table 50: block_prng_seed_bits_code meaning

`prng_seed_bits_code` value	Meaning
0	16 bits
1	32 bits
2	64 bits
3	Reserved

The operational variable `ps_bits` uses the `prng_seed_bits_code` value to determine the number of bits the subsequent `prng_seed` field requires. It can also be calculated as follows:

```
ps_bits = 2^(prng_seed_bits_code + 4)
```

If the reserved value of `prng_seed_bits_code` (3) is received, it is considered an error.

6.1.5.25.3 prng_seed

`prng_seed` specifies the seed value used to initialize the PRNG for generating coefficients for the block or packet.

6.1.5.25.4 prng_density_percentage

`prng_density_percentage` describes the percentage of coding coefficients that are, on average, non-zero. The field is encoded as a fraction in range (0, 1] and the actual percentage is determined by the following equation:

$$\text{density percentage} = 100 \times (\texttt{prng\_density\_percentage} + 1) / 128$$

An operational variable, `density_int` is defined as:

$$\texttt{density\_int} = (\texttt{prng\_density\_percentage} + 1)$$

6.1.6 packet()

6.1.6.0 Introduction

The packet (`packet()`) structure contains the `packet_header()`, as well as a coded symbol (an individual symbol group), which is a fundamental piece of encoded data.

Packets, as well as other fields contained within larger structures, inherit values from previous `subatom()` instances. For example, the `packet()` structure inherits the value of the `block_field_size_exp` field specified in the associated `block_header()`, that is, the `block_header()` with the `block_index` value that matches the `packet_block_index` value.

6.1.6.1 packet_block_index

The operational variable `packet_block_index`, which is represented by the `block_index_or_count_value()` structure if `block_count` is greater than `1`, describes which block this `packet()` (or `packet_header_only()`) instance is associated with. If the bitstream only contains a single block (i.e. `block_count` is `1`), then the `block_index_or_count_value()` structure is not present in the bitstream and takes on a value of `0`.

As an example, if the `packet_block_index` is `1`, then the associated block and its related `block_header()` instance has a `block_index` value of `1` as well.

6.1.6.2 coded_symbol

A `coded_symbol` is either a systematic symbol or the encoded version of a number of source symbols, generated by applying coding to the original data symbols. The size of this field is determined by the operational variable `css_bits` calculated using the following formula:

```
                    css_bits = (subatom_size × 8) -
sizeof(packet_block_index,packet_block_index_ext,packet_header())
```

In the equation above, `subatom_size` is the size indicated in the subatom header for this instance of the `packet()` subatom. The `sizeof` function determines the number of bits used by the fields, or expansion of the structures, prior to the `coded_symbol_data` field.

If the value of the `block_symbol_size` field in the associated `block_header()` instance is not `0`, then the value of `css_bits` shall be equal to `block_symbol_size × 8`.

If the coding coefficient information is encrypted (i.e. bit `3` of the `block_mask` field is set), and the `b_systematic_symbol` field is also set, and the `b_systematic_symbol_encrypted` field is also set, then the `coded_symbol` field is symmetrically encrypted using the `block_cce_key` and associated parameters.

6.1.7 packet_header()

6.1.7.0 Introduction

The `packet_header()` structure provides information about how the coded symbols were encoded. The `packet_header()` will be different depending on whether it is encapsulated in a packet subatom or in a packet header only subatom.

6.1.7.1 b_systematic_symbol

When True, this boolean indicates that the coded symbol in the packet is systematic (i.e. not coded). In this case, a symbol index is present in the bitstream so that the original source symbol can be identified and used to decode the other packets. When this field is False, no immediate indication is given that the coded symbol is systematic, it may indeed be systematic but more information is needed, for example a coefficient vector, to determine wither it is a systematic symbol or not.

If the block contains separated coding coefficients in packet header only subatoms (i.e. bit 2 of the `block_mask` field is set), then the `b_systematic_symbol` field shall be cleared in the packet subatom, and its correct value shall be indicated in the `b_systematic_symbol` field of the associated packet header only subatom.

6.1.7.2 packet_mask

The packet_mask field is an array of Booleans that describes which subsequent fields are present in the bitstream as listed in Table 51.

Table 51: packet_mask meaning

packet_mask bit	Meaning
0	Symbol/row index present: This Boolean indicates whether index fields are present in the bitstream that identify the symbol or provide an encoding rank.
1	Coding coefficient encryption parameter(s) present: This Boolean indicates whether encryption parameters relevant to this packet are present in the bitstream. This bit can only be set if bit 3 of block_mask is also set.
2	Window size: This Boolean indicates whether the coded symbol in the packet is created from a subset of the original symbols contained within the block - a window of symbols, and the start and stop index are present in the bitstream. Otherwise, the coded symbol was created from all the symbols (as given by the coding coefficients) contained within the block.
3	Packet seed: This Boolean indicates whether fields specifying the pseudorandom number generator seed used to generate the coding coefficient for the coded symbol in the packet are present in the bitstream. Otherwise, the coding coefficients are transmitted by other means in the bitstream.
4	Coding vector present: This Boolean indicates whether the coefficient_vector() structure is present in the bitstream. This structure explicitly describes the coding coefficients that were used to generate the coded symbol in the packet.
5	Packet integrity: This Boolean indicates whether the packet_integrity() structure is present in the bitstream. This structure provides a mechanism to verify that the received data in the packet matches that of the original source bitstream.
6	Packet header extension: This Boolean indicates whether the packet_header_extension() structure is present in the bitstream. The packet_header_extension() structure provides a mechanism to extend the packet header syntax with additional information and is an instance of the extension() structure.

Certain bits of the packet_mask are set or cleared depending on the values of other fields. Including:

- If b_systematic_symbol is set, then:
  - Bit 0 of packet_mask shall be set.
  - Neither bit 2 (window start/stop indexes), nor bit 3 (packet seed), nor bit 4 (coefficient vector) shall be set. That is, the only allowed packet_mask value is BB0001b (see note below).
- Bit 3 (packet seed) and bit 4 (coefficient vector) are mutually exclusive, and shall not both be set simultaneously.
- If bit 1 of the addl_block_coding_mask is set, indicating that a sliding window is used, bit 2 of packet_mask shall be set.
- If bit 2 of block_mask is set, indicating that the coding coefficients are separated and described in the packet header only subatom, then:
  - Bit 0 shall be set in the packet_mask field in both the packet subatom as well as the associated packet header only subatom. The corresponding packet_symbol_index values shall have the same value in both instances.
  - Bits 1, 2, 3, 4, 5, and 6 of packet_mask in packet subatom shall be cleared. That is, the only allowed packet_mask value is 000001b (see note). These bits may be set as needed in the associated packet header only subatom packet_mask field.

NOTE: Bit values indicated as B mean that the value of this bit can be set, or not set depending on other settings in an encoder.

An example packet_mask can be 0100001b, with bit 0 set indicating the packet_symbol_index field is present in the bitstream, and bit 5 set indicating the presence of the packet integrity check.

6.1.7.3 psi_bits

The operational variable `psi_bits` is used in the syntax to specify the number of bits used to encode the `packet_symbol_index` field. `psi_bits` is represented by the `num_bits_code()` structure.

6.1.7.4 packet_symbol_index

The `packet_symbol_index` field defines an index whose meaning is context dependent:

- If the `b_systematic_symbol` is set, the `packet_symbol_index` value is the original symbol index number.
- For certain `code_type` values, (e.g. 1 (RaptorQ) or 2 (Reed-Solomon)), the `packet_symbol_index` value represents the encoded symbol index to identify each coded symbol.
- If bit 2 of `block_mask` is set, the `packet_symbol_index` value associates packet header only subatoms carrying the separated coding coefficient information, and the packet subatoms carrying the `coded_symbol`.
- If bit 4 of `block_mask` is set, the `packet_symbol_index` value is the row of the coding coefficient matrix generated by using the block PRNG. Further, if the `b_systematic_symbol` field is set for this packet, or the value of `packet_symbol_index` is less than `block_num_symbols` of the associated block header subatom, then the identified row of coefficients generated by the block PRNG shall be discarded and replaced with a coefficient vector consisting of all `0`'s except at the index identified by the `packet_symbol_index` value, where the coefficient shall be `1`.

The value of the `packet_symbol_index` field is `0`-indexed. For example, a `packet_symbol_index` value of `0` can reference the 1st original input symbol for a systematic packet.

6.1.7.5 Encryption Parameters

6.1.7.5.0 Introduction

When bit `1` of the `packet_mask` field is set, additional encryption parameters needed to decrypt the coding coefficient information within the `packet_header()` are present in the bitstream. The coding coefficient information within the `packet_header()` can be any of:

- `coefficient_vector()` in `packet_header()` within the packet subatom or the packet header only subatom
- `packet_prng_seed` in `packet_header()` or `packet_header_only()`
- `coded_symbol` in `packet_header()` for systematic packets

When the coding coefficient information is encrypted, keys and other encryption parameters are needed in order to decrypt the coefficient information and ultimately decode the coded symbols.

The following fields describe the additional packet or packet header only encryption parameters.

For more details on encrypting coding coefficient information, see clause 7.3.

6.1.7.5.1 b_systematic_symbol_encrypted

This Boolean indicates whether the `coded_symbol` field is encrypted. This field can only be set if the `b_systematic_symbol` field is also set.

6.1.7.5.2 b_addl_packet_cce_params_present

This Boolean indicates whether additional encryption parameters required to decrypt the coding coefficient information, contained in the `addl_cce_parameters()` structure, follow in the bitstream.

6.1.7.6 window_start_index, window_stop_index

The `window_start_index` and `window_stop_index` fields indicate the index of the first symbol present in the coded symbol, the start index, and the index of the last symbol present in the coded symbol, the stop index, to form a window of contiguous symbols from amongst the total set of symbols contained within the block.

The operational variable `window_size` is used in the syntax when the `window_start_index` and `window_stop_index` fields are present in the bitstream, the maximum `window_size` value is 65 536 and is calculated as follows:

```
window_size = modulo(window_stop_index - window_start_index + 1, 65 536)
```

NOTE: Although the maximum window_size value is 65 536 when the window_start_index and window_stop_index fields are present in the bitstream, other practical considerations may impose a stricter limit on the maximum window_size value.

If the `window_start_index` and `window_stop_index` fields are not present in the bitstream, the `window_size` variable is calculated as follows:

```
window_size = block_num_symbols
```

Where `block_num_symbols` is the value taken from the associated `block_header()`.

6.1.7.7 byte_align

This bit field of size 0 to 7 bits is used for the byte alignment of the associated `packet_header()` structure. Byte alignment is defined relative to the start of the enclosing syntactic element.

6.1.8 encoder_content_info()

6.1.8.0 Introduction

The CMMF bitstream has applications in online delivery of audio/video content. The optional encoder content information (`encoder_content_info()`) structure contains information about the specific encoder utilized, and the source assets, such as a content identification. This subatom is useful when CMMF bitstreams for a given asset are generated by multiple encoders in a distributed and/or decentralized fashion. The information contained within this subatom can be used to both validate that the encoder used to generate the bitstream is authorized and the associated bitstream is compatible with bitstreams created by other encoders.

6.1.8.1 b_encoder_id_present

This Boolean indicates whether the `encoder_uuid` field is present the bitstream.

6.1.8.2 encoder_uuid

The `encoder_id` field identifies the specific encoder used to create the bitstream. The field carries a version 4, variant 1 universally unique identifier as specified in IETF RFC 9562 [1]. This field enables efficient use of a plurality of encoders in decentralized settings.

6.1.8.3 Content Identification

The content encoded in an CMMF bitstream can have a content identification.

6.1.8.4 b_content_id_present

6.1.8.4.0 Introduction

This Boolean indicates whether the `content_id_type`, `content_id_size_minus1`, and `content_id`, fields are present the bitstream.

6.1.8.4.1 content_id_type

Several different identification options are supported. The `content_id_type` field indicates the type of identification present in the bitstream. The description of the values is specified in Table 52.

Table 52: content_id_type description

`content_id_type` Value	Identification Description Type
`0x0`	Entertainment Identification Registry (EIDR)
`0x1`	Advertising Digital Identifier (Ad-iD)
`0x2`	Custom identification defined by the user
`0x3`	Universally Unique Identifier (UUID)
`0x4 - 0xE`	Reserved
`0xF`	Unspecified

6.1.8.4.2 content_id_size_minus1

This field indicates the size (in bytes) of the `content_id` field that immediately follows this field, minus one. For example, a `content_id_size_minus1` value of '11' indicates a `content_id` of 12 bytes.

6.1.8.4.3 content_id

This field shall contain the content ID associated with the current content as specified by the `content_id_type`:

- EIDR: For an EIDR content ID type, the `content_id` field shall contain a 96-bit identifier that is registered with EIDR (http://eidr.org) in Compact Binary Format as defined by EIDR [2].
- Ad-ID: For an Ad-ID content ID type, the `content_id` field shall contain an Ad-ID String that represents an identifier registered with Ad-ID (http://www.ad-id.org/) as defined by Ad-ID [3].
- UUID: For a UUID content ID type, the `content_id` field shall contain a 128-bit UUID as defined by IETF RFC 9562 [1].
- Custom ID: For a custom ID type, the `content_id` field is "free-form" and can be populated with ASCII [7] or UTF-8 [11] data.

6.1.8.5 b_content_location_present

This Boolean indicates whether the `content_location_size` and `content_location` fields are present in the bitstream.

6.1.8.6 content_location_size, content_location

The `content_location` field provides the location of the content with size indicated by the `content_location_size` field (in bytes). This field is a string of ASCII [7] or UTF-8 [11] characters. The `content_location_size` field shall not have a value of `0`.

6.1.8.7 b_content_type_present

This Boolean indicates whether the `content_type_size` and `content_type` fields are present in the bitstream.

6.1.8.8 content_type_size, content_type

The `content_type` field is a human-readable description of the type of content with size indicated by the `content_type_size` field (in bytes). This field is a string of ASCII [7] or UTF-8 [11] characters. The `content_type_size` field shall not have a value of `0`.

6.1.8.9 b_content_header_present

This Boolean indicates whether the `content_type_size` and `content_type_size` fields are present in the bitstream.

6.1.8.10 content_header_size, content_header

The `content_header` field contains the first `content_type_size` bytes of the source data, unencoded. The `content_header` field can be parsed using existing methods to determine details about the content encoded within the bitstream. Example details include bit rate, resolution for video content, sampling frequency for audio content, etc. The `content_type_size` field shall not have a value of `0`.

6.1.8.11 b_file_integrity_present

This Boolean indicates whether the `file_integrity()` structure is present in the bitstream. The `file_integrity()` structure is just an instance of the `fb_integrity()` structure that applies to the full file.

6.1.8.12 b_media_preso_dur_present

This Boolean indicates whether the `media_presentation_duration()` structure is present in the bitstream. An encoded bitstream may only represent a segment of the overall source content, and the `media_presentation_duration()` structure indicates the duration of the entire presentation of the source content. This structure is an instance of the `cmmf_time()` structure. While this duration is optional, it, along with other fields in this subatom, is useful to a decoder to determine if the bitstream can be decoded with bitstreams received from other encoders.

6.1.9 media_segment_info()

6.1.9.0 Introduction

The media segment information (`media_segment_info()`) structure provides information about the underlying CMMF encoded media. This optional structure enables a system to inspect the underlying CMMF coded media without requiring a decode operation in settings that do not support access to a centralized service entity, manifest files, etc. e.g. information centric networks, peer-to-peer systems, distributed file systems, etc. The media information may be populated from a DASH or HLS manifest file as an example. The `media_segment_info()` subatom is useful when content is stored or cached across multiple endpoints (e.g. a CMMF bitstream containing only `packet()` subatoms is cached in one location while another CMMF bitstream containing other subatoms is cached in another location). In these cases, it may be useful to have information about the underlying essence attached to both bitstreams from both a network management and a delivery standpoint. As a more concrete example, assuming the encoding of an HLS asset. The HLS manifest is encoded within one CMMF bitstream while each segment of that stream is encoded into their own CMMF bitstreams. A management layer can be implemented that links the HLS manifest CMMF bitstream with all of the segment bitstreams. Alternatively, the `media_segment_info()` subatom can be used to perform this linking, which also adds the benefit of not having to duplicate the HLS manifest. This subatom complements the higher-level manifest and allows for processes within the network to manage how the overall asset is distributed within that network.

A media segment information subatom describes a single type of media (e.g. audio or video). If the media is multiplexed and contains multiple types of media, or if different blocks contain different types of media, then multiple instances of the media segment information subatom may be used.

6.1.9.1 media_segment_block_index, media_segment_block_index_ext

The `media_segment_block_index` field, which is extendable by the `media_segments_index_ext` field, describes which block this `media_segment_info()` instance is associated with.

6.1.9.2 media_segment_index, media_segment_index_ext

The `media_segment_index` field, which is extendable by the `media_segment_index_ext` field, identifies the media index of the instance of the `media_segment_info()` structure. The index is 0-based and shall have a value 0 except in cases where the media is multiplexed where each multiplexed stream shall have a unique `media_segment_index` value.

6.1.9.3 b_composite_source_index _present

This Boolean indicates whether `media_segment_composite_source_index` is present in the bitstream.

6.1.9.4 media_segment_composite_source_index

When the block associated with the relevant media segment (as indicated by the `media_segment_block_index` field) is a composite of different data sources, the `media_segment_composite_index` identifies which data source in that block the media segment applies to.

6.1.9.5 b_asset_name_present

This Boolean indicates whether an asset name is present in the bitstream.

6.1.9.6 asset_name_size, asset_name

The `asset_name` is the name of the media asset of size indicated by the `asset_name_size` field (in bytes). This field is free-form and can be any ASCII [7] or UTF-8 [11] string of characters. The `asset_name_size` field shall not have a value of `0`.

6.1.9.7 segment_tag_mask

The `segment_tag_mask` field is a bit mask that describes which subsequent fields are present in the bitstream as listed in Table 53.

Table 53: segment_tag_mask meaning

`segment_tag_mask` bit	Meaning
0	Segment duration present: This bit indicates whether the duration of the encoded media segment, as described by the `segment_duration()` structure, is present in the bitstream. This structure is an instance of the `cmmf_time()` structure. The duration is optional, and can be used to replicate the corresponding information from a manifest.
1	Segment start time present: This bit indicates whether the start time of the encoded media segment, as described by the `segment_start_time()` structure, is present in the bitstream. This structure is an instance of the `cmmf_time()` structure. The start time is optional, and can be used to replicate the corresponding information from a manifest.
2	Segment index present: This bit indicates whether the `segment_index` field is present in the bitstream.
3	Segment count present: This bit indicates whether the `segment_count` field is present in the bitstream.

6.1.9.8 segidx_bits, segcnt_bits

The operational variables `segidx_bits` and `segcnt_bits` are represented by the `num_bits_code()` structure.

6.1.9.9 segment_index

The `segment_index` field identifies which segment of the media is contained within the encoded data.

6.1.9.10 segment_count

The `segment_count` field describes how many segments of this media exist.

6.1.9.11 b_media_mime_type_present

This Boolean indicates whether information about the MIME type of the media is provided in the bitstream. In this case, the `media_mime_type` and `media_mime_type_size` fields are present in the bitstream.

6.1.9.12 media_mime_type_size, media_mime_type

`media_mime_type` is a string of ASCII [7] characters that describes the content using a discrete media type value as defined in section 4 of IETF RFC 2046 [23] and/or IETF RFC 4337 [22]. Up to the first 63 bytes of the media type can be indicated. The size (in bytes) of the `media_mime_type` field is indicated by the `media_mime_type_size` field. The `media_mime_type_size` field shall not have a value of `0`.

6.1.9.13 b_media_codec_present

This Boolean indicates whether information about the codec is provided in the bitstream. In this case, the `media_codec` and `media_codec_size` fields are present in the bitstream.

6.1.9.14 media_codec_size, media_codec

`media_codec` is a string of ASCII [7] or UTF-8 [11] characters conforming to either the simp-list or fancy-list productions of IETF RFC 6381 [5], section 3.2, without the enclosing DQUOTE characters. Up to the first 63 bytes of a single codec identifier for the media format, mapped into the name space for codecs as specified in IETF RFC 6381 [5], section 3.3, shall be used. The size (in bytes) of the `media_codec` field is indicated by the `media_codec_size` field. The `media_codec_size` field shall not have a value of `0`.

6.1.9.15 b_bit_rate_present

This Boolean indicates whether the `bit_rate` field is present in the bitstream.

6.1.9.16 bit_rate_bits_code

The `bit_rate_bits_code` field describes how many bits are used to describe the bit rate of the media segment.

Table 54: bit_rate_bits_code meaning

`bit_rate_bits_code` Value	Meaning
`0b`	24 bits
`1b`	32 bits

The operational variable `bps_bits` is used to specify the number of bits used to represent the bit rate and is calculated using the `bit_rate_bits_code` as follows:

```
bps_bits = (bit_rate_bits_code + 3) × 8
```

6.1.9.17 bit_rate

The `bit_rate` field describes the bit rate of the media segment and has units of bits per second.

6.1.9.18 b_ms_content_type_present

This Boolean indicates whether the `ms_content_type` field is present in the bitstream.

6.1.9.19 ms_content_type

The `ms_content_type` field describes the type of content contained within the media segment. The field is described in Table 55.

Table 55: ms_content_type description

`ms_content_type` value	Description
`0x0`	Video
`0x1`	Audio
`0x2`	Multiplexed media of different content types
`0x3`	Application data
`0x4`	Text
`0x5-0x7`	Reserved

6.1.9.20 b_ms_content_type_info_present

This Boolean indicates whether additional information based on the value of the `ms_content_type` field is present in the bitstream. It shall not be set unless `b_ms_content_type_present` is set.

6.1.9.21 b_aspect_ratio_present

This Boolean indicates whether information about the picture aspect ratio is provided in the bitstream. In this case, the `sample_aspect_ratio` and potentially the `sar_width` and `sar_height` fields are present in the bitstream.

6.1.9.22 sample_aspect_ratio

The picture aspect ratio describes the ratio of the width dimension to the height dimension of the video. The `sample_aspect_ratio` field has the same meaning as the SampleAspectRatio field as described in ISO/IEC 23091-2 [10].

6.1.9.23 sar_width, sar_height

The picture aspect ratio describes the ratio of the width dimension to the height dimension of the video. When the `sample_aspect_ratio` has a value of `255`, then the `sar_width` and `sar_height` fields are present in the bitstream and indicate the horizontal and vertical dimensions, respectively, of the sample aspect ratio of the media segment.

6.1.9.24 b_dynamic_resolution_video

This Boolean indicates whether the resolution of the video varies throughout the program. In this case, and if `b_resolution_present` is set, the `resolution_width` and `resolution_height` fields indicate the maximum resolution of the program.

6.1.9.25 b_resolution_present

This Boolean indicates whether the resolution of the video is provided in the bitstream. In this case, the `resolution_width` and `resolution_height` fields are present in the bitstream.

6.1.9.26 resolution_width, resolution_height

The `resolution_width` and `resolution_height` fields describe the horizontal pixel count and vertical line count, respectively, of the related video media segment.

6.1.9.27 b_frame_rate_present

This Boolean indicates whether the video frame rate is provided in the bitstream. In this case, the `frame_rate` field and a `reserved` field are present in the bitstream.

6.1.9.28 frame_rate

The `frame_rate` field describes how many frames per second (fps) appear in the associated video. The meaning of this field is specified in Table 56. In the case of interlaced video, the frame rate is 1/2 the field rate. Fractional frame rates expressed as decimal fractions, such as 23,976 fps or 59,94 fps, are approximations of the exact frame rate. The exact frame rate is calculated by applying a factor of 1 000 / 1 001 to the nearest adjacent integer frame rate (e.g. 23,976 fps represents 24 fps × 1 000 / 1 001).

Table 56: frame_rate values

frame_rate value	Frame rate (fps)
0	23,976
1	24
2	25
3	29,97
4	30
5	47,95
6	48
7	50
8	59,94
9	60
10	100
11	119,88
12	120
13 - 31	Reserved

6.1.9.29 b_hdr_info_present

This Boolean indicates whether High Dynamic Range (HDR) video information follows in the bitstream. Otherwise, the video should not be considered HDR.

6.1.9.30 hdr_compatibility_mask

The `hdr_compatibility_mask` field is a bit mask where each bit indicates that the video is compatible with a specific HDR format. The meaning of this field is specified in Table 57.

Table 57: hdr_compatibility_mask values

`hdr_compatibility_mask` Bit	Meaning (compatible with)
0	HDR10
1	HLG™
2	SMPTE 2094-10
3	SMPTE 2094-20
4	SMPTE 2094-30
5	SMPTE 2094-40
6	Reserved
7	Reserved
8	Dolby Vision™
9	HDR10+™
10	SL-HDR1™
11	SL-HDR2™
12	SL-HDR3™
13	Reserved
14	Reserved
15	Reserved

6.1.9.31 b_addl_hdr_info_present

This Boolean indicates whether additional High Dynamic Range (HDR) video information follows in the bitstream as indicated by the `hdr_compat_mask_index`, `hdr_profile`, `hdr_level`, and `hdr_compatibility_id` fields.

6.1.9.32 hdr_compat_mask_index

The `hdr_compat_mask_index` field is an index into `hdr_compatibility_mask` field to identify which HDR format the subsequent `hdr_profile`, `hdr_level`, and `hdr_compatibility_id` fields correspond to.

For example, if `hdr_compat_mask_index` has a value of `8`, then the subsequent fields relate to Dolby Vision™.

6.1.9.33 hdr_profile

The `hdr_profile` field specifies the profile of the HDR video.

6.1.9.34 hdr_level

The `hdr_level` field specifies the level of the HDR video.

6.1.9.35 hdr_compatibility_id

The `hdr_compatibility_id` field specifies a particular form of a base layer substream.

6.1.9.36 b_addl_video_info_present

This Boolean indicates whether the `addl_video_info()` structure is present in the bitstream. The `addl_video_info()` structure provides a mechanism to extend the video content type with additional information and is an instance of the `extension()` structure.

6.1.9.37 b_sampling_freq_present

This Boolean indicates whether the audio sampling frequency is provided in the bitstream. In this case, the `b_sampling_freq_is_48k` field is present in the bitstream.

6.1.9.38 b_sampling_freq_is_48k

This Boolean indicates whether the audio in this media segment has an audio sampling frequency of 48 000 Hz. Otherwise, the `sampling_frequency` field is present in the bitstream, providing the sampling frequency of the audio.

6.1.9.39 sampling_frequency

The `sampling_frequency` field describes the sampling rate of the audio in the media segment. The value of this field is specified in Table 58.

Table 58: sampling_frequency values

`sampling_frequency` value	Sampling frequency (kHz)
0	6
1	8
2	12
3	24
4	48
5	96
6	192
7	22,05
8	44,1
9	88,2
10	176,4
11 - 15	Reserved

6.1.9.40 b_audio_config_present

This Boolean indicates whether the audio channel configuration is described in the bitstream. In this case, the `audio_channel_config` field is present in the bitstream.

6.1.9.41 audio_channel_config

The `audio_channel_config` field defines the audio channel configuration of the media segment. The field is a bit mask of speaker groups present in the channel configuration.

The value of this field is specfied in Table 59.

Table 59: audio_channel_config values

`audio_channel_config` Bit	Speaker group location(s)
0	Left (L)
	Right (R)
1	Centre (C)
2	Left surround (Ls)
	Right surround (Rs)
3	Left back (Lb)
	Right back (Rb)
4	Top front left (Tfl)
	Top front right (Tfr)
5	Top back left (Tbl)
	Top back right (Tbr)
6	Low-Frequency Effects (LFE)
7	Top Left (Tl)
	Top Right (Tr)
8	Top side left (Tsl)
	Top side right (Tsr)
9	Top front centre (Tfc)
10	Top back centre (Tbc)
11	Top centre (Tc)
12	Low-frequency effects 2 (LFE2)
13	Bottom front left (Bfl)
	Bottom front right (Bfr)
14	Bottom front centre (Bfc)
15	Back centre (Bc)
16	Left screen (Lscr)
	Right screen (Rscr)
17	Left wide (Lw)
	Right wide (Rw)
18	Vertical Height Left (Vhl)
	Vertical Height Right (Vhr)
19	Reserved
20	Reserved
21	Reserved
22	Reserved
23	Reserved

All references to ITU channel names for each speaker group are with regard to Recommendation ITU-R BS.2051-3 [20].

Bit 23 indicates whether the audio presentation is object based and thus channel presence does not apply. In this case, bits 22...0 are reserved. Otherwise, bits 18...0 indicate the presence of individual channel groups in the audio presentation that together form the corresponding channel configuration. In this case bits 18...0 shall indicate the presence of channel groups in the original content.

For example, if the audio channel configuration contains the L and R channels, then Bit `0` is set to `1`. As another example, for a stream with a 5.1.2 channel configuration of L, C, R, Ls, Rs, LFE, Tsl, Tsr, the `audio_channel_config` value is `0x00 0147` (the hexadecimal equivalent of the binary value `0000 0000 0000 0001 0100 0111`).

6.1.9.42 b_audio_props_present

This Boolean indicates whether additional fields describing other audio properties are provided in the bitstream. In this case, the `b_virtualized_bin`, `b_object_audio` and `b_complexity_index_present` fields, as well as two `reserved` fields, are present in the bitstream.

6.1.9.43 b_virtualized_bin

This Boolean indicates whether the audio contains a binaural virtualized experienced. In this case, and if `b_audio_config_present` is set, then the `audio_channel_config` field shall have a value of `0x00 0001` indicating that only the L and R channels are present.

6.1.9.44 b_object_audio

This Boolean indicates whether the media segment contains an object-based audio presentation. Otherwise, the audio presentation is channel based.

6.1.9.45 b_complexity_index_present

This Boolean indicates whether the complexity index information is provided in the bitstream. In this case, the `complexity_index` field is present in the bitstream.

6.1.9.46 complexity_index

The `complexity_index` field specifies the number of channels and/or objects of the audio.

6.1.9.47 b_addl_audio_info_present

This Boolean indicates whether the `addl_audio_info()` structure is present in the bitstream. The `addl_audio_info()` structure provides a mechanism to extend the audio content type with additional information and is an instance of the `extension()` structure.

6.1.9.48 b_addl_ms_content_type_info_present

This Boolean indicates whether the `addl_ms_content_type_info()` structure is present in the bitstream. The `addl_ms_content_type_info()` structure provides a placeholder to define additional information for other `ms_content_type` values. This structure is an instance of the `extension()` structure.

6.1.9.49 accessibility_mask

The `accessibility_mask` field is a bit mask that describes which subsequent accessibility-related fields are present in the bitstream as listed in Table 60.

Table 60: accessibility_mask meaning

`accessibility_mask` bit	Meaning
0	Language present: If this bit is set, the language of the media segment is present in the bitstream.
1	Reserved present: If this bit is set, then a `reserved` field is present in the bitstream.
2	Reserved present: If this bit is set, then a `reserved` field is present in the bitstream.
3	Additional accessibility information present: If this bit is set, the `addl_accessibility_info()` structure is present in the bitstream. This structure is an instance of the `extension()` structure.

6.1.9.50 language_size, language

The `language` field contains a language tag that conforms to the syntax and semantics defined in IETF RFC 4646 [6]. The size (in bytes) of the field is indicated by the `language_size` field. The minimum sequence length shall be two bytes, supporting a two-character language tag as specified in ISO 639 [4]. The maximum supported length of the language tag shall be 42 bytes.

6.1.10 cmmf_time()

6.1.10.0 Introduction

The `cmmf_time()` structure represents temporal information, with a range of 31 days, 23 hours, 59 minutes, and 59,999 seconds, and a granularity of one microsecond (1 second / 1 000), one nanosecond (1 second / 1 000 000), or one picosecond (1 second / 1 000 000 000). Multiple instances of this syntax are present in the CMMF bitstream, e.g. `media_presentation_duration` and `start_time` fields.

6.1.10.1 b_ddhhmmss

This Boolean indicates whether the subsequent temporal information is presented in a days, hours, minutes, and seconds (DD:HH:MM:SS) format, and the `b_dd_present`, `b_hh_present`, `b_mm_present`, and `b_ss_present` fields are present in the bitstream. Otherwise, the temporal information is presented in seconds, and the `int_seconds_bits_code` and `int_seconds` fields are present in the bitstream.

6.1.10.2 DD:HH:MM:SS format

If the `b_ddhhmmss` field is set, the temporal information is indicated by the `days`, `hours`, `minutes`, and `seconds` fields. Each of these fields has an associated presence flag indicating whether the field is present in the bitstream. A field that is not present in the bitstream has a value of `0`.

Table 61: DD:HH:MM:SS format fields and meaning

Presence flag	Associated field	Allowable range
`b_dd_present`	days	[0, 31]
`b_hh_present`	hours	[0, 23]
`b_mm_present`	minutes	[0, 59]
`b_ss_present`	seconds	[0, 59]

6.1.10.3 int_seconds_bits_code

The `int_seconds_bits_code` field describes the number of bits used to encode the `int_seconds` field. The meaning of this field is specified in Table 62.

Table 62: int_seconds_bits_code meaning

`int_seconds_bits_code` value	Meaning
`00b`	The value of `int_seconds_bits_code` is 8 bits.
`01b`	The value of `int_seconds_bits_code` is 16 bits.
`10b`	The value of `int_seconds_bits_code` is 24 bits.
`11b`	Reserved.

The operational variable `is_bits` can be calculated directly from the `int_seconds_bits_code` field according to the following equation:

$$is\_bits = (int\_seconds\_bits\_code + 1) \times 8$$

6.1.10.4 int_seconds

If the `b_ddhhmmss` field is cleared, the temporal information is indicated by the `int_seconds` field, describing the number of integer seconds. This field has a range of [0, 2 764 799] (up to 31 days, 23 hours, 59 minutes, 59 seconds).

6.1.10.5 b_fract_seconds_present

This Boolean indicates whether the `fract_seconds` field is present in the bitstream.

6.1.10.6 fract_seconds_bits_code

The `fract_seconds_bits_code` field describes the number of bits used to encode the `fract_seconds` field. The meaning of this field is specified in Table 63.

Table 63: fract_seconds_bits_code meaning

`fract_seconds_bits_code` value	Meaning
`00b`	The value of `fract_seconds_bits_code` is 10 bits.
`01b`	The value of `fract_seconds_bits_code` is 20 bits.
`10b`	The value of `fract_seconds_bits_code` is 40 bits.
`11b`	Reserved.

The operational variable `s_bits` can be calculated directly from the `fract_seconds_bits_code` field according to the following equation:

```
fs_bits = 10 × 2^fract_seconds_bits_code
```

6.1.10.7 fract_seconds

The `fract_seconds` field describes the number of fractional seconds added to the integer seconds described above. The field has a range depending on the value of `fs_bits`, as shown in Table 64.

Table 64: fract_seconds range and divisor

`fs_bits` value	`fract_seconds` range	`fract_seconds` divisor
10 bits	[0, 999]	1 000
20 bits	[0, 999 999]	1 000 000
40 bits	[0, 999 999 999 999]	1 000 000 000 000

The number of fractional seconds is then calculated by dividing the `fract_seconds` value in the bitstream by the indicated divisor value as shown in the following equation:

fractional seconds = `fract_seconds / ` divisor

6.1.11 chunked_subatom()

6.1.11.0 Introduction

The `chunked_subatom()` structure contains a 'chunk' of subatom data. It provides a mechanism to partition large subatom data into multiple smaller chunk segments. For example if the coded symbol size in a packet subatom is very large for the transmission protocol in use, the subatom `packet()` data can be split into smaller chunk segments that fit the transmission requirements.

A `chunked_subatom()` cannot itself be chunked. When subatom data is chunked, it shall be split into more than one `chunked_subatom()` instances.

6.1.11.1 chunk_segment_id

The `chunk_segment_id` field identifies a series of `chunked_subatom()` instances as containing different portions of the same subatom data. When subatom data is chunked, all `chunked_subatom()` instances of that data shall have the same value in the `chunk_segment_id` field.

6.1.11.2 chunk_segment_index

The `chunk_segment_index` field identifies the index of the chunk segment of subatom data contained within the `chunked_subatom()`. A parser can use this information to reassemble the chunked subatom data to its original order.

If a chunk segment is lost in transmission, and a complete set of chunk segments is not received, it is up to the decoder implementation to determine how to recover. For instance, the decoder may introduce a timer, if all the chunk segments are not received within the appropriate time range, then that entire series of `chunked_subatom()` instances (i.e. all instances with the same `chunk_segment_id`), may be discarded.

6.1.11.3 num_chunk_segments

The `num_chunk_segments` field describes the number of chunk segments that the subatom data (associated with the indicated `chunk_segment_id`) has been split into. The value of this field shall be greater than 1.

6.1.11.4 original_subatom_id, original_subatom_id_ext

The `original_subatom_id` field, which is extendable by the `original_subatom_id_ext` field, specifies the `subatom_id` of the segment data contained within the `chunked_subatom()`.

6.1.11.5 oss_bits

The operational variable `oss_bits` is used in the syntax to specify the number of bits used to encode the original subatom data size and is represented by the `num_bits_code()` structure.

6.1.11.6 original_subatom_size

The `original_subatom_size` field describes the size of the original subatom data prior to being chunked, in bytes.

6.1.11.7 byte_align

This bit field of size 0 to 7 bits is used for the byte alignment of the associated `chunked_subatom()` structure. Byte alignment is defined relative to the start of the enclosing syntactic element.

6.1.11.8 chunked_subatom_segment_data

The `chunked_subatom_segment_data` field contains a chunk segment of the chunked subatom data.

The operational variable `cssd_bits` is used in the syntax to specify the number of bits used to encode `chunked_subatom_segment_data` and is based on several fields according to the following semantic equation:

```
  cssd_bits = (subatom_size × 8) - sizeof(chunk_segment_id, chunk_segment_index,
num_chunk_segments, original_subatom_id, original_subatom_id_ext, num_bits_code(),
                        original_subatom_size, byte_align)
```

In the equation above, `subatom_size` is the size indicated in the subatom header for this instance of the `chunked_subatom()`. The `sizeof` function determines the number of bits used by the fields prior to the `chunked_subatom_segment_data`.

6.1.12 block_group_directory()

6.1.12.0 Introduction

The block group directory (`block_group_directory()`) contains fields to indicate the start of block header subatom (i.e. subatoms with the `block_header()` structure) or packet group subatom (i.e. subatoms with the `packet_group()` structure) instances in the bitstream. The block group directory subatom shall not contain any additional padding after the `block_group_directory()` structure.

6.1.12.1 block_group_dir_mask

The `block_group_dir_mask` field is an array of Booleans that describes which subsequent fields are present in the bitstream as listed in Table 65.

Table 65: block_group_dir_mask meaning

`block_group_dir_mask` bit	Meaning
0	Block header directory symbols present: This Boolean indicates whether fields that specify the offset to the block header subatoms are present in the bitstream.
1	Packet group directory: This Boolean indicates whether fields that specify the offset to the packet group subatoms are present in the bitstream.
2	Packet group headers present: This Boolean indicates whether instances of the `packet_group()` structure, one for each of the packet group subatoms, are present and follow in the bitstream. These `packet_group()` instances shall have a `packet_group_type` value of `01b` indicating that only the packet group header is present, no coded symbols. This bit shall only be set if and only if bit 1 is set.
3	Multiple block packet group directory present: This Boolean indicates whether fields that specify the offset to the multiple block packet group subatoms are present in the bitstream.
4-7	Reserved.

6.1.12.2 block_header_subatom_offset[block]

The `block_header_subatom_offset[block]` field describes the offset, in bytes, until the block header subatom with a `block_index` value of `block`. The offset is relative to the '*directory_end*' label.

6.1.12.3 num_packet_groups[block]

The `num_packet_groups[block]` field describes the number of packet group subatom instances present for `block'th` block.

6.1.12.4 packet_group_index[block][pg]

The `packet_group_index[block][pg]` field identifies the `pg'th` instance of the `packet_group()`associated with the `block'th` block.

This field is only present if bit 2 of the `block_group_dir_mask` field is `0`. Otherwise this field is present as part of the `packet_group()` structures in the block group directory.

6.1.12.5 packet_group_subatom_offset[block][pg]

The `packet_group_subatom_offset[block]` field describes the offset, in bytes, until the `pg'th` packet group subatom with a `packet_group_block_index` value of `block`. This field shall only describe the offset to packet group subatoms that have a `packet_group_type` value of `00b` or `10b`, i.e. the `coded_symbol` fields are present. The offset is relative to the '*directory_end*' label.

6.1.12.6 num_multi_block_packet_groups

The `num_multi_block_packet_groups` field describes the number of multiple block packet group subatom instances present.

6.1.12.7 multi_block_packet_group_subatom_offset[mbpg]

The `multi_block_packet_group_subatom_offset[mbpg]` field describes the offset, in bytes, until the instance of the multiple block packet group subatom with a `mpbg__index` value of `mbpg`. The offset is relative to the '*directory_end*' label.

6.1.13 fb_integrity()

6.1.13.0 Introduction

The `fb_integrity()` contains fields to verify the integrity of the received and decoded full file or block.

The `fb_integrity()` element provides a mechanism to verify the integrity of the decoded data in the full file or a given block. This element is referenced by other structures in the syntax When referenced in the `encoder_content_info()` this structure relates to the file. When referenced in the `block_header()` this structure relates to the indicated `block_index`.

6.1.13.1 fb_hash_type

The `fb_hash_type` field describes how the hash on the data is calculated. The meaning of this field is specified in Table 66.

Table 66: fb_hash_type meaning

`fb_hash_type` value	Meaning
`00b`	The hash is calculated on the full file or block data.
`01b`	The hash is a Merkel root hash of the original symbols.
`10b-11b`	Reserved.

6.1.13.2 fb_hash_algorithm

The `fb_hash_algorithm` field describes the algorithm used to calculate the hash of the data. The meaning of this field is specified in Table 67.

Table 67: fb_hash_algorithm meaning

`fb_hash_algorithm` value	Meaning
`000b`	The hash is calculated using the SHA-1 [9] algorithm.
`001b`	The hash is a CRC of the data, modulo the polynomial $x^8 + x^6 + x^5 + x + 1$ where the relevant shift register is initialized to `0xA2` before the computation.
`010b`	The hash is calculated using the MD5 [21] Message-Digest algorithm.
`010b-111b`	Reserved.

6.1.13.3 fb_hash_size

The `fb_hash_size` field describes the number of bits of the hash digest. The meaning of this field is specified in Table 68.

Table 68: fb_hash_size meaning

`fb_hash_size` value	Meaning
`0000b`	The hash digest size is 8 bits.
`0001b`	The hash digest size is 32 bits.
`0010b`	The hash digest size is 64 bits.
`0011b`	The hash digest size is 128 bits.
`0100b`	The hash digest size is 160 bits.
`0101b`	The hash digest size is 256 bits.
`0110b`	The hash digest size is 512 bits.
`0111b`	The hash digest size is 1 024 bits.
`1000b`	The hash digest size is 2 048 bits.
`1001b-1111b`	Reserved.

If the reserved values of `fb_hash_size` (`1001b-1111b`) is received, it is considered an error.

Depending on the value of `fb_hash_algorithm`, `fb_hash_size` is constrained as indicated in Table 69.

Table 69: Allowed fb_hash_size values based on fb_hash_algorithm

`fb_hash_algorithm` value: algorithm	Allowed `fb_hash_size` value: bits
`000b`: SHA-1	`0100b`: 160 bits
`001b`: CRC	`0000b`: 8 bits
`010b`: MD5	`0011b`: 128 bits

The operational variable `fb_hash_bits` is the number of bits indicated by the `fb_hash_size` field. For example, if the `fb_hash_size` field is `0111b`, then `fb_hash_bits = 1 024`.

6.1.13.4 b_fb_integrity_ext

This Boolean indicates whether the `fb_integrity_ext()` structure is present in the bitstream. The `fb_integrity_ext()` structure provides a mechanism to extend the file or block integrity check with additional information and is an instance of the extension() structure.

6.1.13.5 fb_hash

The `fb_hash` field is the hash calculated on the relevant data. Its size (in bits) is determined by the `fb_hash_bits` operational variable. A decoder can use this value to compare against its own calculated hash to ensure that the data was received without error.

6.1.14 packet_integrity()

6.1.14.0 Introduction

The `packet_integrity()` element provides a mechanism to verify the integrity of the data (i.e. coded symbols) in a packet, packet group, or multiple block packet group.

6.1.14.1 packet_hash_algorithm

The `packet_hash_algorithm` field describes the algorithm used to calculate the hash of the data. The meaning of this field is specified in Table 70.

Table 70: packet_hash_algorithm meaning

`packet_hash_algorithm` value	Meaning
`000b`	Reserved
`001b`	The hash is a CRC of the data, modulo the polynomial $x^8 + x^4 + x^3 + x2 + 1$ where the relevant shift register is initialized to zero before the computation.
`010b`	The hash is calculated using the MD5 [21] Message-Digest algorithm.
`010b-111b`	Reserved.

6.1.14.2 packet_hash_size

The `packet_hash_size` field describes the number of bits of the hash digest. The meaning of this field is specified in Table 71.

Table 71: packet_hash_size meaning

`packet_hash_size` value	Meaning
`000b`	The hash digest size is 8 bits.
`001b`	The hash digest size is 32 bits.
`010b`	The hash digest size is 64 bits.
`011b`	The hash digest size is 128 bits.
`100b`	The hash digest size is 256 bits.
`101b`	The hash digest size is 512 bits.
`110b`	The hash digest size is 1 024 bits.
`111b`	The hash digest size is 2 048 bits.

Depending on the value of `packet_hash_algorithm`, `packet_hash_size` is constrained as indicated in Table 72.

Table 72: Allowed packet_hash_size values based on packet_hash_algorithm

`packet_hash_algorithm` value: algorithm	Allowed `packet_hash_size` value: bits
`001b`: CRC	`000b`: 8 bits
`010b`: MD5	`011b`: 128 bits

The operational variable `pkt_hash_bits` is the number of bits indicated by the `packet_hash_size` field. For example, if the `packet_hash_size` field is `110b`, then `pkt_hash_bits = 1 024`.

6.1.14.3 b_packet_integrity_ext

This Boolean indicates whether the `packet_integrity_ext()` structure is present in the bitstream. The `packet_integrity_ext()` structure provides a mechanism to extend the packet integrity check with additional information and is an instance of the `extension()` structure.

6.1.14.4 packet_hash

The `packet_hash` field is the hash calculated on the coded symbol data. The size (in bits) of this field is indicated by the `packet_hash_size` field. A decoder can use this value to compare against its own calculated hash to ensure that the data was received without issue.

The hash is calculated on the `coded_symbol` field(s) contained within the packet, packet group, or multiple block packet group subatoms as indicated by the `/* start hash */` and `/* end hash */` labels.

6.1.15 coefficient_vector()

6.1.15.0 Introduction

The `coefficient_vector()` contains the explicit coding coefficients used to encode the original data symbols for the coded symbol contained within the packet. The element contains a `window_size` worth of coefficients, with each coefficient requiring `block_field_size_exp_val` number of bits.

If the coding coefficient information is encrypted (i.e. bit `3` of the `block_mask` field is set), then this field is symmetrically encrypted using the `block_cce_key` and associated parameters.

6.1.15.1 coded_symbol_coeff[index]

The `coded_symbol_coeff[index]` field is the coefficient for the index*th* original symbol used to encode the coded symbol in the packet. This field requires `block_field_size_exp_val` number of bits.

For more information on `block_field_size_exp`, see clause 6.1.5.23.

6.1.16 extension()

6.1.16.0 Introduction

The `extension()` element defines a mechanism to extend other structures in the bitstream. Multiple instances of this element are referenced in the bitstream and appear in place of substructures not yet defined.

6.1.16.1 extension_byte_size

The `extension_byte_size` field describes the number of subsequent bytes that follow as part of the `extension()` structure. These bytes should be skipped by a parser conforming to the version of the syntax in the present document.

6.1.17 packet_header_only()

The `packet_header_only()` structure contains only packet headers.

Packets, as well as other fields contained within larger structures, inherit values from previous `subatom()` instances. For example, the `packet()` structure inherits the value of the `block_field_size_exp` field specified in the associated field `block_header()`, that is, the `block_header()` with the `block_index` value that matches the `packet_block_index` value.

The `packet_header_only()` structure that is part of the packet header only subatom carries the coding coefficient information describing how the coded symbols were encoded. However, it does not contain the actual `coded_symbol` field. The `packet_header()` will be different depending on whether it is encapsulated in a packet subatom, or in a packet header only subatom.

For more information on `packet_header()`, see clause 6.1.7.

6.1.18 packet_group()

6.1.18.0 Introduction

The packet group (`packet_group()`) structure can contain a `packet_group_header()` as well as coded symbols. The structure is similar to a `packet()` structure. However, unlike the `packet()` structure which contains only single coded symbol, a `packet_group()`can have multiple coded symbols from the same block (i.e. a single block symbol group).

Packet groups, as well as other fields contained within larger structures, inherit values from previous `subatom()` instances. For example, the `packet_group()` structure inherits the value of the `block_field_size_exp` field specified in the associated `block_header()`, that is, the `block_header()` with the `block_index` value that matches the `packet_group_block_index` value.

6.1.18.1 packet_group_block_index

The `packet_group_block_index`, which is extendable by the `packet_group_block_index_ext` field, describes which block this `packet_group()` instance is associated with. If the bitstream only contains a single block (i.e. `block_count` is `1`), then this field is not present in the bitstream and takes on a value of `0`.

As an example, if the `packet_group_block_index` is `1`, then the associated block and its related `block_header()` instance has a `block_index` of `1` as well.

6.1.18.2 packet_group_index

The `packet_group_index` identifies the instance of the `packet_group()`.

6.1.18.3 pgns_bits

The operational variable `pgns_bits` is used in the syntax to specify the number of bits used to encode the `packet_group_num_symbols` field and is represented by the `num_bits_code()` structure.

6.1.18.4 packet_group_num_symbols

The `packet_group_num_symbols` field describes the number of coded symbols present within the packet group.

6.1.18.5 packet_group_type

The `packet_group_type` field describes the type of packet group instance and whether the packet group header and coded symbols are present. The meaning of this field is specified in Table 73.

Table 73: packet_group_type meaning

`packet_group_type` value	Meaning
`00b`	Standard packet group with both `packet_group_header()` and `coded_symbol` fields present.
`01b`	Only `packet_group_header()` present. `coded_symbol` fields are not present.
`10b`	Only `coded_symbol` fields present. `packet_group_header()` not present.
`11b`	Reserved.

6.1.18.6 coded_symbol

A `coded_symbol` is either a systematic symbol or the encoded version of a number of source symbols, generated by applying coding to the original data symbols. The size of this field is indicated by the `block_symbol_size` field in the associated `block_header()` instance. The `block_symbol_size` field shall not use the value `0` when packet group subatoms are present in the bitstream.

6.1.19 packet_group_header()

6.1.19.0 Introduction

The `packet_group_header()` structure provides information about how the coded symbols contained in the `packet_group()` are arranged and encoded.

Because a packet group contains multiple coded symbols, there are less options for how the group of coded symbols can be encoded compared to if each coded symbol had been individually contained within a packet subatom. Within a packet group, the coded symbols can be identified with either a symbol/row index, or a coefficient vector. Sliding windows are not supported within a packet group.

6.1.19.1 packet_group_symbol_arrangement

The `packet_group_symbol_arrangement` field describes the order that coded symbols (including systematic symbols) appear within the packet group subatom. The meaning of this field is specified in Table 74.

Table 74: packet_group_symbol_arrangement Meaning

`packet_group_symbol_arrangement` Value	Symbol Arrangement
`0000b`	Not indicated, the arrangement of coded symbols is not described by a pattern and each symbol is identified by the `packet_symbol_index` field, or by a coefficient vector.
`0001b`	Arbitrary, coded symbols are identified explicitly by the `packet_symbol_index` field.
`0010b`	Arithmetic sequence, symbols are arranged with a constant difference between symbol indexes (e.g. if the first coded symbol in the packet group contains symbol 11, the next coded symbol is 21, then the third coded symbol has symbol index 31 etc.). The sequence is determined using the `packet_group_index_difference` and `packet_group_first_symbol_index` fields.
`0011-1111b`	Reserved.

6.1.19.2 packet_group_mask

The `packet_group_mask` field is an array of Booleans that describes which subsequent fields are present in the bitstream as listed in Table 75.

Table 75: packet_group_mask meaning

`packet_group_mask` bit	Meaning
0	Reserved: This Boolean indicates whether a 32-bit reserved field is present in the bitstream.
1	Symbol index present: This Boolean indicates that an array of symbol indexes (`packet_group_symbol_index`), one for each coded symbol in the packet group, is present in the bitstream. The symbol indexes are either encoding symbol indexes, or row indexes in the case that a block PRNG seed is used.
2	Additional coding coefficient encryption parameter(s) present: This Boolean indicates whether additional encryption parameters relevant to this packet group are present in the bitstream. This bit can only be set if bit 3 of `block_mask` is also set.
3	Coding vector present: This Boolean indicates whether an array of the `coefficient_vector()` structure is present in the bitstream. This array explicitly describes the coding coefficients that were used to generate the coded symbols in the packet group. Note that the `coefficient_vector()` uses `block_num_symbols` instead of `window_size` as an input since sliding windows are not allowed within a packet group.
4	Packet group integrity: This Boolean indicates whether the `packet_group_integrity()` structure, which is an instance of `packet_integrity()`, is present in the bitstream. This structure provides a mechanism to verify that the received data in the packet group matches that of the original source of the bitstream.
5	Packet header extension: This Boolean indicates whether the `packet_header_extension()` structure is present in the bitstream. The `packet_header_extension()` structure provides a mechanism to extend the packet header syntax with additional information and is an instance of the `extension()` structure.

Certain bits of the packet_group_mask are set or cleared depending on the values of other fields. Including:

- Bit 1 (symbol index) and bit 3 (coefficient vector) are mutually exclusive, and shall not both be set simultaneously.
- If bit 3 (coefficient vector) is set, then the value of packet_group_symbol_arrangement shall be 0000b, meaning the coded symbol arrangement is not indicated.
- If addl_block_coding_mask is present in the relevant block_header() instance, bit 1 of the addl_block_coding_mask (indicating that a sliding window is used) shall be cleared.
- If bit 2 of block_mask is set, indicating that the coding coefficients are separated and described in the packet group header only subatom, then:
  - The packet_group_index and packet_group_num_symbols values in the packet group subatom and the associated packet group header only subatom shall have the same value.
  - Bits 1, 2, 3, 4, and 5 of packet_group_mask in packet group subatom shall be cleared. That is, the only allowed packet_group_mask value is 00000Bb (see note). These bits may be set as needed in the associated packet group header only subatom packet_group_mask field.
- If bit 3 of block_mask is set, indicating that the coding coefficients are encrypted, then bit 2 of packet_group_mask (additional encryption parameters) shall only be set if and only if bit 3 of packet_group_mask (coding vector present) is set.
- If bit 4 of block_mask is set, indicating that a block PRNG seed is used, then bit 1 of packet_group_mask (symbol index present) shall be set, and the value of packet_group_symbol_arrangement shall be 0000b, meaning the coded symbol arrangement is not indicated.

NOTE: Bit values indicated as B mean that the value of this bit can be set, or not set depending on other settings in an encoder.

An example packet_group_mask can be 011000b, with bit 3 set indicating the array of coefficient_vector fields is present in the bitstream, and bit 4 set indicating the presence of the packet group integrity check.

6.1.19.3 pgsi_bits

The operational variable pgsi_bits is used in the syntax to specify the number of bits used to encode the packet_group_symbol_index fields and is represented by the num_bits_code() structure.

6.1.19.4 packet_group_symbol_index

For each coded symbol in the packet group, the packet_group_symbol_index field provides either an encoded symbol index value (for code types such as Reed-Solomon, or RaptorQ), or row index value in the case that a block PRNG seed is used.

6.1.19.5 pgfsi_bits

The operational variable pgfsi_bits is used in the syntax to specify the number of bits used to encode the packet_group_first_symbol_index fields and is represented by the num_bits_code() structure.

6.1.19.6 packet_group_first_symbol_index

The packet_group_first_symbol_index field describes the symbol index of the first coded symbol in the packet group. Subsequent symbol indexes are then calculated using this value and the value of the packet_group_index_difference field.

6.1.19.7 packet_group_index_difference

The packet_group_index_difference field describes the symbol index difference between successive coded symbols in a packet group. The value 0 shall not be used for this field.

For example, if the first coded symbol in a packet group has a symbol index value of 6, and the packet_group_index_difference field has a value of 3, then the second coded symbol has a symbol index of 9.

6.1.19.8 pgfsii_bits

The operational variable pgsii_bits is used in the syntax to specify the number of bits used by the packet_group_symbol_index_info structure which is an instance of the extension() structure. pgsii_bits is represented by the num_bits_code() structure.

6.1.19.9 Symbol Arrangements in a Packet Group

Within a packet group subatom, a group of coded symbols from the packet_group_block_index'th block are arranged as described by the packet_group_symbol_arrangement field.

When the coded symbol arrangement in a packet group is not indicated (the value of the packet_group_symbol_arrangement is 0000b) or is arbitrary (the value of the packet_group_symbol_arrangement is 0001b), and bit 1 of packet_group_mask (symbol index present) is set, then each coded symbol can be identified by an instance of the packet_group_symbol_index field present in the bitstream.

When the coded symbols in a packet group are arranged in an arithmetic sequence (the value of the packet_group_symbol_arrangement is 0010b), and bit 1 of packet_group_mask (symbol index present) is set, then the value of packet_group_symbol_index for the n'th coded symbol in the packet group is calculated using the packet_group_first_symbol_index and packet_group_index_difference fields according to the equation:

```
packet_group_symbol_index[n] = packet_group_first_symbol_index +
                (n x packet_group_index_difference)
```

Where n = 0,1,2,… and represents the n^{th} coded symbol in the packet group.

If the value of the packet_group_symbol_arrangement is 0010b, a CMMF decoder shall determine the packet_group_symbol_index value for each coded symbol using the pseudocode in Table 76.

Table 76: Arithmetic coded symbol arrangement in a packet group pseudocode

Pseudocode

```
packet_group_arithmetic_arrangement(packet_group_num_symbols, packet_group_first_symbol_index,
   packet_group_index_difference)
{
   packet_group_symbol_index[packet_group_num_symbols] = {0};
   symbol_index = packet_group_first_symbol_index;
   for (n = 0; n < packet_group_num_symbols; n++) {
      packet_group_symbol_index[n] = symbol_index;
      symbol_index = symbol_index + packet_group_index_difference;
   }
   return;
}
```

6.1.19.10 Encryption Parameters

6.1.19.10.0 Introduction

When bit 2 of the packet_group_mask field is set, additional encryption parameters needed to decrypt the coding coefficient information within the packet_group_header() are present in the bitstream. The encrypted coding coefficient information within the packet_group_header() can be the instances of the coefficient_vector() in packet_group_header() within the packet group subatom or the packet group header only subatom

When the coding coefficient information is encrypted, keys and other encryption parameters are needed in order to decrypt the coefficient information and ultimately decode the coded symbols.

The following fields describe the additional packet group or packet group header only encryption parameters.

For more information on encrypting coding coefficient information, see clause 7.3.

6.1.19.10.1 b_addl_packet_group_cce_params_present

This Boolean indicates whether additional encryption parameters required to decrypt the coding coefficient information, contained in the `addl_cce_parameters()` structure, follow in the bitstream.

6.1.20 num_bits_code()

6.1.20.0 Introduction

The `num_bits_code()`is a helper tool used to describe the number of bits used by a field in the bitstream. This tool returns the `num_bits` variable value.

6.1.20.1 bits_code

The `bits_code` field describes the number of bits a subsequent field in the bitstream uses and returned by the `num_bits` variable. The meaning of this field is specified in Table 77.

Table 77: bits_code meaning

`bits_code` value	Meaning
`00b`	The value of `num_bits` is 8 bits.
`01b`	The value of `num_bits` is 16 bits.
`10b`	The value of `num_bits` is 24 bits.
`11b`	The value of `num_bits` is 32 bits.

The operational variable `num_bits` can be calculated directly from the `bits_code` field according to the following equation:

```
num_bits = (bits_code + 1) × 8
```

6.1.21 block_index_or_count_value()

6.1.21.0 Introduction

The `block_index_or_count_value()` structure is a helper tool used to describe either the block index or block count as required within the context of the encompassing structure. This tool returns the `block_index_or_count_val` variable value.

6.1.21.1 block_index_or_count, block_index_or_count_ext

The `block_index_or_count` field, which is extendable by the `block_index_or_count_ext` field, describes either a block index or a block count as appropriate by the context.

The operational variable `block_index_or_count_val` can be calculated directly from the `block_index_or_count` and `block_index_or_count_ext` fields according to the following equation:

```
block_index_or_count_val = block_index_or_count + block_index_or_count_ext
```

6.1.22 multi_block_packet_group()

6.1.22.0 Introduction

The multiple block (or multi-block) packet group (`multi_block_packet_group()`) contains the `mbpg_header()` as well as coded symbols. The structure is similar to the `packet_group()`structure. However, unlike the `packet_group()` structure which contains coded symbols associated with a single block, a `multi_block_packet_group()` structure may carry coded symbols from multiple blocks (i.e. a multiple block symbol group).

There are several restrictions for the use of a multi-block packet group. Including:

- The multi-block packet group shall only be used when the associated code type utilizes encoded symbol indexes to identify each symbol or row indexes to identify the row of a coding coefficient matrix generated by a block PRNG. Code types that transmit coefficient vectors are not supported. The use of sliding windows is also not supported.
- Within a multi-block packet group, all blocks shall have identical symbol size (`block_symbol_size`), and if applicable, shall have identical field size (`block_field_size_exp_val`) values.
- The blocks represented in a multi-block packet group shall be from a contiguous set of blocks.
- If `addl_block_coding_mask` is present in the relevant `block_header()` instances, bit 1 of the `addl_block_coding_mask` (indicating that a sliding window is used) shall be cleared.
- The `block_mask` field in the relevant block header subatom instances for the blocks represented in a multi-block packet group shall have:
  - Bit 1 (block field size) and bit 4 (block seed) shall be set to the same value across all instances.
  - Bit 2 (separated coding coefficients) and bit 3 (encrypted coding coefficient information) cleared in all instances.

Multi-block packet groups, as well as other fields contained within larger structures, inherit values from previous `subatom()` instances. For example, the `multi_block_packet_group()` structure inherits the value of the `block_symbol_size` field specified in the associated block header subatom instances.

6.1.22.1 mbpg_index

The `mbpg_index` field identifies the `multi_block_packet_group()` instance.

6.1.22.2 mbpg_start_block_index

The `mbpg_start_block_index` field identifies the start block (i.e. first block) of the set of blocks represented in the multi-block packet group.

6.1.22.3 mbpg_num_blocks

The `mbpg_num_blocks` field describes the number of blocks present within the multi-block packet group.

The multi-block packet group contains blocks with `block_index` values in range [`mbpg_start_block_index`, `mbpg_start_block_index + mbpg_num_blocks - 1`]. For example, if `mbpg_start_block_index` has a value of 2, and `mbpg_num_blocks` has a value of 3, then the multi-block packet group contains symbols from blocks 2, 3, and 4.

6.1.22.4 mbpg_num_symbols

The `mbpg_num_symbols` field describes the number of coded symbols present within the multi-block packet group.

6.1.22.5 coded_symbol

A `coded_symbol` is either a systematic symbol or the encoded version of several source symbols, generated by applying coding to the original data symbols. The size of this field is indicated by the `block_symbol_size` field in the associated block header subatom instance as determined by the `mbpg_start_block_index` value. The `block_symbol_size` field shall not use the value `0` when multi-block packet group subatoms are present in the bitstream.

6.1.23 mbpg_header()

6.1.23.0 Introduction

The `mbpg_header()` structure provides information about how the coded symbols contained in the `multi_block_packet_group()` are arranged.

Because a multiple block (or multi-block) packet group contains multiple coded symbols from different blocks, there are less options for how the group of coded symbols can be encoded compared to if each coded symbol had been individually contained within a packet subatom. Within a multi-block packet group, the coded symbols can only be identified by a symbol/row index. Coefficient vectors, and sliding windows are not supported within a multi-block packet group. Further, encryption is not supported for any fields within a multi-block packet group.

6.1.23.1 mbpg_symbol_arrangement

The `mbpg_symbol_arrangement` field describes the order that coded symbols (including systematic symbols) appear within the multi-block packet group subatom. The meaning of this field is specified in Table 78.

Table 78: mbpg_symbol_arrangement Meaning

`mbpg_symbol_arrangement` value	Symbol Arrangement
`0000b`	Not indicated, the arrangement of coded symbols is not described by a pattern and each symbol is identified by the values of `mbpg_source_block_index` and the `mbpg_symbol_index` field.
`0001b`	Explicitly-Identified Symbol Arrangement: Arbitrary, coded symbols are identified explicitly by the values of the `mbpg_source_block_index` and the `mbpg_symbol_index` fields.The arrangement of coded symbols in this case may follow some unspecified pattern.
`0010b`	Source-Symbol Interleaved Arrangement: Separate systematic symbol and interleaved coded symbol arithmetic sequence, systematic symbols are arranged in an arithmetic sequence (i.e. constant difference between symbol index values) by block, then coded symbols are also arranged with a constant difference between symbol indexes, with each coded symbol being from a different block.
`0011b`	Coded-Symbol Interleaved Arrangement: Interleaved by block arithmetic sequence, coded symbols with the same symbol index from different blocks are placed followed by the set of symbols with the next symbol index (e.g. symbol 11 from block 0, symbol 11 from block 1, symbol 12 from block 0, symbol 12 from block 1, etc.).
`0100b-1111b`	Reserved.

Certain values of the `mbpg_symbol_arrangement` field impose additional constraints on other fields. Including:

- If the value of `mbpg_symbol_arrangement` is 0010b or 0011b, then the `b_block_max_symbol_index_present` shall be TRUE and the `block_max_symbol_index` field shall be present in each relevant block header subatom instance.
- If bit 4 of `block_mask` is set in the relevant block header subatom instance, indicating that a block PRNG seed is used, then the value of `mbpg_symbol_arrangement` shall be 0000b, meaning the coded symbol arrangement is not indicated.

6.1.23.2 mbpgsi_bits

The operational variable `mbpgsi_bits` is used in the syntax to specify the number of bits used to encode the `mbpg_symbol_index` field and is represented by the `num_bits_code()` structure.

6.1.23.3 mbpg_source_block_index, mbpg_symbol_index

For each coded symbol in the multiple block packet group the value of `mbpg_source_block_index` identifies the block the symbol comes from (i.e. the source block number), and the `mbpg_symbol_index` field provides either an encoded symbol index (for code types such as Reed-Solomon, or RaptorQ), or a row index in the case that a block PRNG seed is used. The block and symbol index uniquely identify the coded symbol.

6.1.23.4 mbpgfsi_bits

The operational variable `mbpgfsi_bits` is used in the syntax to specify the number of bits used to encode the `mbpg_first_symbol_index` field and is represented by the `num_bits_code()` structure.

6.1.23.5 mbpg_first_symbol_index, b_mbpg_is_symbol_group_subset, mbpg_symbol_group_subset_index

After arrangement to symbol groups, a multi-block packet group may contain a contiguous subset of symbols from a symbol group.

The `b_mbpg_is_symbol_group_subset` Boolean indicates whether multi-block packet group contains a subset or an entire symbol group. When TRUE, the multiple block packet group contains a subset of symbols from the symbol group and the `mbpg_symbol_group_subset_index` field is present in the bitstream.

The `mbpg_first_symbol_index` field defines the symbol index of the first symbol, from the first block (as indicated by `mbpg_start_block_index` field), of the contained symbols in the symbol group associated with the multi-block packet group.

The `mbpg_symbol_group_subset_index` field defines the index into the symbol group to form the subset contained within the multi-block packet group. When `b_mbpg_is_symbol_group_subset` is FALSE, the value of `mbpg_symbol_group_subset_index` shall be equal to 0.

6.1.23.6 mbpg_index_difference

The `mbpg_index_difference` field describes the symbol index difference between successive coded symbols in a multi-block packet group. The value of this field may also correspond to the total number of symbol groups. The value 0 shall not be used for this field.

For example, if the first coded symbol in a multi-block packet group has a symbol index value of 6, and the `mbpg_index_difference` field has a value of 3, then the next coded symbol from the same block has a symbol index of 9.

6.1.23.7 mbpgsai_bits_code

The `mbpgsai_bits_code` field describes the number of bits used by the `mbpg_symbol_arrangement_info` structure which is an instance of the `extension()` structure. The value of the `mbpgsai_bits_code` field is specified in Table 79.

Table 79: mbpgsai_bits_code meaning

`mbpgsai_bits_code` value	Meaning
0	8 bits
1	16 bits
2	32 bits
3	64 bits

The operational variable `mbpgsai_bits` uses the `mbpgsai_bits_code` value to determine the number of bits the subsequent `mbpg_symbol_arrangement_info` structure requires. It can also be calculated as follows:

```
mbpgsai_bits = 2^(mbpgsai_bits_code + 3)
```

6.1.23.8 Symbol Arrangements in a Multiple Block Packet Group

Within a multiple block packet group subatom, a group of coded symbols from blocks `mbpg_start_block_index` … `mbpg_start_block_index+mbpg_num_blocks-1` are arranged into symbol groups as described by the `mbpg_symbol_arrangement` field. This symbol group is either directly used in its entirety, or a subset is used to form the multiple block packet group.

When the coded symbol arrangement in a multi-block packet group is not indicated (the value of the `mbpg_symbol_arrangement` is 0000b) or is an explicitly-identified symbol arrangement (the value of the `mbpg_symbol_arrangement` is 0001b), then each coded symbol can be identified by instances of the `mbpg_source_block_index` field `mbpg_symbol_index` fields present in the bitstream.

When the coded symbols in a multi-block packet group are arranged in a source-symbol interleaved arrangement (the value of the `mbpg_group_symbol_arrangement` is 0010b), then the values of `mbpg_source_block_index` and `mbpg_symbol_index` are calculated using the `block_num_symbols` and `block_max_symbol_index` from each relevant block header subatom instance, `mbpg_first_source_block_index`, `mbpg_num_blocks`, `mbpg_first_symbol_index`, `mbpg_index_difference`, and `mbpg_symbol_group_subset_index` fields. An arithmetic sequence of systematic symbols from block `mbpg_start_block_index` are present, followed by an arithmetic sequence of systematic symbols from block `mbpg_start_block_index+1`, and so on until block `mbpg_start_block+mbpg_num_blocks-1`. In the context of a multi-block packet group, a systematic symbol has an `mbpg_symbol_index` value less than `block_num_symbols` for a given block. Then coded symbols are arranged with a constant difference between symbol indexes, however, each coded symbol comes from a different block.

If the value of the `mbpg_symbol_arrangement` is 0010b, a CMMF decoder shall determine the `mbpg_source_block_index` and `mbpg_symbol_index` values for each coded symbol using the pseudocode in Table 80.

Table 80: Separate systematic and interleaved coded symbol arithmetic sequences symbol arrangement in a multi-block group pseudocode

Pseudocode

```
def mbpg_arrangement_0010b(
      block_info, # contains block_num_symbols and block_max_symbol_index for each block
      mbpg_start_block_index,
      mbpg_num_blocks,
      mbpg_num_symbols,
      mbpg_index_difference,
      mbpg_first_symbol_index,
      mbpg_symbol_group_subset_index,
      ):
   # source and coded/repair symbol pattern arrays
   cr_pattern_block_indexes = []
   cr_pattern_symbol_indexes = []
   ss_pattern_block_indexes = []
   ss_pattern_symbol_indexes = []
   # symbol group arrays
   symbol_group_block_indexes = []
   symbol_group_symbol_indexes = []
   # multi-block packet group (or subset) arrays
   mbpg_source_block_index = []
   mbpg_symbol_index = []

   # ----- arrange all symbols according to pattern -----
   # arrange source symbols and get source symbols in symbol
   num_source_symbols = 0
   for blk in range(mbpg_start_block_index,(mbpg_start_block_index+mbpg_num_blocks)):
      for symbol in range(block_info[blk]['block_num_symbols']):
         ss_pattern_block_indexes.append(blk)
         ss_pattern_symbol_indexes.append(symbol)
         num_source_symbols += 1

   # create arrays for coded/repair symbols
   # find max number of coded/repair symbols of all blocks
```

Pseudocode

```
    max_num_cr_symbols = 0
    for blk in range(mbpg_start_block_index,(mbpg_start_block_index+mbpg_num_blocks)):
        num_cr_symbols_in_blk = block_info[blk]['max_symbol_index'] - block_info[blk]['block_num_symbols'] + 1
        max_num_cr_symbols = max(max_num_cr_symbols, num_cr_symbols_in_blk)

    # create arrays of length max_num_cr_symbols
    for blk in range(mbpg_start_block_index,(mbpg_start_block_index+mbpg_num_blocks)):
        cr_blocks = [-1] × max_num_cr_symbols
        cr_symbols = [-1] × max_num_cr_symbols
        idx = 0
        for si in range(block_info[blk]['block_num_symbols'], block_info[blk]['max_symbol_index']+1):
            cr_blocks[idx] = blk
            cr_symbols[idx] = si
            idx += 1
        cr_pattern_block_indexes.append(cr_blocks)
        cr_pattern_symbol_indexes.append(cr_symbols)

    # ----- extract symbol group -----
    # source symbols
    for idx in range(mbpg_first_symbol_index, num_source_symbols, mbpg_index_difference):
         symbol_group_block_indexes.append(ss_pattern_block_indexes[idx])
         symbol_group_symbol_indexes.append(ss_pattern_symbol_indexes[idx])

    # coded/repair symbols
    for idx in range(mbpg_first_symbol_index, max_num_cr_symbols, mbpg_index_difference):
        for blk in range(mbpg_start_block_index,(mbpg_start_block_index+mbpg_num_blocks)):
            if (cr_pattern_symbol_indexes[blk][idx] != -1) and (cr_pattern_symbol_indexes[blk][idx] <=
block_info[blk]['max_symbol_index']):
                symbol_group_block_indexes.append(cr_pattern_block_indexes[blk][idx])
                symbol_group_symbol_indexes.append(cr_pattern_symbol_indexes[blk][idx])

    # ----- extract multi-block-packet group (or smaller subset) from symbol group -----
    mbpg_source_block_index =
symbol_group_block_indexes[mbpg_symbol_group_subset_index:(mbpg_symbol_group_subset_index+mbpg_num_s
ymbols)]
    mbpg_symbol_index       =
symbol_group_symbol_indexes[mbpg_symbol_group_subset_index:(mbpg_symbol_group_subset_index+mbpg_num
_symbols)]

    return mbpg_source_block_index, mbpg_symbol_index
```

An example symbol arrangement 0010b (Source-Symbol Interleaved Arrangement) is shown in Figure 12. In this example, symbols from three blocks (0, 1, and 2) are represented, where:

- `block_num symbols` is 10 and `block_max_symbol_index` is 30 for block 0
- `block_num symbols` is 10 and `block_max_symbol_index` is 30 for block 1
- `block_num symbols` is 9 and `block_max_symbol_index` is 30 for block 2
- `mbpg_num_symbols` is 30, 30, 29
- `mbpg_symbol_group_subset_index` is 0, 0, 0
- `mbpg_first_symbol_index` is 0, 1, 2
- `mbpg_index_difference` is 3

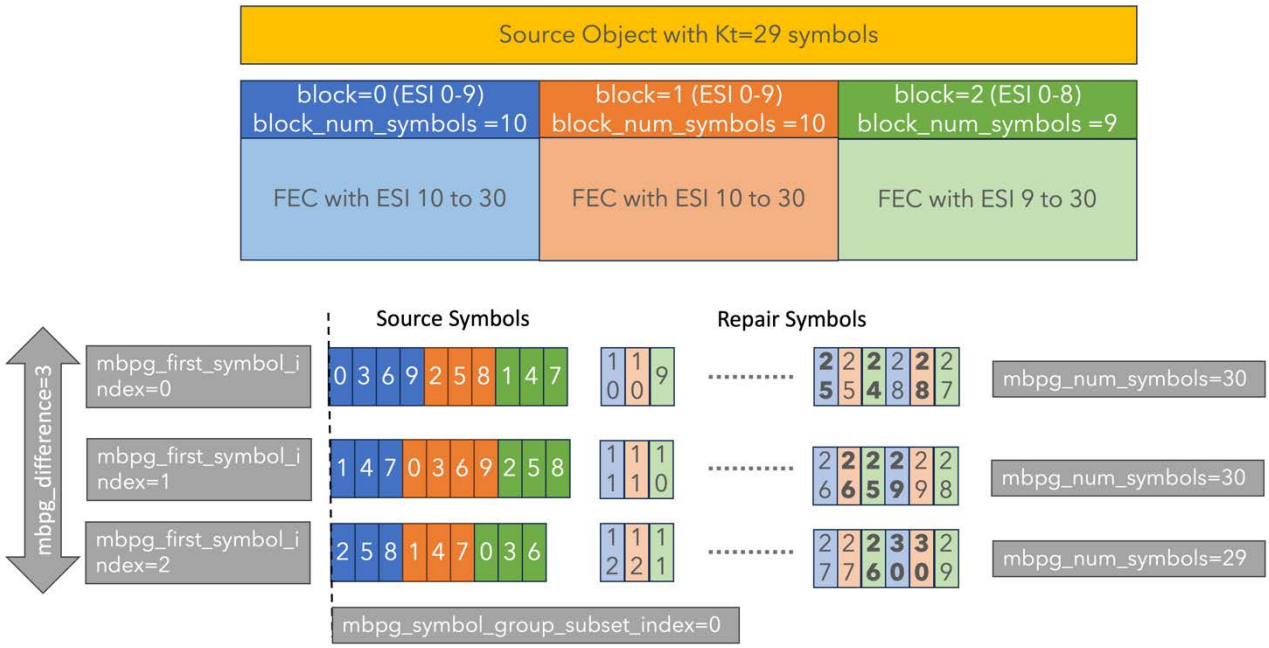

Figure 12: Example symbol arrangement for mbpg_symbol_arrangement=0010b (Source-Symbol Interleaved Arrangement)

When the coded symbols in are arranged in a coded-symbol interleaved arrangement (the value of the `mbpg_symbol_arrangement` is 0011b), then the values of `mbpg_source_block_index` and `mbpg_symbol_index` are calculated using the `block_max_symbol_index` from each relevant block header subatom instance, `mbpg_start_block_index`, `mbpg_num_blocks`, `mbpg_first_symbol_index`, `mbpg_index_difference`, and `mbpg_symbol_group_subset_index` fields. The coded symbols are arranged in an arithmetic sequence; however, each coded symbol comes from a different block.

If the value of the `mbpg_symbol_arrangement` is 0011b, a CMMF decoder shall determine the `mbpg_source_block_index` and `mbpg_symbol_index` values for each coded symbol using the pseudocode in Table 81.

Table 81: Interleaved by block coded symbol arithmetic sequences symbol arrangement in a multi-block group pseudocode

Pseudocode

```
def mbpg_arrangement_0011b(
      block_info, # contains block_num_symbols and block_max_symbol_index for each block
      mbpg_start_block_index ,
      mbpg_num_blocks,
      mbpg_num_symbols,
      mbpg_index_difference,
      mbpg_first_symbol_index,
      mbpg_symbol_group_subset_index,
      ):
   # source and coded symbol pattern arrays
   coded_symbol_pattern_block_indexes = []
   coded_symbol_pattern_symbol_indexes = []
   # symbol group arrays
   symbol_group_block_indexes = []
   symbol_group_symbol_indexes = []
   # multi-block packet group (or subset) arrays
   mbpg_source_block_index = []
   mbpg_symbol_index = []

   # ----- arrange all symbols according to pattern -----
   # find max number of coded/repair symbols of all blocks
   max_num_coded_symbols = 0
   for blk in range(mbpg_start_block_index,(mbpg_start_block_index+mbpg_num_blocks)):
      num_coded_symbols_in_blk = block_info[blk]['max_symbol_index'] + 1
      max_num_coded_symbols = max(max_num_coded_symbols, num_coded_symbols_in_blk)

   # create arrays of length max_num_coded_symbols
   for blk in range(mbpg_start_block_index,(mbpg_start_block_index+mbpg_num_blocks)):
      cr_blocks = [-1] × max_num_coded_symbols
      cr_symbols = [-1] × max_num_coded_symbols
      idx = 0
      for si in range(block_info[blk]['max_symbol_index']+1):
         cr_blocks[idx] = blk
   return mbpg_source_block_index, mbpg_symbol_index
```

An example symbol arrangement 0011b (Encoded-Symbol Interleaved Arrangement) is shown in Figure 13. In this example, symbols from three blocks (0, 1, and 2) are represented, where:

- `block_num symbols` is 10 and `block_max_symbol_index` is 29 for block 0
- `block_num symbols` is 10 and `block_max_symbol_index` is 29 for block 1
- `block_num symbols` is 9 and `block_max_symbol_index` is 26 for block 2
- `mbpg_num_symbols` is 25, 25, 25
- `mbpg_symbol_group_subset_index` is 4 for each object
- `mbpg_first_symbol_index` is 0, 1, 2
- `mbpg_index_difference` is 3

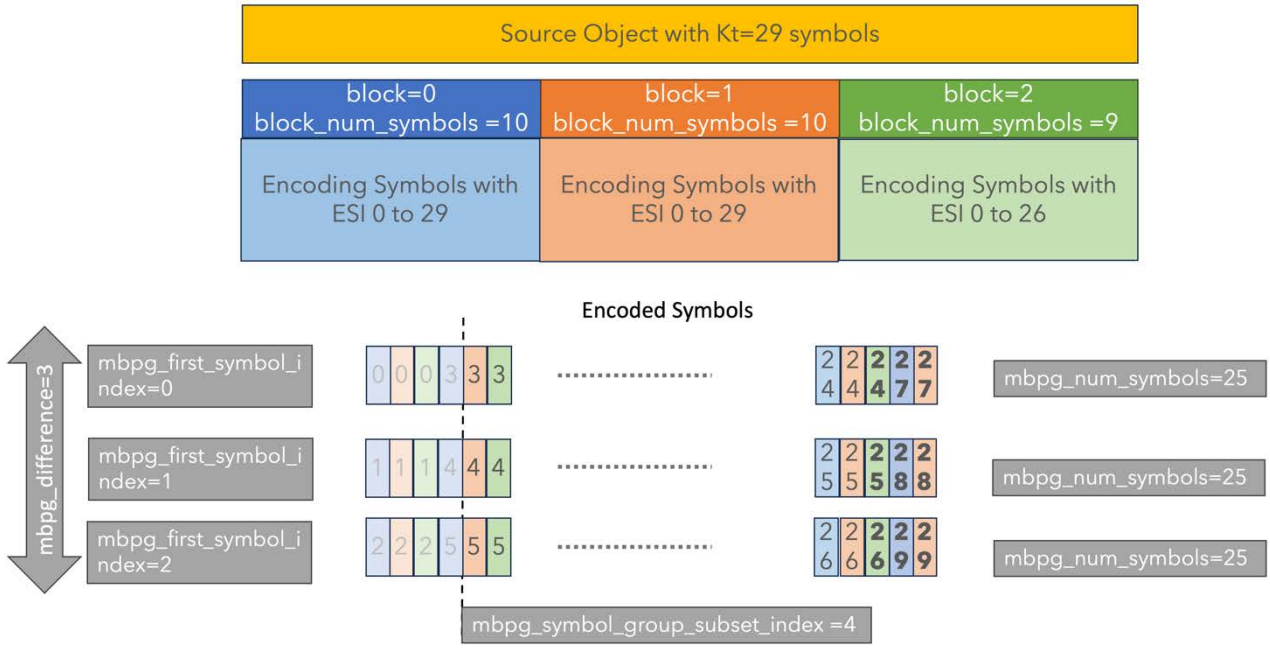

Figure 13: Example symbol arrangement for mbpg_symbol_arrangement=0011b (Encoded-Symbol Interleaved Arrangement)

6.1.23.9 b_mbpg_integrity_present

This Boolean indicates whether the `mbpg_integrity()` structure is present in the bitstream. The `mbpg_integrity()` structure is an instance of the `packet_integrity()` structure that applies to the group of coded symbols within the multiple block packet group.

6.1.23.10 b_mbpg_header_ext_present

This Boolean indicates whether the `mbpg_header_extension()` structure is present in the bitstream. The `mbpg_header_extension()` structure provides a mechanism to extend the packet header syntax with additional information and is an instance of the `extension()` structure.

7 Design considerations

7.0 Introduction

This clause describes design considerations for encoders, and parsers/decoders.

7.1 Coding coefficients

7.1.0 Generating coding coefficients using a PRNG

A PRNG can be used to generate coding coefficients that are applied to source symbols to create coded symbols. Rather than transmit the coding coefficients in the bitstream, the means for generating the coefficients are parameterized and the resulting parameters are transmitted for a decoder to regenerate the coefficients.

The bitstream supports transmission of a seed per block or per packet via the fields in `prng_parameters()`. The block or packet seed in the bitstream is used to initialize the PRNG.

A density controller (that also uses the PRNG) controls the amount of non-zero coefficients. The density controller uses either the block or packet `prng_density_percentage` field(s) to determine whether the coefficient should be set to `0`.

When a block seed is used, rows of `block_num_symbols` worth of columns are generated, starting with the first index (`0`) until the last index (`block_num_symbols - 1`) for the first row, and repeating for the subsequent rows to generate a complete coding matrix. At least `block_num_symbols` worth of rows need to be generated to be able to represent the entire source block. Once the rows are generated, the `packet_symbol_index` value in arriving packet subatoms identifies which row of the generated coding matrix was applied to the `coded_symbol`.

For a given packet, packet group, or multi-block packet group subatom, if a block PRNG is used and the `b_systematic_symbol` field is set, or if the row index is less than `block_num_symbols`, then the identified row of coefficients generated by the block PRNG is discarded and replaced with a coefficient row vector consisting of all `0`'s except at the index identified by the `packet_symbol_index` , or `packet_group_symbol_index`, or `mbpg_symbol_index` value, where the coefficient is set to `1`. For example, assuming a `block_field_size_exp_val` of `1`, if a systematic packet's `packet_symbol_index` value is `2`, and the block PRNG generated has a coefficient row of [0 1 1 0 1 1], then this will be replaced with [0 0 1 0 0 0].

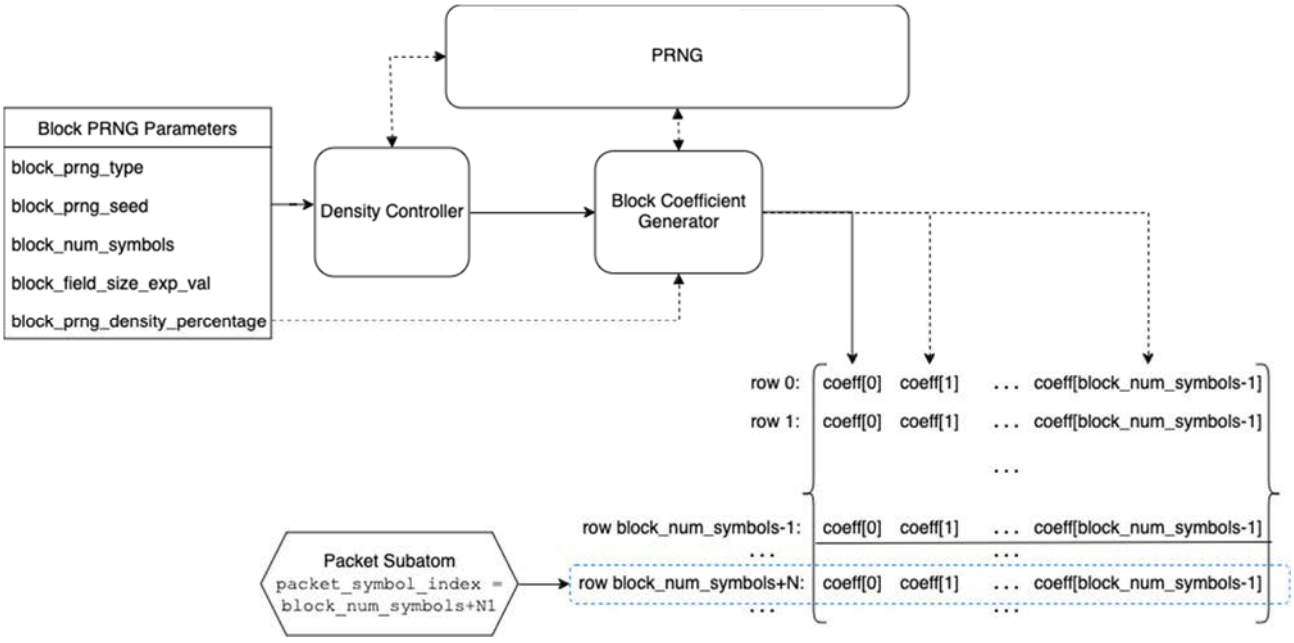

Figure 14: Block PRNG

When a packet seed is used, a `window_size` worth of symbols is generated starting with the first index (`0`) of the `coefficient_vector`, and ending with the final index (`window_size - 1`).

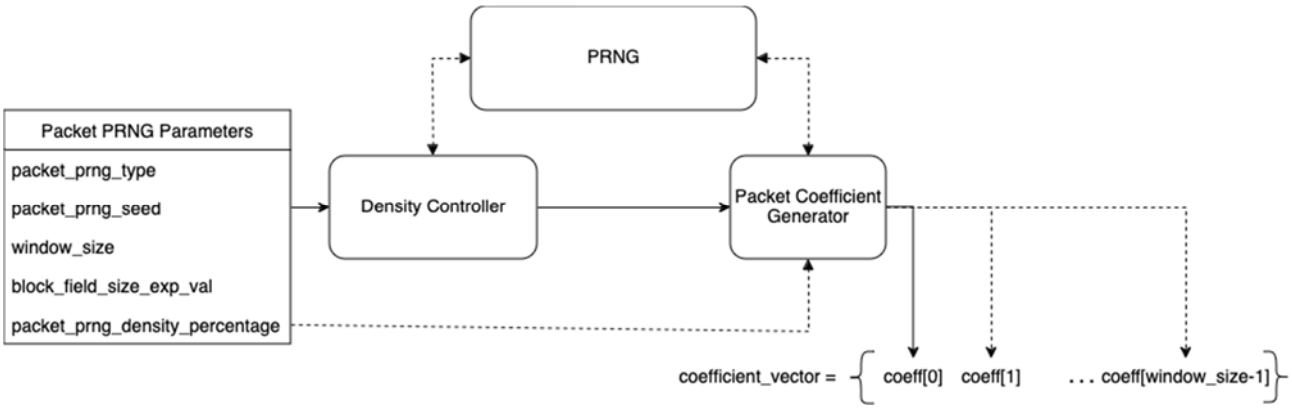

Figure 15: Packet PRNG

7.1.1 Coefficient density control

The density controller manages the sparsity of non-zero coefficients. The controller can be thought of a function that accepts a number generated by the PRNG and the density percentage value for either the block or packet (as an integer, `density_int`) and returns either a 0 or a 1. When the value is 1, then the block (or packet) coefficient generator block generates a coefficient. Pseudocode for the density control function is shown in Table 82.

Table 82: Density controller pseudocode

Pseudocode

```
density_controller(prng_generated_number, density_int)
{
   random_int = prng_number() % 128 + 1;
   if (random_int >= density_int) {
      return 0;
   } else {
      return 1;
   }
}
```

7.1.2 Mersenne twister PRNG type

When either the block or packet `prng_type` field has a value of `000b`, a Mersenne Twister (MT) based PRNG is used. The standardized MT19937 [8] shall be used. The PrngMT class shown in the pseudo-code in Table 83 takes the seed, density (as an integer) and the field size exponent, controls the density and subsequently generates coefficient numbers.

Table 83: Mersenne twister pseudocode

Syntax

```
class PrngMT {
public:
{
   prngMT(unsigned long seed) {
      m_mt.seed(seed)
   }

   unsigned long get_coefficient(unsigned int density_int, int field_size_exp)
   {
      // Step 1: use density to determine if coefficient should be zero
      auto density_rand = m_mt();
      if (density_rand % 128 + 1 >= density_int) {
         return 0;
      } else {
         // Step 2: Pick a random non-zero coefficient.
         if (field_size_exp == 1) {
            // Special case of binary field needs no random input
            return 1;
         }
         auto coeff_rand = m_mt();
         return (coeff_rand % ((1 << field_size_exp) - 1)) + 1;
      }
   }
private:
   std::mt19937 m_mt;
}
```

7.2 Handling variable source symbol size

Applications of the CMMF bitstream may include cases where the size of the data source symbols is not constant. Since most code types require equal-sized source symbols, the CMMF bitstream format was designed not to support explicit description of the size for each symbol.

When the source symbols have varying sizes, applications shall prepend the size of the source symbol to the data before passing it to the CMMF bitstream encoder. This size information becomes part of the data source symbol from the encoder's perspective and will be encoded. Within the encoder, an appropriate amount of zero-padding is applied to the data to equalize the lengths for encoding. The encoder may choose to make the length to be that of the largest data source symbol among the symbols to be encoded together. Another option available to the encoder is to apply a global maximum length and pad all the symbols to a uniform size. If the encoder chooses to transmit a symbol within a systematic packet, and the `block_symbol_size` field in the associated `block_header()` instance is `0`, then the subatom size can be shortened to omit the padding within the `coded_symbol`.

Following decoding, a size/padding peeler is used to parse the size marker, strip the size data and any excess padding to obtain the original source symbol and provide the data to the application.

An example of how such a system may work is shown in Figure 16.

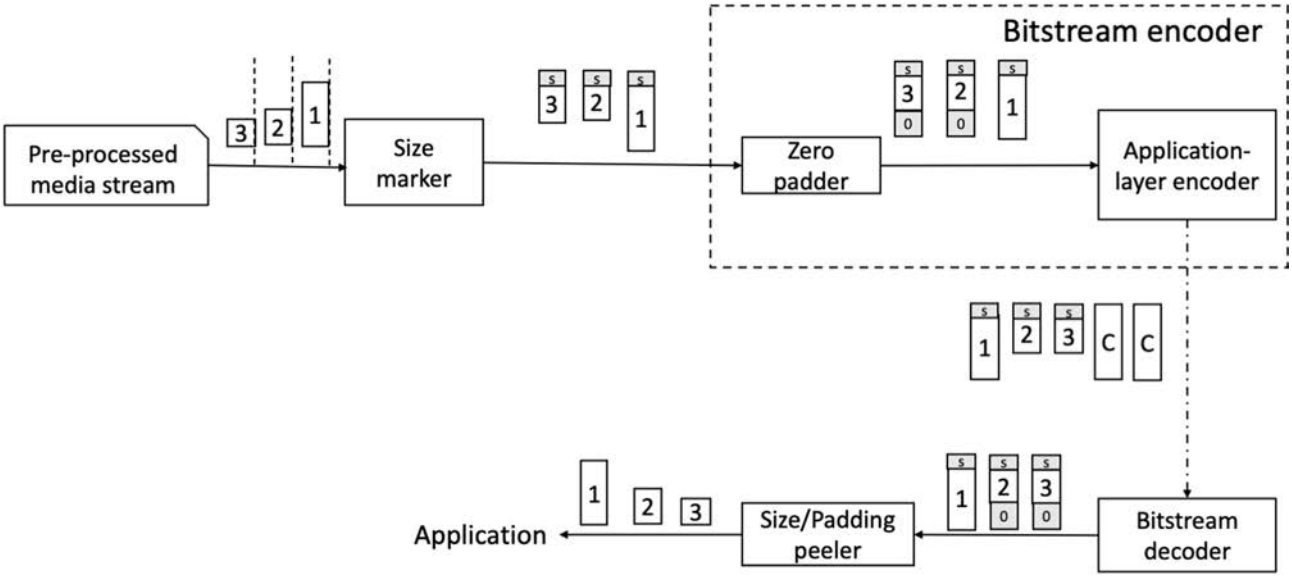

Figure 16: Handling variable size data

In the example above, three symbols (P1, P2, P3) of different sizes are encoded together. Since P1 is the largest, the size marker sets the symbol length to the length of size prepended P1. Thus, P2 and P3 will be padded with zeros before encoding.

Again, the encoder may choose not to pad symbols that will be sent in systematic packets. When a decoder receives a systematic packet, the size prepended symbol is passed to the size/padding peeler which will parse the size revealing that there is no padding to remove, only the size marker. In Figure 16 above, assume transmission of P1 is successful, so the decoder receives the size prepended P1 with no additional padding, which is passed to the size/padding peeler.

For an encoded symbol, the encoder pads each source symbol to equal size. When the decoder receives an encoded symbol, after decoding the size marker will be present in the recovered symbol, and potentially may have some amount of padding. In the example above, assume the transmission of P2 failed, so P2 is retrieved via decoding of the coded packets. The decoded P2 has both the size marker and padding which will be removed by the size/padding peeler.

7.3 Encrypting coding coefficient information

7.3.0 Introduction

The CMMF bitstream supports encryption of coding coefficient information. Encrypting the coding coefficient information can provide an extra layer of security for protecting content. Depending on the infrastructure an application uses, different methods for handling the necessary encryption keys can be used.

7.3.1 Using a bitstream/session key and symmetric keys

As an example of one method to handle encryption of coding coefficient information, a bitstream or session key, and symmetric keys can be used. In this example, there a single bitstream consisting of multiple blocks.

This method uses a combination of keys to provide secure delivery of coefficient vectors to clients. Each coefficient vector is encrypted using a stream cypher keyed with a symmetric key unique to each block. These block unique symmetric keys are themselves encrypted using a stream cypher keyed with a symmetric key unique to each bitstream or session. The block symmetric keys are stored within the bitstream's block header subatoms. The bitstream symmetric key that is used to encrypt each block symmetric key is not stored within the bitstream. Rather a key identifier is stored within the bitstream header that enables the client to request the bitstream symmetric key from a key management system (KMS). Upon client authentication, the bitstream symmetric key is transmitted to the client using a secure channel encrypted using an asymmetric key (e.g. TLS [i.1]).

This approach limits the amount of communication between the client and KMS (only one key needs to be communicated). Once the bitstream key is obtained from the KMS, all encrypted fields within the bitstream are accessible.

The bitstream key does not necessarily have to be unique to each bitstream. Rather, this key can be shared across multiple bitstreams. For example, assume that a streaming session consists of segmented content where each segment is packaged within its own bitstream. Rather than having a unique key for each bitstream/segment, a session key can be used that is shared across all bitstreams. In this case, the session key is used to encrypt each of the block keys.

Figure 17 shows how the various keys are encrypted and stored within the bitstream and KMS.

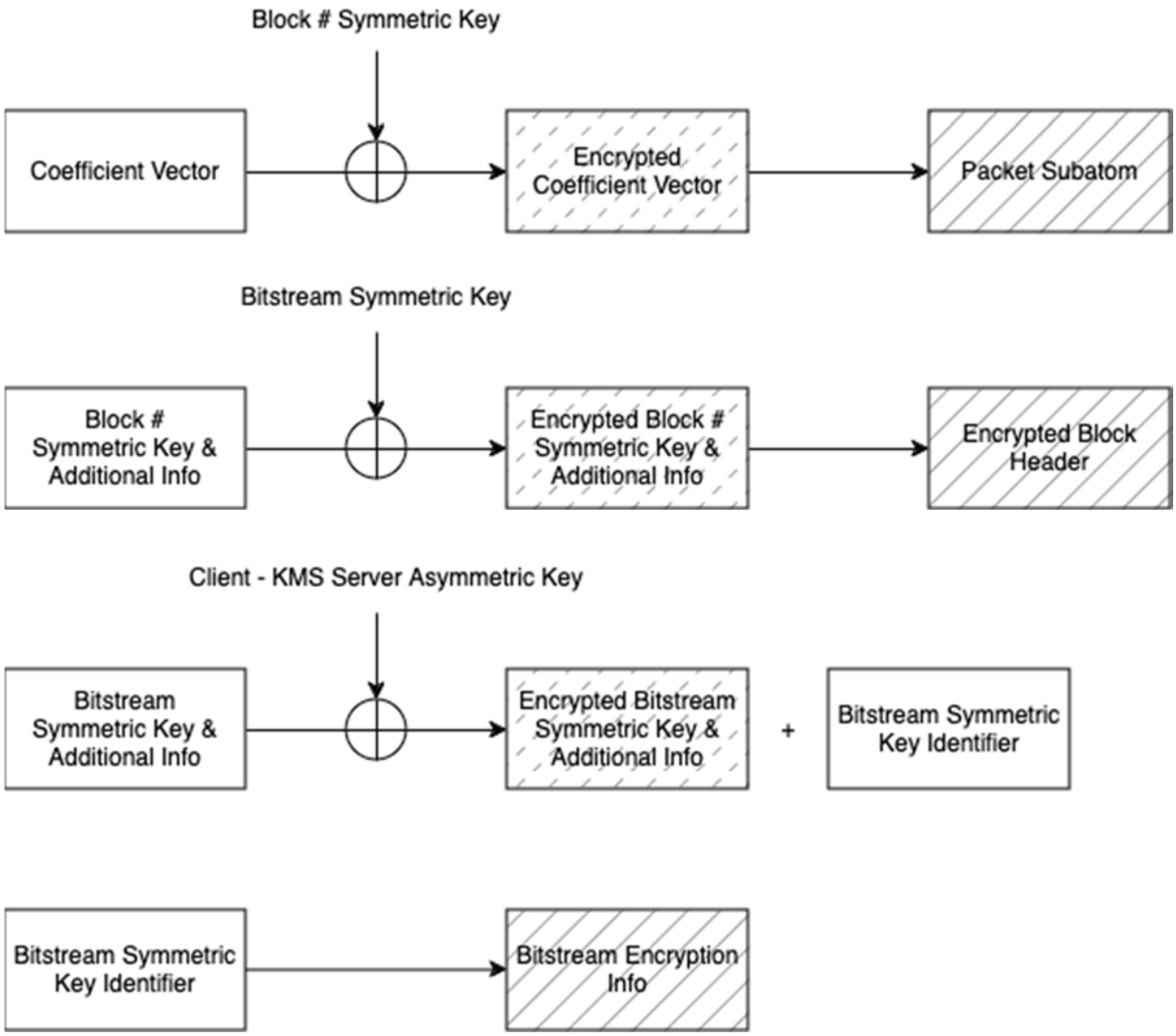

Figure 17: Bitstream, block, key management

Figure 18 shows the client process for decrypting each coefficient vector.

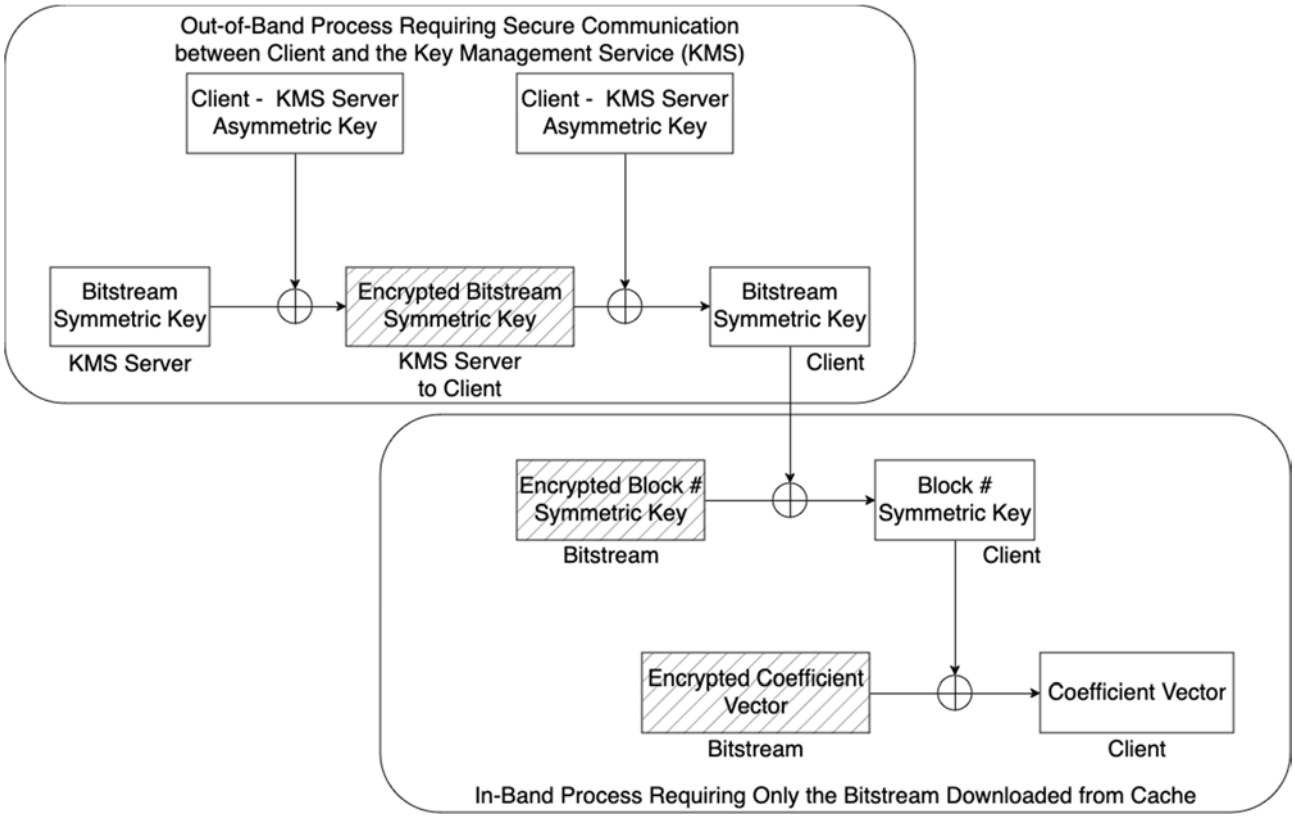

Figure 18: Client process for decryption

Figure 19 demonstrates one possible method of downloading a bitstream.

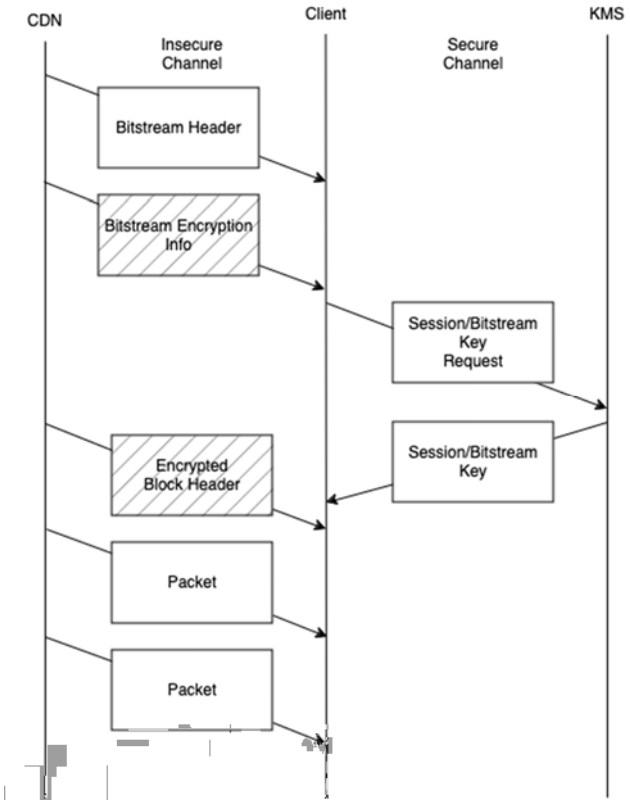

Figure 19: Bitstream and key download process

Annex A (normative): xCD-1

A.0 Introduction

This annex shows the functionality of code type 0, xCD-1.

A.1 Encoding

Segments of source coded audio-visual data are partitioned into one or more blocks. Each block is encoded into multiple equal-sized xCD-1 coded symbols. The symbols are carried in one or more correlated CMMF bitstreams. Segment boundaries, number of blocks, symbols, and bitstreams may be chosen to optimize operations including processing, caching, transmission paths, or to increase redundancy as required for the specific application.

Each CMMF bitstream with code_type xCD-1 shall contain at least one `bitstream_header()` subatom and at least one `block_header()` subatom.

Recommendations for `subatom()` structures (see clause 6.1.2), when multiple bitstreams are utilized:

- The `b_bitstream_id_present` field should be set.
- The `bitstream_id` field should be set to uniquely identify each bitstream, e.g. an ordinal number of the bitstream starting at `0`.
- It is recommended that the `bitstream_id` field be added to the bitstream header, and block header subatoms at a minimum.

Recommendations for `sync()` structures (see clause 6.1.3):

- The `b_content_encode_uuid` field should be set.
- The `content_encode_uuid` field should be set so that all correlated bitstreams carry a matching value.

Constraints for the `bitstream_header()` structure (see clause 6.1.4):

- The `code_type` field shall be set to `0` (xCD-1).
- The `b_rfc5052` field shall be cleared.
- The `content_source_size` field shall be set to the size of the source segment.
- The `block_count_minus1` field shall be set to the number of blocks, minus one.

Constraints for the `block_header()` structure (see clause 6.1.5):

- The `block_size` field shall be set to the size of the current block.
- The `block_num_symbols` field shall be set to the number of symbols in the current block.
- The `block_symbol_size` field shall be set to the size of each symbol in the current block (all symbols of a block are of equal size).
- The `b_block_max_symbol_index_present` field shall be cleared.
- Bit 1 of the `block_mask` field shall be cleared, indicating that the `block_field_size_exp_val` parameter shall have a value of `1`.

The symbol size is calculated according to the following formula:

```
block_symbol_size = ceil(block_size / block_num_symbols)
```

Padding data of zero or more bytes is added to the end of the last symbol, with the size calculated according to the following formula:

```
block_num_symbols × block_symbol_size - block_size
```

Figure A.1 shows an example with a segment size of 500 kBytes in one block, encoded into six symbols of 83 334 bytes, and 4 bytes of padding.

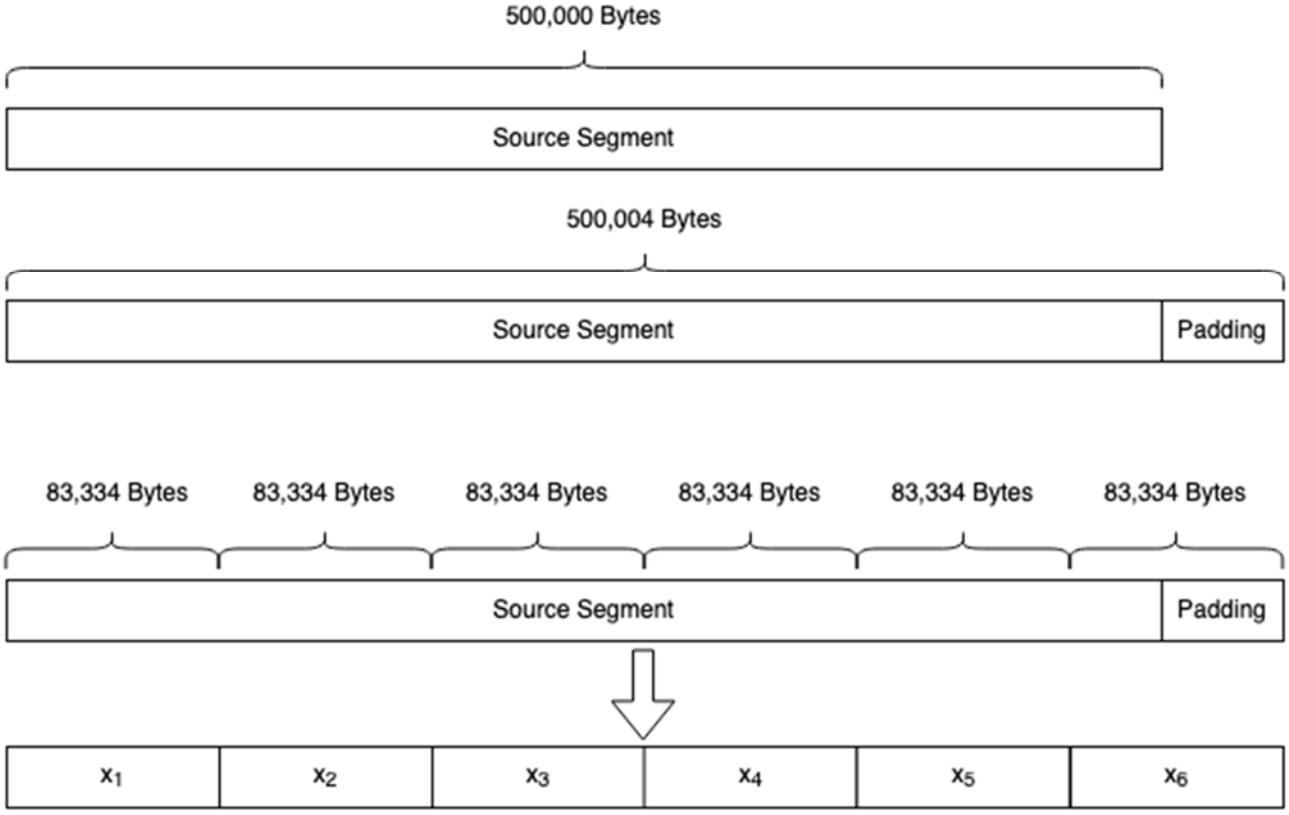

Figure A.1: xCD-1 segment to symbols

The symbols can be mathematically represented as a vector X of length `block_num_symbols`.

A coefficient matrix G with `block_num_symbols` columns and `block_num_symbols` × *number of CMMF bitstreams* rows is generated. Each of the coefficients within the matrix is contained in the Galois Field GF{2} (i.e. each coefficient is either a 0 or a 1). Multiple methods exist for determining these coefficients. The coefficients can be randomly or deterministically generated.

Figure A.2 illustrates a case with six symbols and two CMMF bitstreams.

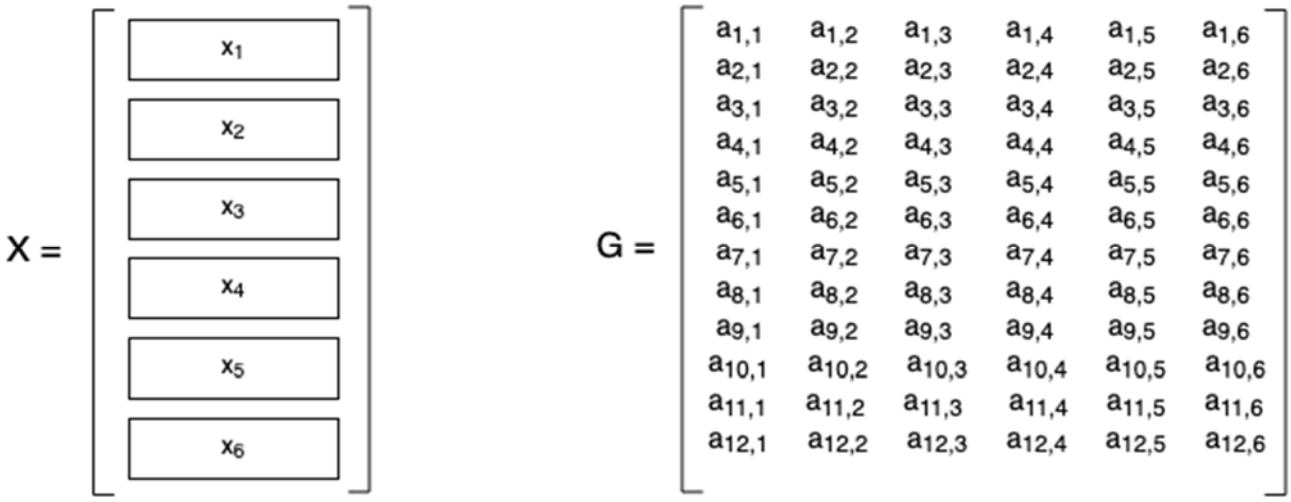

Figure A.2: xCD-1 symbol vector and coefficient matrix

The vector of symbols X is then multiplied with coefficient matrix G to create a vector of encoded symbols Y. These encoded symbols along with their respective row in matrix G are then packaged within the CMMF bitstreams.

Figure A.3 shows the process for the above example with six symbols and two CMMF bitstreams.

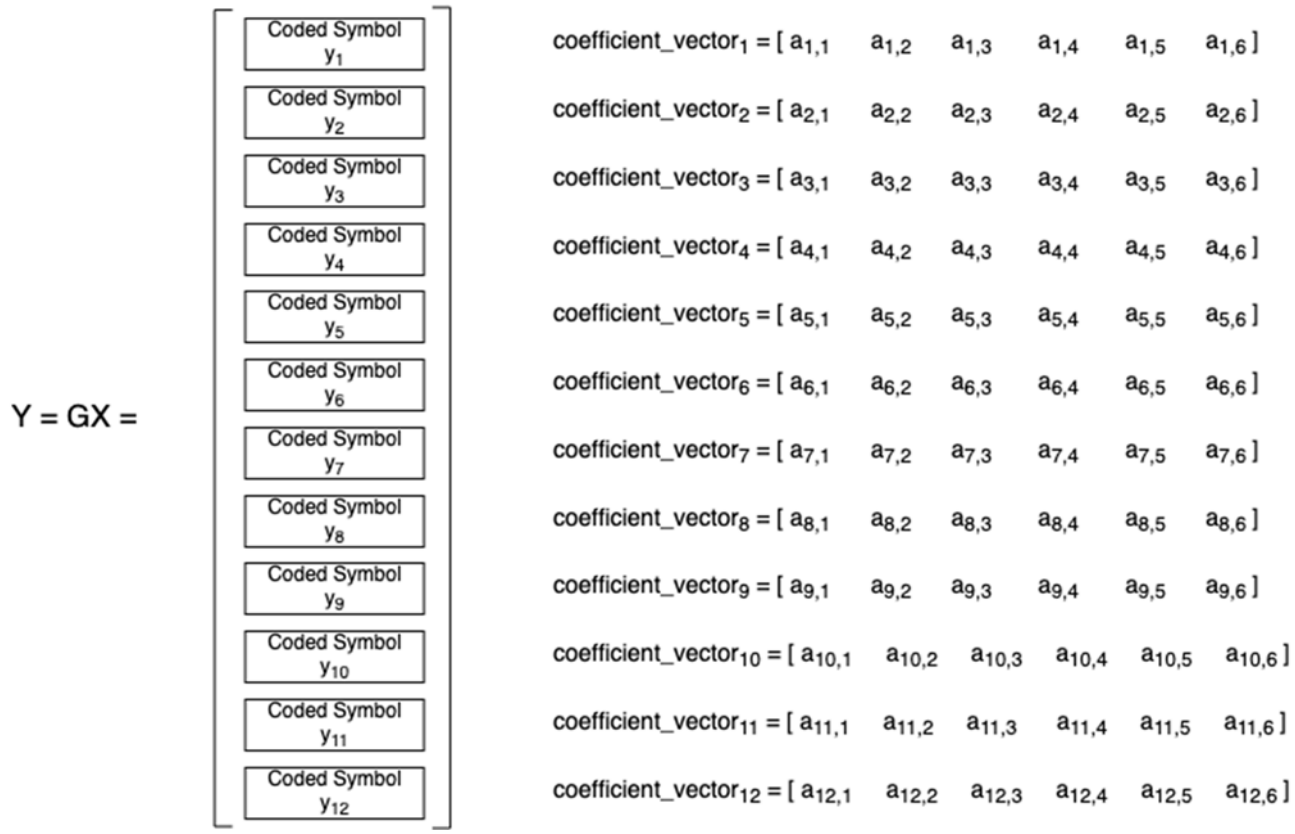

Figure A.3: xCD-1 coefficient matrix

In the simplest case, each CMMF bitstream carries `block_header()` subatoms with the `b_sufficient_symbols_present` fields set, followed by `block_num_symbols` subatoms of the `packet()` type, with the `packet_block_index` field indicating the block number, and the `coded_symbol` field holding the data of the coded symbol (one element of Y). More complex cases with subsets of symbols in each block or chunked subatoms are also possible.

Constraints for the `packet_header()` structure (see clause 6.1.7), e.g. contained in the `packet()` subatom:

- The `b_systematic_symbol` field shall be cleared, indicating coded symbols.
- Bit 4 of the `packet_mask` field shall be set, indicating the presence of the `coefficient_vector()` structure.

Constraints for the `coefficient_vector()` structure (see clause 6.1.15):

- Bit 2 of the `packet_mask` field shall be cleared, indicating that the `window_size` parameter shall have a value of `block_num_symbols`.
- The `block_field_size_exp_val` parameter shall have a value of `1`, indicating Galois Field GF{2} coefficients.
- The structure contains the coefficient vector for the respective row of G, matching the coded symbol.

Each coded symbol shall be carried exactly once. It may be carried, in its entirety, in either bitstream. Each bitstream shall carry exactly `block_num_symbols` coded symbols of a block.

A.2 Decoding

When receiving CMMF bitstreams with xCD-1 (i.e. `code_type` field with a value of `0`), the decoder assures matching `content_encode_uuid` (if present), `content_source_size`, `content_source_type`, and `block_count_minus1` field values across all bitstreams, and matching `block_size`, `block_num_symbols`, and `block_symbol_size` field values for each block.

The decoder then initializes an empty coded symbol vector Y' with an element size of `block_symbol_size`, and an empty coefficient matrix G' with `block_num_symbols` columns. The length of Y' as well as the row count of G' is at least `block_num_symbols`, and no more than `block_num_symbols` × *number of CMMF bitstreams*.

For each received `packet_header()` structure and completed coded symbol, the corresponding coded symbol is appended to the coded symbol vector Y' and the corresponding coefficient vector is appended (as a row) to the coefficient matrix G'. This process is repeated until the decoder confirms that the coefficient matrix G' has a rank equal to `block_num_symbols`.

Figure A.4 shows the result of this process for an example with six symbols and two bitstreams, where a sufficient number of coded symbols was received in the order of 1, 7, 8, 2, 9, 10.

Figure A.4: xCD-1 coded symbol vector and decoded coefficient matrix

At that point, the decoder inverts G' and multiplies it with the coded symbol vector Y' to determine the source symbol vector X'. Figure A.5 shows an example with six symbols.

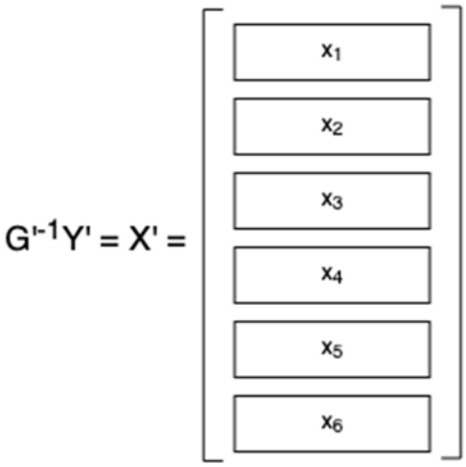

Figure A.5: xCD-1 symbol reconstruction

The decoder then appends each source symbol in X' together, removes any padding (if present, e.g. to create equal-sized source symbols), and finally delivers the decoded segment of source data.

Figure A.6 shows an example with a segment size of 500 kBytes in one block, decoded from six symbols, with 4 bytes of padding at the end of the final symbol.

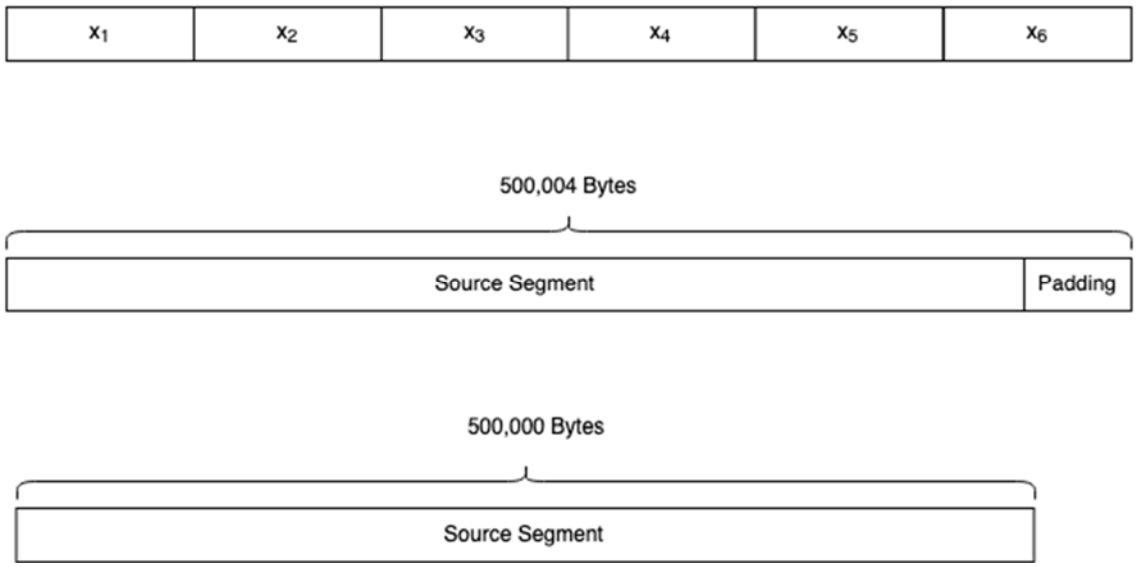

Figure A.6: xCD-1 data extraction

Annex B (informative): Media service architecture Examples

B.0 Introduction

The Coded Multisource Media Format (CMMF) provides an extensible method for the management and interchange of source coded audio-visual media and related metadata in one or more unique representations resulting from the use of linear, network or channel coding methods (see note). The CMMF bitstream structure supports multiple linear, network or channel code types and related information for use in delivering audio-visual media over multipath and/or multi-access networks in settings targeted at improving cloud/network-powered QoS and rendered QoE. This annex provides various media service architecture examples showing how CMMF can be employed within existing media service architectures to enable multisource delivery.

NOTE: Where, linear, network or channel coding methods are directly applied to source coded (or source coded and packaged) audio-visual media.

B.1 MPEG-DASH HTTP adaptive streaming service example

A typical MPEG-DASH HTTP adaptive streaming system is setup similarly to the non-shaded blocks shown in Figure B.1. Source media (e.g. audio/video elementary streams) are segmented and encoded into multiple representations, each with a different quality and bit rate. These segments are packaged together using MPEG-DASH and stored on an origin server located within the network. One or more CDNs are setup to distribute and deliver this content to an OTT service providers' customer base. These CDNs obtain every requested MPEG-DASH segment from an origin server, caches these segments at the networks' edge, and delivers these segments to clients.

Enabling multisource using CMMF within this existing delivery architecture can be accomplished through the addition of a CMMF Bitstream Generator/Source before segments are delivered to the CDNs and a CMMF Receiver on each client as illustrated by the shaded boxes in Figure B.1.

While Figure B.1 shows the CMMF Bitstream Generator/Source between the Origin Server and the CDNs (i.e. CMMF bitstreams are created on demand), the CMMF Bitstream Generator/Source can just as easily be located between the MPEG-DASH Packager and Origin Server. In the former case, the original MPEG-DASH segments are stored on the Origin Server and CMMF representations of those segments are cached on each CDN. In the latter case, the CMMF Bitstream Generator/Source creates multiple CMMF representations of each segment produced by the MPEG-DASH Packager and stores them on the Origin Server for later retrieval by a CDN.

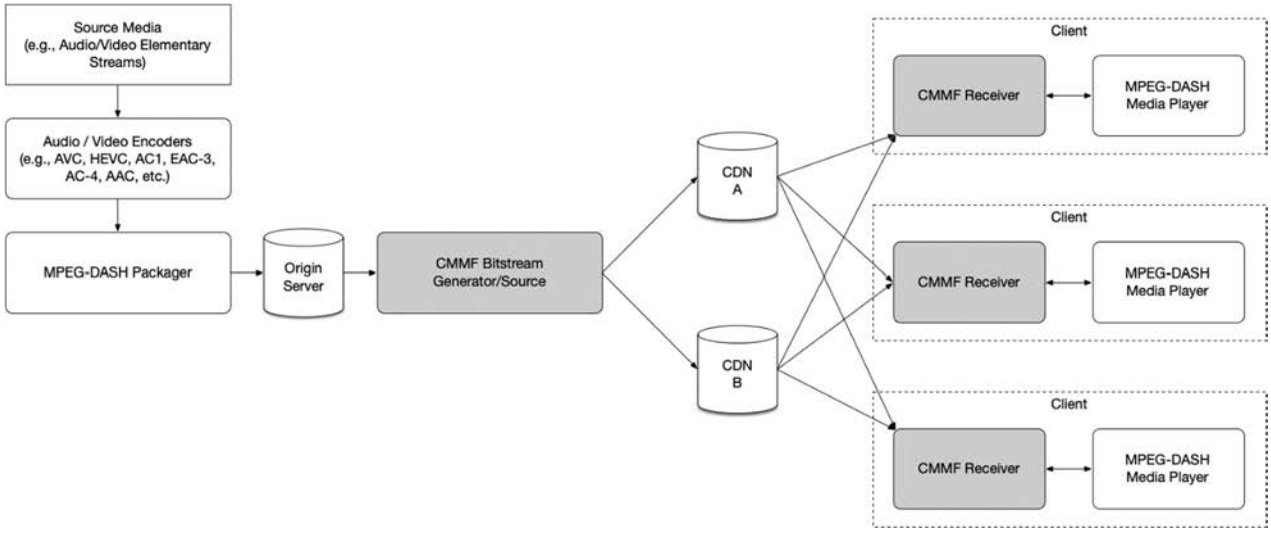

Figure B.1: MPEG-DASH with CMMF Delivery System Example

The reference architecture for this example is shown in Figure B.2. In this architecture, the Application Provider is responsible for segmenting, encoding, and packaging the media. Each MPEG-DASH segment is equivalent to a single Source Transport Object; and the Media Information consists of the media's corresponding MPEG-DASH master manifest. This master manifest (see Figure B.3) contains relative URLs to the files/segments (shown in a dotted outline) that make up the adaptation sets. This information is transferred via the C1 interface from the Application Provider to the Media Player (using a method preferred by the Application Provider) and is used by the Media Player to determine which segments are to be downloaded (via the CMMF Receiver) and played.

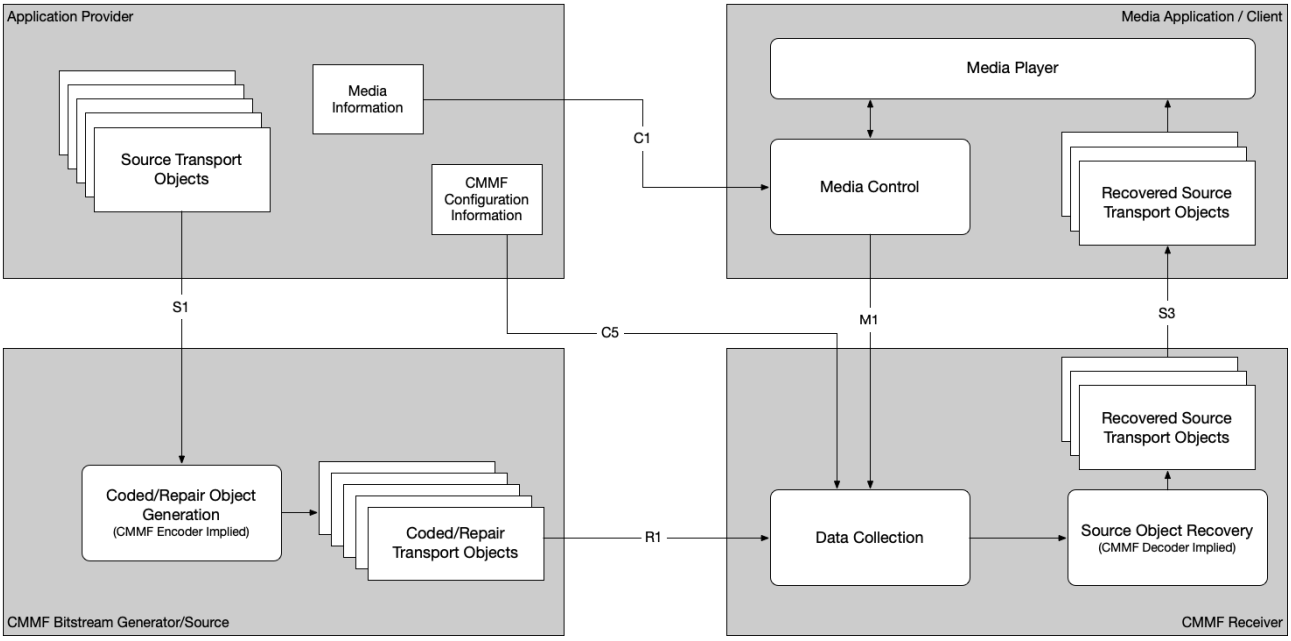

Figure B.2: CMMF reference architecture in relation to MPEG-DASH HTTP adaptive etreaming example

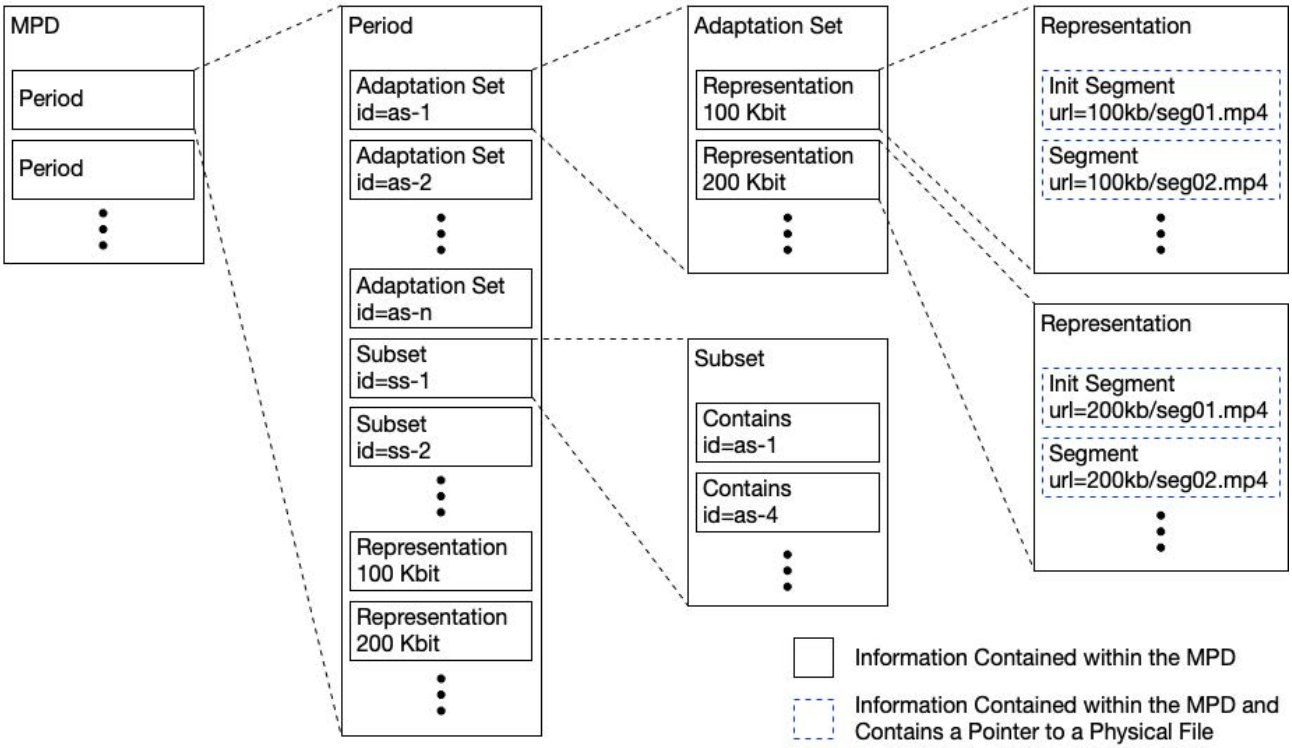

Figure B.3: Example MPEG-DASH master manifest

The Application Provider for this example is also responsible for determining and setting up the delivery system (i.e. CDNs) to distribute CMMF encoded media. Information about this setup is captured within the CMMF Configuration Information as a list of host URLs to each of the CDNs. It is important to note that both the MPEG-DASH master manifest and the CMMF Configuration Information is required to download the media. It is assumed that the Application Provider is utilizing two CDNs and each segment listed in the MPEG-DASH master manifest is encoded and packaged into two unique CMMF bitstreams, one intended for the first CDN and the other for the second CDN according to the list of host URLs provided in the CMMF Configuration Information. An example of this is shown in Figure B.4. Furthermore, example bitstream constructions can be found in annex C.

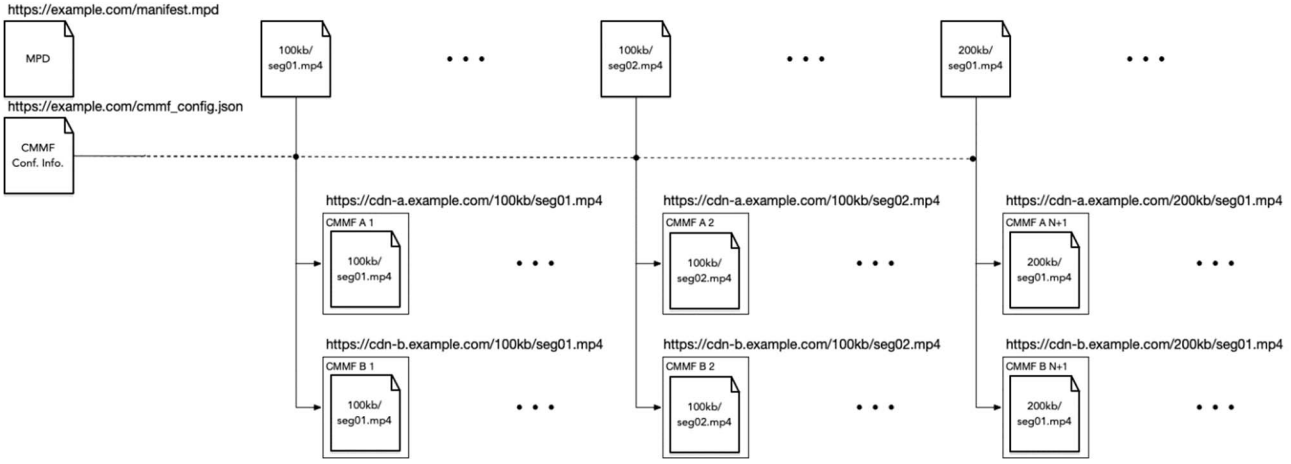

Figure B.4: CMMF bitstreams generated to deliver the MPEG-DASH packaged content

The process for streaming this content is shown in Figure B.5.

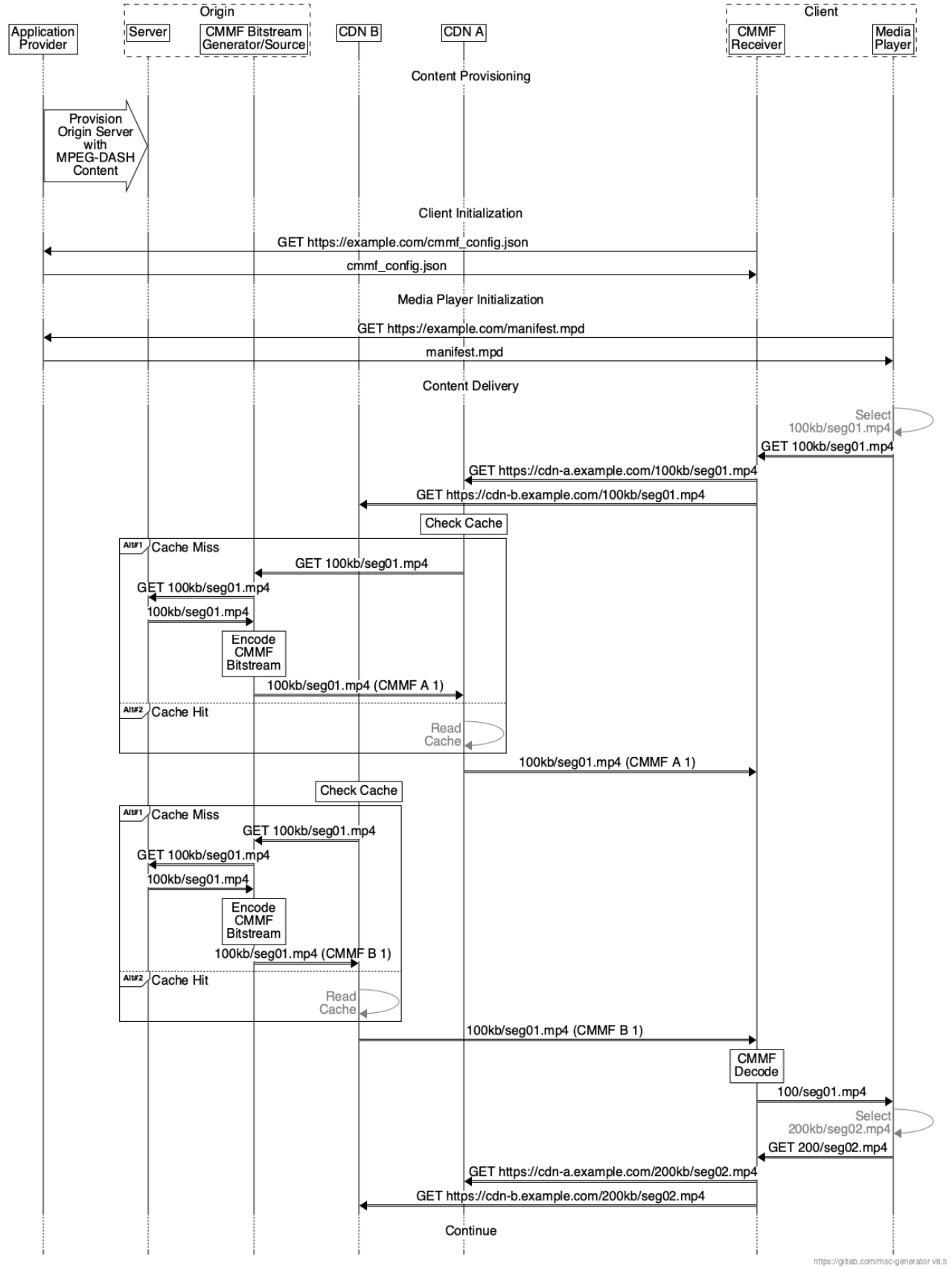

Figure B.5: CMMF request and content delivery example for MPEG-DASH

An Application Provider provisions an origin server with the original source segments shown in Figure B.1. Upon client initialization, the CMMF Receiver requests the CMMF Configuration Information (in this case it is stored as a JSON file) from the Application Provider. This information communicates a list of host URLs specifying the locations for which the client can retrieve CMMF encoded content for every segment listed within the master manifest file. Upon media player/delivery session initialization, the media player retrieves and parses the master manifest file from the Application Provider and chooses the adaptation set(s) it wishes to stream. Once selected, the media player requests the appropriate segment from the CMMF Receiver using the relative URL communicated in the master manifest. The CMMF Receiver joins the relative URL with each of the host URLs from the CMMF Configuration Information file and requests two distinct CMMF bitstreams of that segment from the two available CDNs. Assuming the appropriate CMMF bitstream is cached, the CDN begins delivery. Otherwise, the CDN requests the segment from the CMMF Bitstream Generator/Source. At which point, the original segment is pulled from storage, encoded, and delivered to the CDN and to the client. The CMMF Receiver downloads the two CMMF bitstreams of that segment until it is capable of decoding (see annex A for an example). It then cancels any outstanding data that has yet to be delivered, decodes the segment, and delivers it to the media player. The media player selects the next segment to be downloaded and the process repeats.

In the case of a third CDN being introduced, a new CMMF bitstream can be generated and cached without replacing or modifying the existing CMMF bitstreams already cached in the initial two CDNs. All that is required is an update to the host URL list managed by the Application Provider.

Furthermore, the above example implies that CMMF bitstreams are retrieved from each CDN using HTTP. As noted earlier, CMMF is agnostic to the underlying transport protocol. Other transport protocols (e.g. webRTC, FLUTE, ROUTE, etc.) may also be used if appropriate.

Annex C (informative): Example bitstreams

C.0 Introduction

The CMMF bitstream is designed to be applicable in multiple use cases. This annex contains example CMMF bitstreams for different applications.

C.1 Multisource Video-on-Demand

C.1.0 Multisource Video-on-Demand example using code_type xCD-1

This example shows one method of implementing the bitstream to support multisource Video-on-Demand (VoD) HTTP Adaptive Streaming (HAS). In a generalized scenario, a client requests and receives, via HTTP, two CMMF bitstreams from two separate CDNs as shown in Figure C.1. The client then merges these bitstreams together allowing it to decode the bitstreams and extract the audio, video, and/or metadata. In this example, communication between the client and the CDNs is reliable and the bitstream served by the CDN is cached.

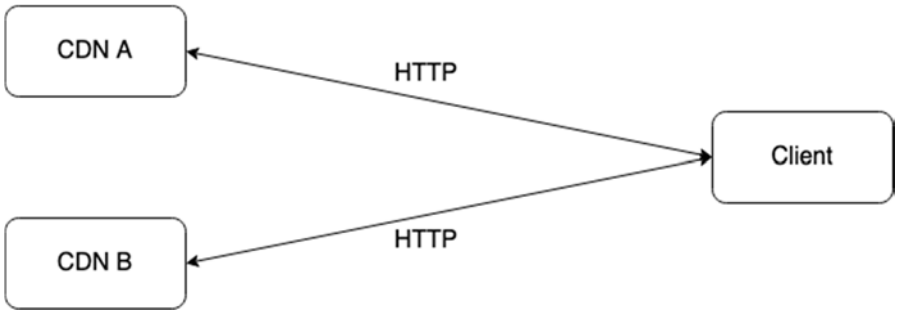

Figure C.1: Multi-source CDN-client communication

C.1.1 Bitstream construction

Multiple, correlated bitstreams can be constructed using the xCD-1 code type to enable efficient delivery of the underlying content. For this approach to work, each bitstream needs to utilize the same code type and encoder settings. Subatoms carrying this encoder setting information are transmitted in each bitstream. Each bitstream may contain different packet subatoms depending on the system's objectives. Figure C.2 shows two bitstreams created using the same data, and same encoder settings, and as a result has similar subatom arrangement.

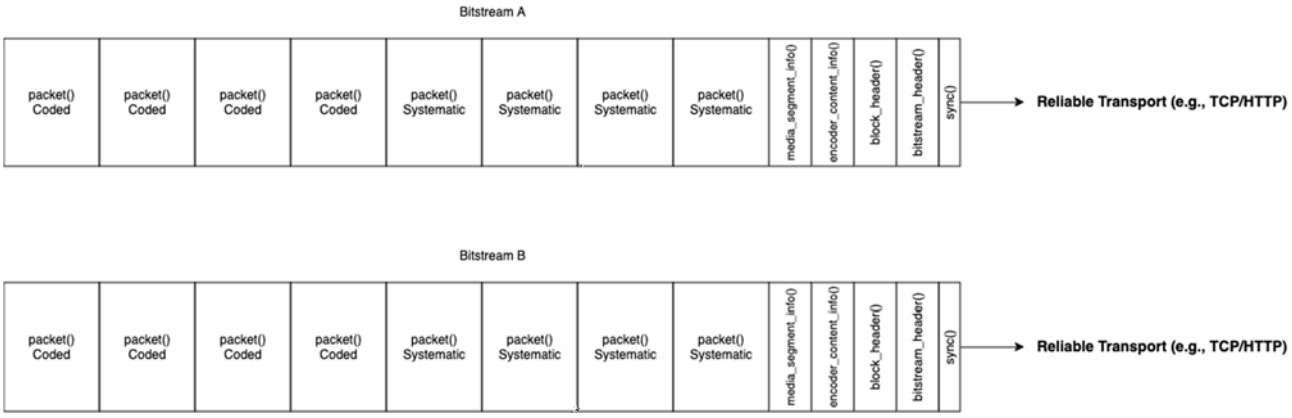

Figure C.2: CMMF multisource example bitstream arrangement

In this example, the content is partitioned into eight packets. Four systematic packets are transmitted in bitstream A, and four systematic packets are transmitted in bitstream B. Four additional coded packet subatoms are included within each bitstream to ensure that the source content can be recovered if a failure to obtain one of the bitstreams occurs. Subatoms required to initialize playback and/or decoders on client devices are sent within each bitstream.

The following clauses provide examples of the `sync()` structure and each subatom's construction for one of the bitstreams shown in Figure C.2 above. Subatoms required to ensure recovery of the original symbols are shown. Examples of optional subatoms, such as the encoder content information and media segment information subatoms, are also provided.

C.1.2 Sync construction

The CMMF bitstream starts with a data structure for synchronization, shown in Table C.1.

Table C.1: sync() structure construction

Syntax	Encoding	Example Value	Notes
`sync()`			
`{`			
`syncword`	v(64)	0x89780D430044AC31	Predefined value.
`version`	v(4)	0x0	Predefined value.
`b_content_encode_uuid`	b(1)	1b (TRUE)	Content encode UUID is used to add another method to ensure that data from correlated bitstreams from multiple sources are matched together.
`reserved`	v(3)	000b	
`if (b_content_encode_uuid){`			
`content_encode_uuid`	v(128)	0x294d34081445441db44f3dd8c1695cf7	Randomly generated UUID
`}`			
`}`			

C.1.3 Subatom construction

C.1.3.1 Bitstream header subatom construction

A subatom containing the Bitstream header structure is shown in Table C.2.

Table C.2: Bitstream header subatom construction

Syntax	Encoding	Example Value	Notes
subatom() { /* subatom header start */			
subatom_id	u(4)	2	SUBATOM_ID_BITSTREAM_HEADER
if (subatom_id == 0xF){			
subatom_id_ext	u(8)	NULL	
subatom_id = subatom_id + subatom_id_ext			
}			
b_bitstream_id_present	b(1)	1b (TRUE)	While optional, the bitstream_id is defined for bitstream identification purposes.
reserved	v(1)	0b	
sas_bits = num_bits_code()			Other instances of num_bits_code() are not expanded for brevity.
{			
bits_code	u(2)	00b	
num_bits = (bits_code + 1) × 8			
return num_bits			The subatom data size is indicated using 8 bits.
}			
if (b_bitstream_id_present){			
bitstream_id	v(16)	0	bitstream_id was chosen to be 0 in this example. The bitstream_id field can be unique to each source to help differentiate between them if necessary.
}			
subatom_size	u(sas_bits)	11	Number of bytes contained between /* subatom data start */ and /* subatom data end */.
/* subatom header end */			
/* subatom data start */ bitstream_header() {			
content_source_size	u(64)	2 000 000	Content source size is 2 MB. This value has to be consistent between bitstreams from each source.
content_source_type	v(3)	001b	Source of the content is the original data file. This value has to be consistent between bitstreams from each source.
reserved	v(1)	0b	
b_content_source_split	b(1)	0b (FALSE)	Bitstream represents the entire file.

Syntax	Encoding	Example Value	Notes
`if (b_content_source_split){`			
content_source_split_start	u(64)	NULL	
content_source_split_end	u(64)	NULL	
`}`			
code_type	u(4)	0000b	xCD-1. This value has to be consistent between bitstreams from each source.
`if (code_type == 0xF){`			
code_type_ext	u(8)	NULL	
`code_type = code_type + code_type_ext`			
`}`			
b_rfc5052	b(1)	0b (FALSE)	This CMMF bitstream is not being used as a content delivery protocol as per IETF RFC 5052 [15].
`if (b_rfc5052){`			
`rfc5052_information()`		NULL	
`}`			
`block_count_minus1 = block_index_or_count_value()`			Other instances of `block_index_or_count_value()` are not expanded for brevity.
`{`			
block_index_or_count	u(8)	0	There is only one block in this bitstream. This value has to be consistent between bitstreams from each source.
`if (block_index_or_count == 0xFF)`			
`{`			
block_index_or_count_ext	u(32)	NULL	
`}`			
`else`			
`{`			
`block_index_or_count_ext = 0`			
`}`			
`block_index_or_count_val = block_index_or_count + block_index_or_count_ext`			
`return block_index_or_count_val`			
`}`			
`block_count = block_count_minus1 + 1`			`block_count = 1`
b_content_block_separate_sources	b(1)	0b (FALSE)	The block is composed of a single source.
`if (b_content_block_separate_sources){`			
num_content_block_sources_minus1	u(8)	NULL	
`}`			
b_profile_information_present	b(1)	0b (FALSE)	Unknown whether this bitstream conforms to any profile.
`if (b_profile_information_present){`			
profile_type_size	u(4)	NULL	
profile_type	v(profile_type_size × 8)	NULL	
profile_description	v(32)	NULL	
`}`			
b_block_cc_encrypted	b(1)	0b (FALSE)	No blocks have encrypted coding coefficient information.
`if (b_block_cc_encrypted){`			
bitstream_encryption_key_id_size_exp	u(4)	NULL	
`bseki_bits = 2^bitstream_encryption_key_id_size_exp`			
bitstream_encryption_key_id	v(bseki_bits)	NULL	

Syntax	Encoding	Example Value	Notes
} } **padding** /* subatom data end */ }	pad	000b	3 bits of padding are needed for byte alignment.

C.1.3.2 Block header subatom construction

A subatom containing the Block header structure is shown in Table C.3.

Table C.3: Block header subatom construction

Syntax	Encoding	Example Value	Notes
subatom() { /* subatom header start */ **subatom_id**	u(4)	6	SUBATOM_ID_BLOCK_HEADER

if (subatom_id == 0xF) {
subatom_id_ext
subatom_id = subatom_id + subatom_id_ext
}
	u(8)	NULL	

b_bitstream_id_present	b(1)	1b (TRUE)	While optional, the bitstream_id is defined for bitstream identification purposes.
reserved	v(1)	0b	
sas_bits = num_bits_code()			The subatom data size is indicated using 8 bits. 2 bits used within expansion of num_bits_code().
if (b_bitstream_id_present) { **bitstream_id**	v(16)	0	bitstream_id was chosen to be 0 in this example. The bitstream_id field can be unique to each source to help differentiate between them if necessary.
}			
subatom_size /* subatom header end */	u(sas_bits)	11	Number of bytes contained between /* subatom data start */ and /* subatom data end */.
/* subatom data start */ block_header() { block_index = block_index_or_count_value()			Block index is 0. 8 bits used in the expansion of block_index_or_count_value().
block_size	u(32)	2 000 000	Block size is 2 MB. This value is identical between bitstreams from each source.
block_symbol_size	u(32)	250 000	Block symbol size = 2 MB / 8. This value is identical between bitstreams from each source.

Syntax	Encoding	Example Value	Notes
bns_bits = num_bits_code()			block_num_symbols is represented by 8 bits. 2 bits used within expansion of num_bits_code().
block_num_symbols	u(bns_bits)	8	Number of symbols in each block is 8. This value is identical between bitstreams from each source.
b_block_max_symbol_index_present	b(1)	0b (FALSE)	Symbol indexes not used in this bitstream.
if (b_block_max_symbol_index_present) {			
bmsi_bits = num_bits_code()		NULL	
block_max_symbol_index	u(bmsi_bits)	NULL	
}			
b_block_content_source_index_present	b(1)	0b (FALSE)	
if (b_block_content_source_index_present) {			
block_content_source_index	u(8)	NULL	
}			
b_block_composite_sources	b(1)	0b (FALSE)	
if (b_block_composite_sources) {			
block_num_composite_sources_minus1	u(8)	NULL	
blk_num_cmp_sources = block_num_composite_sources_minus1 + 1			
for (cmpsrc=0; cmpsrc < blk_num_cmp_sources; cmpsrc++) {			
bcss_bits = num_bits_code()		NULL	
block_composite_source_size	u(bcss_bits)	NULL	
}			
}			
b_addl_block_coding_info_present	b(1)	0b (FALSE)	
if (b_addl_block_coding_info_present) {			
addl_block_coding_mask	v(3)	NULL	
if (addl_block_coding_mask & 0x1) {			
b_addl_window_info_present	b(1)	NULL	
if (b_addl_window_info_present) {			
addl_window_info() = extension(4)			
}			
}			
if (addl_block_coding_mask & 0x2) {			
b_reserved_block_coding_params_present	b(1)	NULL	
if (b_reserved_block_coding_params_present) {			
reserved_block_coding_params() = extension(4)		NULL	
}			
}			
if (addl_block_coding_mask & 0x4) {			

Syntax	Encoding	Example Value	Notes
reserved_block_coding_info() = extension(4)		NULL	
}			
}			
block_mask	v(6)	000001b	See below.
if (block_mask & 0x1)			
{			
b_sufficient_symbols_present	b(1)	1b (TRUE)	Sufficient symbols are present in the bitstream to decode. This value may be FALSE if the number of packet subatoms located at a source is less than 8.
if (!b_sufficient_symbols_present)			
{			
bsp_bits = num_bits_code()		NULL	
block_symbols_present	u(bsp_bits)	NULL	
}			
}			
else			
{			
/* Symbols present count not available */			
}			
if (block_mask & 0x2)			
{			
block_field_size_exp	v(3)	NULL	
}			
else			
{			
/* GF{2^1}, block_field_size_exp_val=1 */			Block field size is GF{2}, `block_field_size_exp_val=1`.
}			
if (block_mask & 0x4)			
{			
/* Coding coeff info in packet header subatoms */			Coding coefficient information is contained within each packet subatom.
}			
if (block_mask & 0x8)			Coding coefficient information is not encrypted.
{			
block_cc_encryption_info_size_bits_code	v(1)	NULL	
byte_align	pad	NULL	
bcceis_bits = (block_cc_encryption_info_size_bits_code + 1) × 8			
block_cc_encryption_info_size	u(bcceis_bits)	NULL	
/* block encryption info start */			
reserved	v(1)	NULL	
block_cc_encryption_algorithm	v(3)	NULL	
block_cc_encryption_mode	v(4)	NULL	
block_cce_key_size_exp	u(4)	NULL	
bck_bits = 2^block_cce_key_size_exp			
block_cce_key	v(bck_bits)	NULL	
b_addl_block_cce_params_present	b(1)	NULL	
if (b_addl_block_cce_params_present)			
{			
addl_cce_parameters()		NULL	
}			
padding	pad	NULL	
/* block encryption info end */			
}			
if (block_mask & 0x10)			A block seed is not used.
{			

Syntax	Encoding	Example Value	Notes

prng_parameters()
}
if (block_mask & 0x20)
{
block_integrity() = fb_integrity()
}
}
padding
 	
/* subatom data end */
}
	pad	NULL
		
		NULL
		
		00b
			
			Block integrity is not present.
			
			2 bits of padding are needed for byte alignment.

C.1.3.3 Encoder content information subatom construction

A subatom containing the encoder content information structure is shown in Table C.4.

Table C.4: Encoder content info subatom construction

Syntax	Encoding	Example Value	Notes

subatom()
{
/* subatom header start */
subatom_id
 	
if (subatom_id == 0xF){
subatom_id_ext
subatom_id = subatom_id + subatom_id_ext
}
 	
b_bitstream_id_present
 	
reserved
 	
sas_bits = num_bits_code()
 	
if (b_bitstream_id_present){
bitstream_id
 	
}
 	
subatom_size
 	
/* subatom header end */
	
	u(4)
	
	u(8)
	
	b(1)
	
	v(1)
	
	v(16)
	
	u(sas_bits)
		
		3
		
		NULL
		
		1b (TRUE)
		
		0b
		
		0
		
		35
			
			SUBATOM_ID_ENCODER_CONTENT_INFO
			
			While optional, the bitstream_id is defined for bitstream identification purposes.
			
			The subatom data size is indicated using 8 bits. 2 bits used within expansion of num_bits_code().
			
			bitstream_id was chosen to be 0 in this example. The bitstream_id field can be unique to each source to help differentiate between them if necessary.
			
			Number of bytes contained between /* subatom data start */ and /* subatom data end */.

/* subatom data start */
encoder_content_info()
{
b_encoder_id_present
if (b_encoder_id_present)
{
	b(1)	1b (TRUE)	

Syntax	Encoding	Example Value	Notes
encoder_uuid	v(128)	0x 8a245264eb e144eba3bd 94db28b1af cb	Randomly generated UUID to represent the encoder that created this bitstream.
}			
b_content_id_present	b(1)	1b (TRUE)	Specifies Entertainment Identifier Registry (EIDR) to identify the content.
if (b_content_id_present)			
{			
content_id_type	v(4)	0x0	Specifies Entertainment Identifier Registry (EIDR) to identify the content.
content_id_size_minus1	u(8)	11	EIDR contains a 96-bit (12 byte) identifier. The value 11 is used because the field is encoded as minus 1.
content_id	v((content_id_size_minus1+1) × 8)	0x1478F85A E100B0685 B8FB1C8	Corresponds to an EIDR value of 10.5240/F85A-E100-B068-5B8F-B1C8-T.
}			
b_content_location_present	b(1)	0b (FALSE)	
if (b_content_location_present)			
{			
content_location_size_minus1	u(11)	NULL	
content_location	v((content_location_size_minus1+1)×8)	NULL	
}			
b_content_type_present	b(1)	0b (FALSE)	
if (b_content_type_present)			
{			
content_type_size	u(8)	NULL	
content_type	v(content_type_size×8)	NULL	
}			
b_content_header_present	b(1)	0b (FALSE)	
if (b_content_header_present)			
{			
content_header_size	u(7)	NULL	
content_header	v(content_header_size×8)	NULL	
}			
b_file_integrity_present	b(1)	0b	File integrity not present.
if (b_file_integrity_present)			
{			
file_integrity() = fb_integrity()		NULL	
}			
b_media_preso_dur_present	b(1)	1b (TRUE)	
if (b_media_preso_dur_present)			
{			
media_presentation_duration() = cmmf_time()			media_presentation_duration = 00:00:03:11.552.
{			
b_ddhhmmss_format	b(1)	1b (TRUE)	
if (b_ddhhmmss_format) {			
if (**b_dd_present**) (	b(1)	0b (FALSE)	
days	u(5)	NULL	
}			
if (**b_hh_present**) (	b(1)	0b (FALSE)	
hours	u(5)	NULL	
}			
if (**b_mm_present**) (	b(1)	1b (TRUE)	
minutes	u(6)	3	
}			

Syntax	Encoding	Example Value	Notes

`if (b_ss_present) (`
`  seconds`
`}`
`} else {`
`  is_bits = num_bits_code()`
`  int_seconds`
`}`
`b_fract_seconds_present`
`if (b_fract_seconds_present)`
`{`
`  fract_seconds_bits_code`
`  fs_bits = 10 × 2^fract_seconds_bits_code`
`  fract_seconds`
`}`
`}`
`}`
`reserved`
`}`
`padding`
`/* subatom data end */`
`}`
	b(1)
	u(6)
	
	u(is_bits)
	b(1)
	
	v(2)
	
	u(fs_bits)
	
	v(4)
	
	pad
		1b (TRUE)
		11
		
		NULL
		1b (TRUE)
		
		00b
		
		552
		
		0000b
		
		000b
			
			The `fract_seconds` field is indicated using 10 bits.
			
			3 bits of padding needed to ensure byte alignment.

C.1.3.4 Media segment information subatom construction

A subatom containing the media segment information structure is shown in Table C.5.

Table C.5: Media segment info subatom construction

Syntax	Encoding	Example Value	Notes

`subatom()`
`{`
`/* subatom header start */`
`subatom_id`
 	
`if (subatom_id == 0xF){`
`subatom_id_ext`
`subatom_id = subatom_id + subatom_id_ext`
`}`
 	
`b_bitstream_id_present`
 	
`reserved`
 	
`sas_bits = num_bits_code()`
 	
`if (b_bitstream_id_present){`
`bitstream_id`
 	
`}`
	
	u(4)
	
	u(8)
	
	b(1)
	
	v(1)
	
	v(16)
		
		4
		
		NULL
		
		1b (TRUE)
		
		0b
		
		0
			
			SUBATOM_ID_MEDIA_SEGMENT_INFO
			
			While optional, the `bitstream_id` is defined for bitstream identification purposes.
			
			The subatom data size is indicated using 8 bits. 2 bits are used in the expansion of num_bits_code().
			
			`bitstream_id` was chosen to be 0 in this example. The `bitstream_id` field can be unique to each source to help differentiate between them if necessary.

Syntax	Encoding	Example Value	Notes
subatom_size	u(sas_bits)	49	Number of bytes contained between `/* subatom data start */` and `/* subatom data end */`.
/* subatom header end */			
/* subatom data start */			
media_segment_info()			
{			
media_segment_block_index = block_index_or_count_value()		0	This media segment information pertains to the content in block 0. 8 bits were used in the expansion of `block_index_or_count_value()`.
reserved	v(4)	0000b	
b_composite_source_index_present	b(1)	0b	
if (b_composite_source_index_present)			
{			
media_segment_composite_source_index	u(8)	NULL	
}			
media_segment_index	u(2)	0	There is only one media segment present, its index is 0.
if (media_segment_index == 0x3)			
{			
media_segment_index_ext	u(6)	NULL	
media_segment_index = media_segment_index + media_segment_index_ext			
}			
if (**b_asset_name_present**)	b(1)	1b (TRUE)	
{			
asset_name_size	u(8)	36	
for (i = 0; i < asset_name_size; i++)			
{			
asset_name[i]	v(36)	repousse_180p_100kb/Number03d001.m4s	
}			
}			
segment_tag_mask	v(4)	1100b	
if (segment_tag_mask & 0x1)			
{			
segment_duration() = cmmf_time()		NULL	
}			
if (segment_tag_mask & 0x2)			
{			
segment_start_time() = cmmf_time()		NULL	
}			
if (segment_tag_mask & 0x4)			
{			
segidx_bits = num_bits_code()			The `segment_index` field is indicated using 8 bits. 2 bits were used in the expansion of `num_bits_code()`.
segment_index	u(segidx_bits)	1	Index of this segment.
}			
if (segment_tag_mask & 0x8)			
{			

Syntax	Encoding	Example Value	Notes
segcnt_bits = num_bits_code()			The segment_count field is indicated using 8 bits. 2 bits were used in the expansion of num_bits_code().
segment_count	u(segcnt_bits)	79	Number of segments in this program.
}			
if (**b_media_mime_type_present**) {	b(1)	0b (FALSE)	
media_mime_type_size	u(6)	NULL	
for (i = 0; i < media_mime_type_size; i++) {			
media_mime_type[i]	v(8)	NULL	
}			
}			
if (**b_media_codec_present**) {	b(1)	0b (FALSE)	
media_codec_size	u(6)	NULL	
for (i = 0; i < media_codec_size; i++) {			
media_codec[i]	v(8)	NULL	
}			
}			
if (**b_bit_rate_present**) {	b(1)	0b (FALSE)	
bit_rate_bits_code	v(1)	NULL	
bps_bits = (bit_rate_bits_code + 3) × 8			
bit_rate	u(bps_bits)	NULL	
}			
if (**b_ms_content_type_present**) {	b(1)	1b (TRUE)	
ms_content_type	v(3)	0x0	The content type of video.
b_ms_content_type_info_present	b(1)	1b (TRUE)	
}			
else			
{			
/* b_ms_content_type_info_present = 0b */			
}			
if (b_ms_content_type_info_present) {			
switch (ms_content_type) {			
case 0x0:			
if (**b_aspect_ratio_present**) {	b(1)	0b (FALSE)	
sample_aspect_ratio	u(8)	NULL	
if (sample_aspect_ratio == 255) {			
sar_width	u(8)	NULL	
sar_height	u(8)	NULL	
}			
}			
b_dynamic_resolution_video	b(1)	0b (FALSE)	
if (**b_resolution_present**) {	b(1)	NULL	
resolution_width	u(16)	NULL	
resolution_height	u(16)	NULL	
}			
if (**b_frame_rate_present**) {	b(1)	1b (TRUE)	
frame_rate	v(5)	4	This video is at 30 frames per second.
}			
reserved	v(4)	0000b	

Syntax	Encoding	Example Value	Notes
if (**b_hdr_info_present**) {	b(1)	0b (FALSE)	
hdr_compatibility_mask	v(16)	NULL	
if (**b_addl_hdr_info_present**) {	b(1)	0b (FALSE)	
hdr_compat_mask_index	u(4)	NULL	
hdr_profile	u(8)	NULL	
hdr_level	u(8)	NULL	
hdr_compatibility_id	u(8)	NULL	
}			
}			
if (**b_addl_video_info_present**) {	b(1)	0b (FALSE)	
addl_video_info = extension(4)		NULL	
}			
break			
case 0x1:			
if (**b_sampling_freq_present**) {	b(1)	NULL	
b_sampling_freq_is_48k	b(1)	NULL	
if (!b_audio_fs_is_48k) {			
sampling_frequency	v(4)	NULL	
}			
}			
if (**b_audio_config_present**) {	b(1)	NULL	
audio_channel_config	v(24)	NULL	
}			
if (**b_audio_props_present**) {	b(1)	NULL	
reserved	v(1)	NULL	
b_virtualized_bin	b(1)	NULL	
reserved	v(2)	NULL	
b_object_audio	b(1)	NULL	
if (**b_complexity_index_present**) {	b(1)	NULL	
complexity_index	u(8)	NULL	
}			
}			
if (**b_addl_audio_info_present**) {	b(1)	NULL	
addl_audio_info = extension(4)		NULL	
}			
break			
default:			
if (**b_addl_ms_content_type_info_present**) {	b(1)	NULL	
addl_ms_content_type_info() = extension(6)		NULL	
}			
}			
}			
accessibility_mask	v(4)	0001b	
if (accessibility_mask & 0x1) {			
language_size	u(6)	2	
for (i = 0; i < language_size; i++) {			

Syntax	Encoding	Example Value	Notes

language[i]
}
}
if (accessibility_mask & 0x2) {
reserved
}
if (accessibility_mask & 0x4) {
reserved
}
if (accessibility_mask & 0x8) {
addl_accessibility_info() = extension(4)
}
}
padding
/* subatom data end */
}
	v(8)
	
	v(3)
	
	v(4)
	
	pad
		en
		
		NULL
		
		NULL
		
		NULL
		
		0000000b
			Language is 'en' for English as specified in ISO 639 [4].
			
			Seven bits of padding needed to ensure byte alignment.

C.1.3.5 Packet subatom construction - systematic symbol

A subatom containing the packet structure for a systematic packet is shown in Table C.6.

Table C.6: Packet subatom - systematic packet construction

Syntax	Encoding	Example Value	Notes

subatom()
{
/* subatom header start */
subatom_id
if (subatom_id == 0xF){
subatom_id_ext
subatom_id = subatom_id + subatom_id_ext
}
 	
b_bitstream_id_present
 	
reserved
 	
sas_bits = num_bits_code()
 	
if (b_bitstream_id_present){
bitstream_id
 	
}
 	
subatom_size
 	
/* subatom header end */
/* subatom data start */
packet()
	
	u(4)
	
	u(8)
	
	b(1)
	
	v(1)
	
	v(16)
	
	u(sas_bits)
		
		6
		
		NULL
		
		1b (TRUE)
		
		0b
		
		0
		
		250 003
			
			SUBATOM_ID_PACKET
			
			While optional, the bitstream_id is defined for bitstream identification purposes.
			
			The subatom data size is indicated using 24 bits.
			
			bitstream_id was chosen to be 0 in this example. The bitstream_id field can be unique to each source to help differentiate between them if necessary.
			
			Number of bytes contained between /* subatom data start */ and /* subatom_data_end */.

Syntax	Encoding	Example Value	Notes
{			
if (block_count > 1)			
{			
packet_block_index = block_index_or_count_value()		NULL	
}			
else			
{			
packet_block_index = 0			The packet is associated with block 0.
}			
packet_header()			
{			
b_systematic_symbol	b(1)	1b (TRUE)	This packet subatom contains a systematic symbol.
packet_mask	v(7)	0000001b	See below.
if (packet_mask & 0x1)			
{			
psi_bits = num_bits_code()			The packet_symbol_index field is indicated using 8 bits. 2 bits were used in the expansion of num_bits_code().
packet_symbol_index	u(psi_bits)	0	The systematic packet corresponds to original symbol index 0, i.e. the first source symbol.
}			
if (packet_mask & 0x2)			Coding coefficient information is not encrypted.
{			
if (b_systematic_symbol)			
{			
b_systematic_symbol_encrypted	b(1)	NULL	
} else {			
reserved	v(1)	NULL	
}			
b_addl_packet_cce_params_present	b(1)	NULL	
if (b_addl_packet_cce_params_present)			
{			
addl_cce_parameters()		NULL	
}			
}			
if (block_mask & 0x4) {			Window boundaries are not defined.
window_start_index	u(16)	NULL	
window_stop_index	u(16)	NULL	
window_size = (window_stop_index - window_start_index + 1) % 65 536			
} else {			
window_size = block_num_symbols			
}			
if (packet_mask & 0x8) {			PRNG seed is not defined for systematic packets.
prng_parameters()		NULL	
}			
if (packet_mask & 0x10) {			Coefficient vectors are not defined for systematic packets.
coefficient_vector(window_size, block_field_size_exp_val)		NULL	
}			
if (packet_mask & 0x20) {			Packet integrity not present.
packet_integrity()		NULL	

Syntax	Encoding	Example Value	Notes

}
if (packet_mask & 0x40){
packet_header_extension() = extension(5)
}
byte_align
}
coded_symbol
}
padding
/* subatom data end */
}
	pad
	v(css_bits)
	pad
		NULL
		000000b
		NULL
			Packet header extension is not present.
			6 bits are necessary for byte alignment.
			Coded symbol size = 250 000 bytes.
			Padding not necessary to ensure byte alignment.

C.1.3.6 Packet subatom construction - coded symbol

A subatom containing the packet structure for a coded packet is shown in Table C.7.

Table C.7: Packet subatom - coded packet construction

Syntax	Encoding	Example Value	Notes

subatom()
{
/* subatom header start */
subatom_id
if (subatom_id == 0xF){
subatom_id_ext
subatom_id = subatom_id + subatom_id_ext
}
b_bitstream_id_present
reserved
sas_bits = num_bits_code()
if (b_bitstream_id_present){
bitstream_id
}
subatom_size
/* subatom header end */
	u(4)
	u(8)
	b(1)
	v(1)
	v(16)
	u(sas_bits)
		6
		NULL
		1b (TRUE)
		0b
		0
		250 002
			SUBATOM_ID_PACKET
			While optional, the bitstream_id is defined for bitstream identification purposes.
			The subatom data size is indicated using 24 bits.
			bitstream_id was chosen to be 0 in this example. The bitstream_id field can be unique to each source to help differentiate between them if necessary.
			Number of bytes contained between /* subatom data start */ and /* subatom_data_end */.

/* subatom data start */
packet()
{
if (block_count > 1)
{
packet_block_index =
block_index_or_count_value()
}
else
		NULL	

Syntax	Encoding	Example Value	Notes
{			
packet_block_index = 0			The packet is associated with block 0.
}			
packet_header()			
{			
b_systematic_symbol	b(1)	0b (FALSE)	
packet_mask	v(7)	0010000b	This packet subatom contains a coded symbol.
if (packet_mask & 0x1)			The pscket_symbol_index is not defined for this packet subatom.
{			
psi_bits = num_bits_code()			
packet_symbol_index	u(psi_bits)	NULL	
}			
if (packet_mask & 0x2)			Coding coefficient information is not encrypted.
{			
if (b_systematic_symbol)			
{			
b_systematic_symbol_encrypted	b(1)	NULL	
} else {			
reserved	v(1)	NULL	
}			
b_addl_packet_cce_params_present	b(1)	NULL	
if (b_addl_packet_cce_params_present)			
{			
addl_cce_parameters()		NULL	
}			
}			
if (block_mask & 0x4) {			Window boundaries are not defined.
window_start_index	u(16)	NULL	
window_stop_index	u(16)	NULL	
window_size = (window_stop_index - window_start_index + 1) % 65 536			
} else {			
window_size = block_num_symbols			
}			
if (packet_mask & 0x8) {			PRNG seed is not defined.
prng_parameters()			
}			
if (packet_mask & 0x10) {			
coefficient_vector(window_size, block_field_size_exp_val)		10110101b	Coefficient vector containing 8 coefficients (one for each original source symbol) where each coefficient is 1 bit.
}			
if (packet_mask & 0x20) {			Packet integrity not present.
packet_integrity()			
}			
if (packet_mask & 0x40){			
packet_header_extension() = extension(5)			Packet header extension is not present.
}			
byte_align	pad	NULL	No bits are required for byte alignment.
}			
coded_symbol	v(css_bits)	coded symbol data	Coded symbol size = 250 000 bytes.
}			
padding	pad	NULL	Padding not necessary to ensure byte alignment.
/* subatom data end */			

Syntax	Encoding	Example Value	Notes
}			

C.2 Encrypted coding coefficient information example using CMMF

The CMMF bitstream supports encryption of coding coefficient information. Following the example described in clause 7.3.1, Figure C.3 shows field values in the bitstream header, block header, and packet subatoms. Fields within the bitstream (such as the `block_cce_key`) and other information (such as the `bitstream_key`) that are encrypted are highlighted within a hatched box.

Figure C.3 also shows the communication between the application and the key management system to obtain the `bitstream_key`. The daisy-chain process of using the `bitstream_key` to decrypt the block encryption parameters in the block header subatom, which are in turn used to decrypt the encrypted coefficient vector within the packet subatom, is also shown.

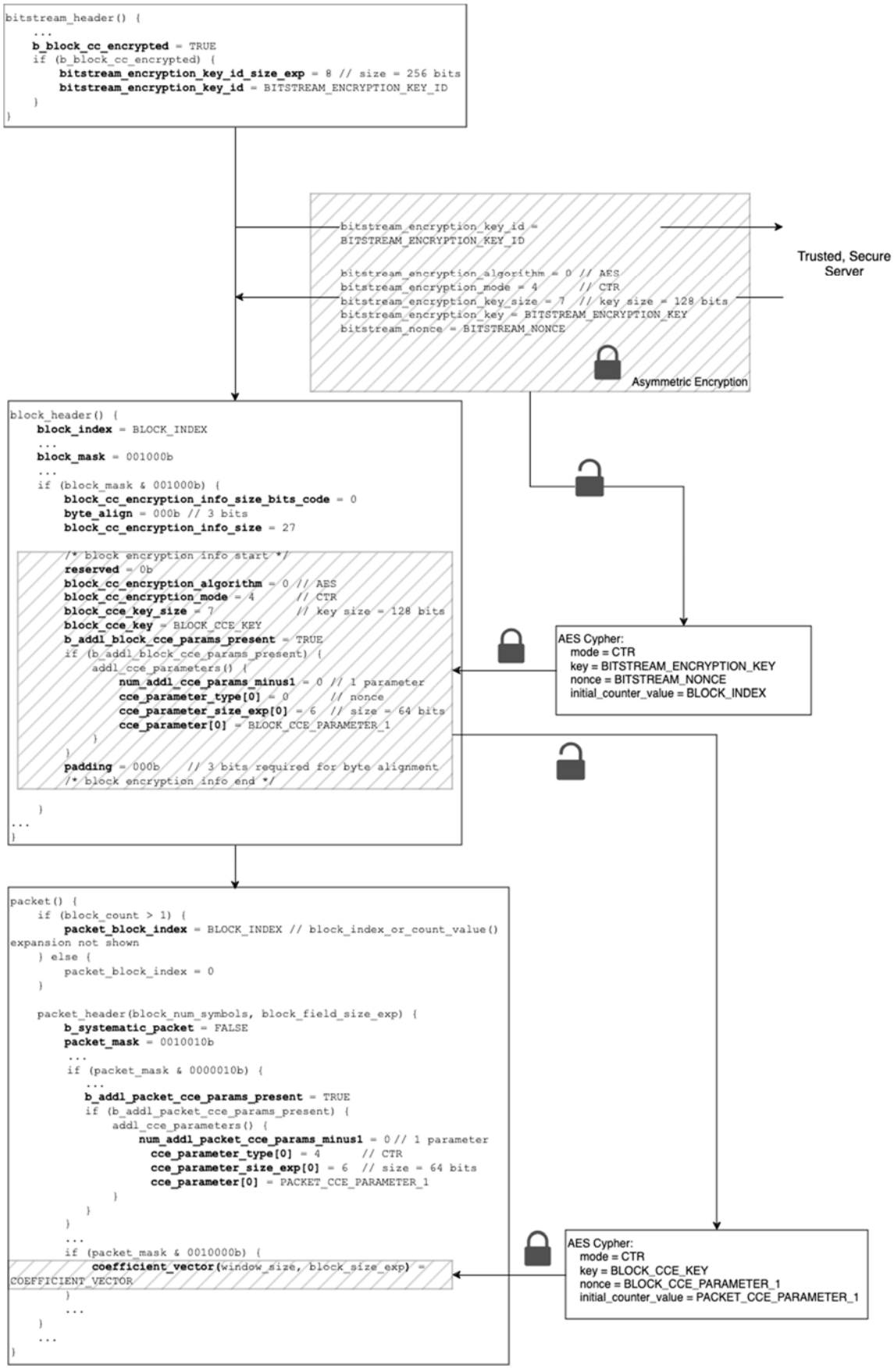

Figure C.3: Bitstream with example encrypted coding coefficient information

Annex D (normative): Content delivery protocol-based instantiations

D.1 CMMF content delivery protocol principles

D.1.1 Introduction

CMMF as introduced in the principles in clause 4, the bitstream syntax in clause 5, as well as the bitstream semantics in clause 6 can be, but not exclusively, considered as a Content Delivery Protocol (CDP) specification as defined in clause 8 of IETF RFC 5052 [15]. This clause provides a mapping of CMMF to the IETF RFC 5052 [15] principles and vice versa and apply when the `b_rfc5052` flag is set to TRUE.

D.1.2 FEC Building Block principles

CMMF provides means to generate coded/repair objects and coded/repair object transport sessions as introduced in clause 4.3.2. In one example instantiation, the CMMF repair framework is based on an existing Forward Error Correction (FEC) Building Block as defined in IETF RFC 5052 [15], where CMMF can be considered as a CDP specification as defined in clause 8 of IETF RFC 5052 [15].

In the definition of the protocol, terminology defined in clause 2 of IETF RFC 5052 [15], is used in this clause. Specifically, the terms for Object, Symbol, Source Symbol, Repair symbol, Encoding symbol, Encoder, Decoder, Receiver, Sender, and FEC Scheme are used in this clause.

IETF RFC 5052 [15] provides the definition and transport of three kinds of information from sender to receiver(s):

- encoding symbols (referred to as coded/repair symbols in the general context of CMMF) themselves;
- ancillary information associated with encoding symbols (or groups of such symbols), such as the group of symbols in a source or repair object; and
- ancillary information associated with the whole object being transferred.

In addition, for FEC the following information is defined and provided in IETF RFC 5052 [15]:

- FEC information associated with an object, referred to as *FEC Object Transmission Information (OTI)*. This includes for example the FEC Encoding ID that identifies the FEC scheme and is an integer assigned by IANA, the transfer length of the object or the encoding symbol length. For details refer to clause 6.2 of IETF RFC 5052 [15].
- FEC information associated with specific encoding symbols for an object, referred to as *FEC Payload ID*. This information indicates how the associated repair symbols were constructed from the object. The semantics and encoding format of the FEC Payload ID, including its size, is defined by the FEC Scheme. CDPs specify how the FEC Payload ID is carried in the respective framework. For details, refer to clause 6.3 of IETF RFC 5052 [15].

D.1.3 FEC Schemes and related information

IETF RFC 5052 [15] includes the concept of an *FEC Scheme*, this concept is different from the concept of an *FEC code* or *FEC algorithm*. An FEC scheme defines the ancillary information and procedures which, combined with an FEC code or algorithm specification, fully define how the FEC code can be used with CDPs. Requirements for FEC scheme specifications are defined in clause 7 of IETF RFC 5052 [15].

CMMF, when used as a CDP, only permits the use of fully specified FEC schemes. For each FEC Scheme, the type, semantics, and an encoding format for the FEC Payload ID and the FEC Object Transmission Information (OTI) are defined.

The FEC OTI contains information which is essential to the decoder in order to decode the coded/repair object. Common FEC OTI elements for fully-specified FEC schemes are defined in clause 6.2.4 of IETF RFC 5052 [15], and include: `Transfer-Length`, `Encoding-Symbol-Length`, `Maximum-Source-Block-Length`, and `Max-Number-of-Encoding-Symbols`.

In addition to common information, scheme-specific FEC OTI element may be used by an FEC Scheme to communicate information that is essential to the decoder and that cannot adequately be represented within the FEC OTI elements. The FEC Scheme defines the structure of this octet string, which may contain multiple distinct elements.

FEC schemes are identified by an *FEC Encoding ID*, which is an integer identifier assigned by IANA (https://www.iana.org/assignments/rmt-fec-parameters/rmt-fec-parameters.xhtml).

The *FEC Payload ID* contains information that indicates to the FEC decoder the relationships between the encoding symbols carried by a particular packet. For example, if packet subatom carries a source symbol, then the FEC Payload ID identifies which source symbol of the object are carried by the packet.

The FEC Payload ID may also contain information about larger groups of encoding symbols of which those contained in the packet (or packet group) are part of. For example, the FEC Payload ID may contain information about the source block the symbols are related to.

D.1.4 FEC Scheme information in CMMF

CDPs are required to provide a mechanism to transport the FEC Encoding ID, FEC Payload ID, common FEC OTI, and scheme-specific information defined by the FEC scheme. CMMF carries this information within the bitstream syntax defined in clause 5 and clause 6. Primarily this information is carried within the `rfc5052_information()` structure present when `b_rfc5052` is TRUE. The Payload ID information is defined as part of the `packet_header()`, `packet_group_header()`, and `mbpg_header()` structures that are part of packet, packet group, and multiple block packet groups respectively.

NOTE: FEC Schemes define a binary representation of the parameters, but for example the FLUTE delivery protocol as defined in IETF RFC 6726 [18] as used in the instantiation in clause D.2 specifies an XML-based encoding format for these elements to be used during delivery. The present document permits the use of the following fully specified FEC schemes when CMMF is used as a CDP (i.e. when `b_rfc5052` is TRUE):

- Raptor Forward Error Correction Scheme as defined in IETF RFC 5053 [16].
- RaptorQ Forward Error Correction Scheme as defined in IETF RFC 6330 [13].
- Reed-Solomon Forward Error Correction (FEC) with the special case of Reed-Solomon codes over GF{2^8} as defined in IETF RFC 5510 [14].

The parameters and coding size for the FEC scheme are provided in Table D.1. This table also describes how to map the FEC scheme information to CMMF bitstream elements.

Table D.1: Parameters and Coding of FEC schemes used in CMMF

Parameter	IETF RFC 5053 [16] (Raptor)	IETF RFC 6330 [13] (RaptorQ)	IETF RFC 5510 [14] (Reed-Solomon GF{2^8})
FEC Encoding ID value	1 (see section 3.2.1)	6 (see section 3.3.1)	5 (see section 5)
FEC Payload ID size	4 byte (see section 3.1)	4 byte (see section 3.2)	4 byte (see section 5.1)
Source Block Number (SBN) size	16 bit	8 bit	24 bit
Encoding Symbol ID (ESI) size	16 bit	24 bit	8 bit
Maximum Transfer Length	2^^45 bytes (i.e. 32 terabytes)	2^^40 bytes (i.e. 1 terabyte)	2^^42 bytes (i.e. 4 terabytes)
Maximum Source Block size	8192	56403	255
FEC Object Transmission Information (FEC-OTI) Common	10 byte (see section 3.2.2)	8 byte (see section 3.3.2)	10 byte (see section 5.2.4.1)
Transfer-Length size F	48 bit	40 bit	48 bit
Reserved	16 bit	8 bit	

Parameter		IETF RFC 5053 [16] (Raptor)	IETF RFC 6330 [13] (RaptorQ)	IETF RFC 5510 [14] (Reed-Solomon GF{2^8})
	Encoding-Symbol-Length T size	16 bit	16 bit	16 bit
	Maximum Source Block Length size	n/a	n/a	8 bit
	maximum number of encoding symbols max_n size	n/a	n/a	8 bit
FEC Object Transmission Information (FEC-OTI) Specific		4 byte (see section 3.2.3)	4 byte (see section 3.3.3)	See section 5.2.3
	number of source blocks (Z)	16 bit	8 bit	n/a
	number of sub-blocks (N)	8 bit	16 bit	n/a
	symbol alignment parameter (Al)	8 bit	8 bit	n/a
FEC Object Transmission Information (FEC-OTI) size		14 byte	12 byte	10 byte

The mapping from the FEC scheme parameters to CMMF bitstream elements/fields are provided in Table D.2. This table also describes how to map the FEC scheme information to CMMF bitstream elements.

Table D.2: FEC scheme parameters to CMMF mapping

Parameter		CMMF Bitstream Parameter	Mapping Function	Relevant clause(s)
FEC Encoding ID		`code_type` in `bitstream_header()`	`code_type` = FEC Encoding ID	6.1.4
FEC Payload ID				
	Source Block Number (SBN)	Either `packet_block_index` in `packet()`, `packet_group_block_index` in `packet_group()`, or `mbpg_source_block_index` in `mbpg_header()`	packet_block_index = SBN, packet_group_block_index = SBN, or mbpg_source_block_index = SBN	6.1.6.1, 6.1.18.1, 6.1.23.3
	Encoding Symbol ID (ESI)	Either `packet_symbol_index` in `packet_header()`, `packet_group_symbol_index` in `packet_group_header()`, or `mbpg_symbol_index` in `mbpg_header()`	packet_symbol_index = ESI, packet_group_symbol_index = ESI, mbpg_symbol_index = ESI	6.1.7.4, 6.1.19.4, 6.1.23.3
FEC Object Transmission Information (FEC-OTI) Common				
	Transfer-Length size F	`content_source_size` in `bitstream_header()` (assuming source itself has not been coded	content_source_size = F transfer_length = F	6.1.4.1, 6.1.4.6
	Reserved	--	--	--
	Encoding-Symbol-Length T	`block_symbol_size` in `block_header()` `encoding_symbol_length` in `rfc5052_information()`	block_symbol_size = T	6.1.5.3, 6.1.4.6
	Maximum Source Block Length (B)	`block_num_symbols` in `block_header()`	block_num_symbols = B	6.1.5.5
	maximum number of encoding symbols (max_ns)	There is no equivalent for this parameter in CMMF. However a related parameter is `block_max_symbol_index` in `block_header()`	block_max_symbol_index = max_ns - 1	6.1.5.8

The present document does not define a complete binary representation of the FEC-OTI information, but a receiver, based on the information in Table D.1 and Table D.2 may reconstruct the FEC-OTI information in binary form for each FEC scheme.

D.1.5 Configuration Information parameters

The Configuration Information allows the CMMF client to map an application request to CMMF receiver operations. Table D.3 provides examples of possible Configuration Information that can be used where source and coded/repair transport objects are described. Some use cases may require additional information or only a subset of this information, and a simpler version of this parameter set may be used.

Table D.3: Example Configuration Information parameters

Parameter	Usage	Definition
Complete	OD	Indicates whether the Configuration Information is complete.
Location	O	Provides information where the Configuration Information can be accessed in carried externally.
Expires	M	Provides information when this Configuration Information is no longer valid and an update is needed, for example using a reload from Location.
Source Flow	1 … S	Provides 1 … S source flows.
TSI	M	Identifier of the source flow.
Object	1 … N	Provides 1 … N objects in the source flow.
TOI	M	Transport Object Identifier (TOI) value that represents the source object.
Size	M	Size of the transmission object in bytes.
Content-Type		Describes media type of file.
Encoding		Describes encoding of file, for example whether the file is zipped prior to transfer.
Message Digest		Message digest of file.
Associated URI		Name, Identification, and Location of file (specified by the URI).
Access URL		The URL where the source object can be accessed. If the field is not present, then the source flow is not directly accessible.
availabilityStartTime		Provides a wall-clock time when the resource is accessible.
availabilityStartTime		Provides a wall-clock time when the resource ceases to be available.
<Additional metadata>		May include cache or entity tag (E-Tag) metadata.
Representation		Refers to a DASH Representation in an MPD or a Track in an HLS manifest.
Coded/Repair Flow	1 … R	Provides 1 … R coded/repair flows.
TSI	M	Identifier of the coded/repair flow.
Object	1 … N	Provides 1 … N objects in the coded/repair flow.
TOI	M	Transport Object Identifier (TOI) value that represents the coded/repair object.
FEC-OTI		If object is coded using a scheme based on IETF RFC 5052 [15], FEC Object transmission information including the FEC Encoding ID and, if relevant, the FEC Instance ID.
includedSourceTOI	M	List of (TSI, TOI pairs) of the included source transport objects forming an aggregated object. Typically, only a single pair is provided.
Content-Type		Media Mime Type of the file.
completeObject	OD FALSE	Indicates whether the transport object includes sufficient information to recover all files included in this coded/repair object.
symbolArrangement	O	Provide this symbol arrangement in the object according to Table 78. If not present, the symbol Arrangement is unknown and only present in the bitstream.
sAParameters	O	may be present if the symbolArrangement is present. If present, it provides the parameters assigned to the symbol arrangement as defined in Table 36. For arrangement 2 and 3, this is a comma-separated list of: • Index difference • Symbol group • Index in symbol group
Access URLs	1, …, N	The URLs where the coded/repair object can be accessed.
availabilityStartTime	OD	Provides a wall-clock time, when the resource is accessible. If not present, it is assumed that the resource is accessible during the validity time of the CI.

Parameter	Usage	Definition
availabilityEndTime	OD	Provides a wall-clock time, when the resource ceases to be available. If not present, it is assumed that the resource is accessible during the validity time of the CI.
<Additional metadata>		Additional metadata may be present for example caching information, processing information, etc.

Usage
M: Mandatory parameter
O: Optional parameter
OD: Optional parameter, if not set default value applies
A, …, B: between A times (typically 0 or 1) and B times (typically 1 or N) parameters may be present

D.1.6 Example Instantiations

Based on FEC schemes and FEC framework defined in the IETF, the remainder of this clause defines an instantiation for CMMF when being used as a CDP. Specifically, an instantiation provides the following information:

- The definition of delivery session.
- An instantiation of the Configuration Information (CI), with related information and a delivery documents.
- The definition of source objects.
- The definition of coded/repair objects based on the bitstream format defined in clause 5.

D.2 FLUTE-based CMMF CDP Instantiation

D.2.1 Introduction

This clause defines an instantiation of the CMMF CDP framework using the FLUTE delivery protocol, in particular the File Delivery Table (FDT) defined in FLUTE. The instantiation is aligned with IETF RFC 5052 [15] as well as IETF RFC 6726 [18], and reuses some of the extensions and concepts defined ETSI TS 126 346 [19], namely the FDT extensions.

The *FLUTE-based CMMF CDP Instantiation* is defined as follows:

- Typical procedures for a FLUTE-based *CMMF CDP Instantiation* are provided in clause D.2.2.
- The CI can be expressed as a File Delivery Table (FDT) as defined in IETF RFC 6726 [18] with minimum extensions to support CMMF, referred to as Extended FDT (EFDT). The EFDT is a document containing one instance. For details refer to clause D.2.3.
- The delivery session is a single Transport Session, and different coded/repair objects may be generated. Each transport object is identified by a unique Transport Object Identifier (TOI).
- Source objects may be distributed within a transport session.
- The coded/repair objects are compatible with CMMF Bitstream as defined in clause 6 (where the `b_rfc5052` field is set to TRUE), but also with ALC Payload formats as defined in IETF RFC 5775 [17] with some additional headers as documented in clause D.2.4.
- Only a single object is associated with a TOI.
- Only the fully specified FEC schemes listed in clause D.1.4 are allowed, i.e. the FEC Encoding ID shall either be set to 1 (Raptor), 5 (Read-Solomon over GF{2^8}), or 6 (RaptorQ).

D.2.2 Procedures for FLUTE-based CMMF CDP Instantiation

A baseline procedure describing the establishment of a FLUTE-based CMMF CDP transport session is shown in Figure D.1.

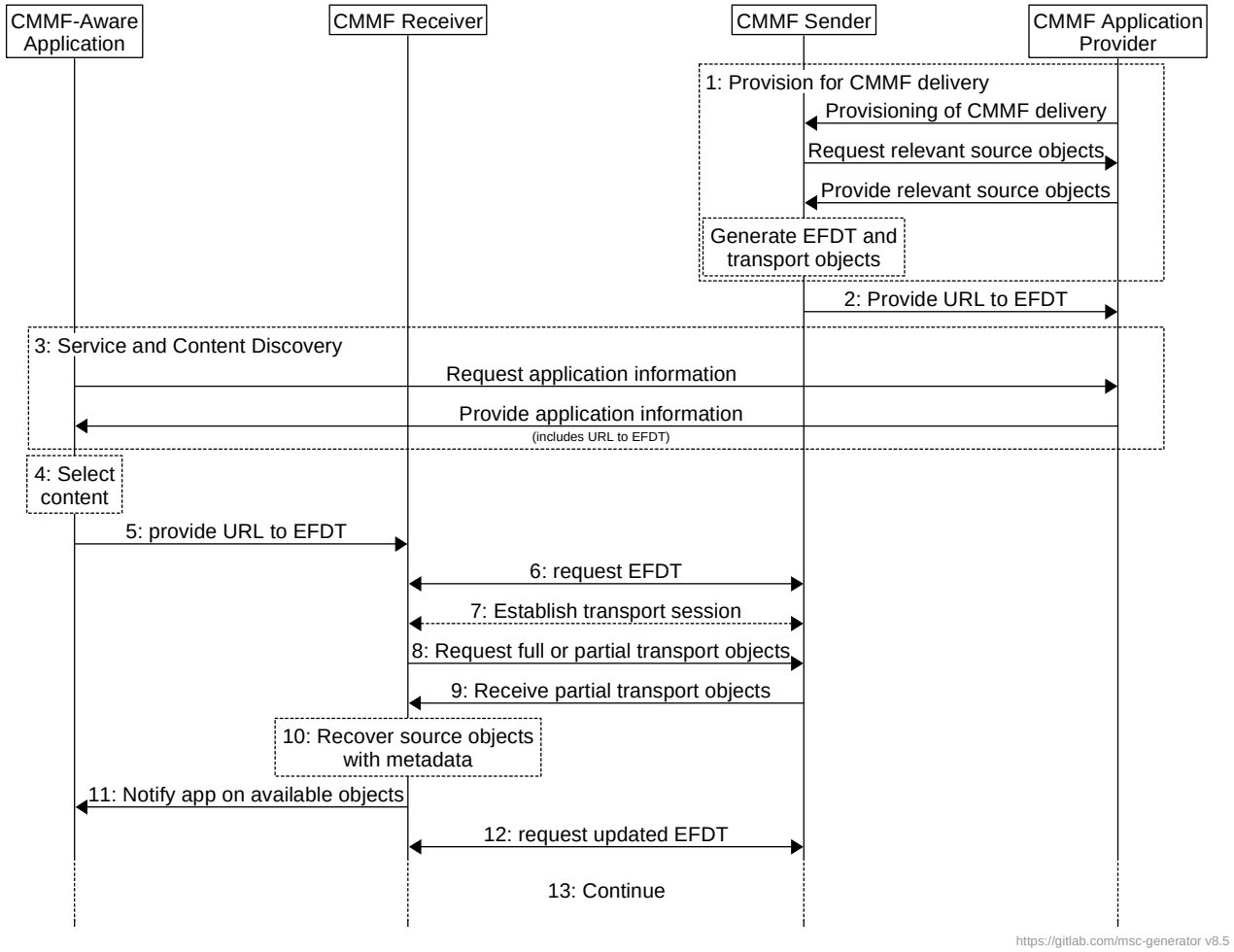

Figure D.1: High Level Procedure for a FLUTE-based CMMF delivery

Prerequisites:

1: The CMMF Application Provider has provisioned the CMMF Sender and has set up content ingest to provide source objects. The CMMF Sender uses the source object to generate transport objects and the EFDT.

2: The CMMF Sender provides the EFDT URL to the Application provider.

3: The CMMF-Aware Application triggers Application Information and Content Discovery procedure. The Application information includes a URL to the EFDT.

Steps:

4: Content is selected.

5: The CMMF-aware Application triggers the CMMF receiver to start the accessing content. The Configuration Information location is provided to the CMMF Receiver.

6: The CMMF Receiver interacts with the CMMF Sender to acquire the Configuration Information.

7: The CMMF Receiver establishes the transport session with the CMMF Sender.

8: The CMMF Receiver sends requests for downloading partial or complete transport objects.

9: The CMMF Receiver receives the partial transport objects and uses those for recovery, when sufficient data is available.

10: The CMMF Receiver recovers source objects together with metadata.

11: The CMMF Receiver notifies the CMMF-Aware application on available source objects and the application access those at appropriate time based on the associated metadata.

12: The CMMF Receiver continuously receives additional transport objects, if not yet complete, based on the EFDT and updated instances of the EFDT.

D.2.3 Extended File Delivery Table

D.2.3.1 Semantics

In order to provide relevant Configuration information to the CMMF receiver a document including an extended File Delivery Table (EFDT) is created to signal CI information to the receiver as follows:

- The attributes and elements from the File Delivery Table in IETF RFC 6726 [18], section 3.2, are re-used as follows:
  - The `Expires` attribute expresses the validity of the EFDT instance, i.e. how long the binding is valid.
  - The `Complete` attribute, when `TRUE`, signals that this "EFDT Instance" includes the set of "File" entries that exhausts both the set of files delivered so far and the set of files to be provided in the session.
  - The following data may be set as default on instance level, or may set on file level. If set on file level, the instance level information is overwritten:
    - The `Content-Type` attribute is included to express the type of the delivered file according to IETF RFC 9110 [i.5].
    - The `Content-Encoding` attribute is included to express the encoding of the delivered file. For details refer to IETF RFC 6726 [18].
    - The `FEC-OTI-FEC-Encoding-ID` attribute provides the "FEC Encoding ID" Object Transmission Information element defined in IETF RFC 5052 [15].
    - The `FEC-OTI-Maximum-Source-Block-Length` attribute provides the "Maximum-Source-Block-Length" Object Transmission Information element defined in IETF RFC 5052 [15], if required by the FEC Scheme.
    - The `FEC-OTI-Max-Number-of-Encoding-Symbols` attribute provides the "Max-Number-of-Encoding-Symbols" Object Transmission Information element defined in IETF RFC 5052 [15], if required by the FEC Scheme.
  - The `TOI` value is a positive integer to express the Transport Object Identifier and identify the object.
  - The attribute `Content-Location` is used for the purpose defined in IETF RFC 9110 [i.5].
  - The attribute `Content-Length` is used for the purpose defined in IETF RFC 9110 [i.5].
  - The attribute `Transfer-Length` is used to carry the transfer length if the file is content encoded before transport (and thus the "Content-Encoding" attribute is used), e.g. if compression is applied before transport to reduce the number of octets that need to be transferred, then the transfer length is generally different.
  - The `Content-MD5` attribute is used for the purpose defined in IETF RFC 2616 [i.4].
- The following new elements and attributes are defined in order access the content:
  - A list of access objects assigned to a File, each defined by:
    - An `access-URL` where the object associated to the TOI can be accessed.

- A `type` attribute that provides the type of the transport object, whether it is a source object or if it is an object according to the bitstream syntax of CMMF.
- If the type is a CMMF bitstream, then an attribute `symbolArrangement` may be added and the value is set according to Table 78. If not present, the symbol Arrangement is unknown and only present in the bitstream. For details see the semantics in clause D.2.4.3.
- If the type is a CMMF bitstream and attribute `symbolArrangement` is set to 2 or 3, the following information may be added as a comma-separated list:
  - the total amount of symbol groups, *C*.
  - the included symbol group, *c*.
  - the index of the first symbol of the symbol group.
- An attribute `complete` if set to TRUE, the file includes sufficient information to recover the source object included in this transport object.
- An availability time window expressed by a an `availabilityStartTime` and an `availabilityEndTime` attribute indicating when the URL is available. If not present, it is assumed that the objects are present at the time when accessing the EFDT until the Expires time of the EFDT.

D.2.3.2 Extended FDT Schema for CMMF

An extended FDT schema is provided below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
 xmlns="urn:ETSI:CMMF:2023:EFD"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="urn:ETSI:CMMF:2023:EFD"
           elementFormDefault="qualified"
           version="1">
   <xs:element name="FDT-Instance" type="FDT-InstanceType"/>
   <xs:complexType name="FDT-InstanceType">
      <xs:sequence>
         <xs:element name="File" type="FileType" maxOccurs="unbounded"/>
         <xs:element name="schemaVersion" type="xs:unsignedInt"/>
         <xs:element name="delimiter" type="DelimiterType"/>
         <xs:any namespace="##other" processContents="skip" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Expires" type="xs:dateTime" use="required"/>
      <xs:attribute name="Complete" type="xs:boolean" use="optional"/>
      <xs:attribute name="Content-Type" type="xs:string" use="optional"/>
      <xs:attribute name="Content-Encoding" type="xs:string" use="optional"/>
      <xs:attribute name="FEC-OTI-FEC-Encoding-ID" type="xs:unsignedLong" use="optional"/>
      <xs:attribute name="FEC-OTI-FEC-Instance-ID" type="xs:unsignedLong" use="optional"/>
      <xs:attribute name="FEC-OTI-Maximum-Source-Block-Length" type="xs:unsignedLong"
use="optional"/>
      <xs:attribute name="FEC-OTI-Encoding-Symbol-Length" type="xs:unsignedLong" use="optional"/>
      <xs:attribute name="FEC-OTI-Max-Number-of-Encoding-Symbols" type="xs:unsignedLong"
use="optional"/>
      <xs:attribute name="FEC-OTI-Scheme-Specific-Info" type="xs:base64Binary" use="optional"/>
      <xs:anyAttribute processContents="skip"/>
   </xs:complexType>
   <xs:complexType name="FileType">
      <xs:sequence>
         <xs:element name="EncodedObjects" type="EncodedObjectType" minOccurs="1"
maxOccurs="unbounded"/>
         <xs:element name="delimiter" type="DelimiterType"/>
         <xs:any namespace="##other" processContents="skip" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="Content-Location" type="xs:anyURI" use="required"/>
      <xs:attribute name="TOI" type="xs:positiveInteger" use="required"/>
      <xs:attribute name="Content-Length" type="xs:unsignedLong" use="optional"/>
      <xs:attribute name="Transfer-Length" type="xs:unsignedLong" use="optional"/>
      <xs:attribute name="Content-Type" type="xs:string" use="optional"/>
      <xs:attribute name="Content-Encoding" type="xs:string" use="optional"/>
      <xs:attribute name="Content-MD5" type="xs:base64Binary" use="optional"/>
      <xs:attribute name="FEC-OTI-FEC-Encoding-ID" type="xs:unsignedLong" use="optional"/>
```

```
      <xs:attribute name="FEC-OTI-FEC-Instance-ID" type="xs:unsignedLong" use="optional"/>
      <xs:attribute name="FEC-OTI-Maximum-Source-Block-Length" type="xs:unsignedLong"
use="optional"/>
      <xs:attribute name="FEC-OTI-Encoding-Symbol-Length" type="xs:unsignedLong" use="optional"/>
      <xs:attribute name="FEC-OTI-Max-Number-of-Encoding-Symbols" type="xs:unsignedLong"
use="optional"/>
      <xs:attribute name="FEC-OTI-Scheme-Specific-Info" type="xs:base64Binary" use="optional"/>
      <xs:anyAttribute processContents="skip"/>
   </xs:complexType>
   <xs:complexType name="EncodedObjectType">
      <xs:simpleContent>
         <xs:extension base="xs:anyURI">
            <xs:attribute name="symbolArrangement" type="interleavingType"/>
            <xs:attribute name="sAParameters" type="xs:string"/>
            <xs:attribute name="objectType" type="ObjectTypeType" default="source"/>
            <xs:attribute name="complete" type="xs:boolean" default="false"/>
            <xs:attribute name="availabilityStartTime" type="xs:dateTime"/>
            <xs:attribute name="availabilityEndTime" type="xs:dateTime"/>
            <xs:anyAttribute namespace="##other" processContents="skip"/>
         </xs:extension>
      </xs:simpleContent>
      <xs:anyAttribute processContents="skip"/>
   </xs:complexType>
   <xs:simpleType name="DelimiterType">
      <xs:restriction base="xs:byte"/>
   </xs:simpleType>
   <xs:simpleType name="ObjectTypeType">
      <xs:restriction base="xs:string">
         <xs:enumeration value="source"/>
         <xs:enumeration value="cmmf"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="interleavingType">
      <xs:restriction base="xs:unsignedLong">
         <xs:enumeration value="0"/>
         <xs:enumeration value="1"/>
         <xs:enumeration value="2"/>
         <xs:enumeration value="3/>
      </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

D.2.3.3 Extended FDT Description for CMMF

The Extended FDT Description (EFD) is a document that contains metadata required by a CMMF Client/CMMF-Aware application to access Objects and to provide the CMMF session. The EFD is an XML document that shall be formatted according to the XML schema provided in clause D.2.3.2.

The EFD shall be authored in such a way that after removing any XML attributes or elements that are not part of the specified XML schema (documented in clause D.2.3.2), the remaining document should still be a valid XML document that conforms to the schema and the present document. Additionally, if all XML attributes and elements from the EFD namespace and other namespaces that are not part of the specified XML schema are removed, the resulting document should still be a valid XML document that complies with the present document.

The MIME type of the EFD document is defined in clause D.2.3.4. The encoding of the EFD and all data provided in extension namespaces shall be UTF-8 as defined in IETF RFC 3629 [11]. If binary data needs to be added, it shall be included in Base64 as described in IETF RFC 4648 [12] within a UTF-8 encoded element with a proper name space or identifier, such that an XML parser knows how to process or ignore it.

The delivery of the EFD is outside the scope of the present document. However, if the EFD is delivered over HTTP, then the EFD document may be transfer encoded for transport, as described in IETF RFC 9110 [i.5].

Selected EFD examples are provided in clause D.2.5.

D.2.3.4 IANA registration for Extended FDT Description

This clause provides the formal MIME type registration for the EFD. It is referenced from the registry at http://www.iana.org/.

The MIME Type and Subtype are defined as follows:

- MIME media type name: `application`
- MIME subtype name: `cmmf-efd+xml`
- Required parameters: None
- Optional parameters: None.
- Encoding considerations: UTF-8
- Security considerations: The EFD contains references to other resources. It is coded in XML, and there are risks that deliberately malformed XML can cause security issues. In addition, an EFD can be authored that causes receiving clients to access other resources; if widely distributed, this can be used to cause a denial-of-service attack.

  The EFD format does not incorporate any active or executable content. However, other forms of material from outside sources can be referenced by an EFD, and this material can contain active or executable content. Such material is expected to be identified by its own MIME type, and the security considerations of that format should be taken into account.

  If operating in an insecure environment and required by the content/service provider, elements and attributes of EFD may be encrypted to protect their confidentiality by using the syntax and processing rules specified in the W3C Recommendation "XML Encryption Syntax and Processing".

  If operating in an insecure environment and required by the content/service provider, the digital signing and verification procedures specified in the W3C Recommendation "XML Signature Syntax and Processing" may be used to protect data origin authenticity and integrity of the EFD.
- Interoperability considerations: The present document defines a platform-independent expression of a document, and it is intended that wide interoperability can be achieved.
- Published specification: ETSI TS 103 973 (the present document)
- Applications which use this media type: Various
- Additional information:
- File extension(s): `efd`
- Intended usage: common
- Other information/general comment: None
- Author/Change controller: ETSI

D.2.4 Transport object formats

D.2.4.1 General

Transport objects are either source transport objects or coded/repair objects.

Source transport objects are unmodified source objects, for details see clause D.2.4.2.

Coded/repair objects conform to the CMMF bitstream format, for details see clause D.2.4.3.

D.2.4.2 Source objects

If the EFD signals a `type` equal to `"source"`, then the object referenced in the `Access-URL` is the original source object. The value of the `Access-URL` may be identical to the value of the `Content-Location`, if the application can also access the object at the original location. The source object as a whole or partially may be used by the CMMF receiver to reconstruct the application object, possibly in combination with coded/repair objects.

D.2.4.3 Coded/repair objects

D.2.4.3.1 General

Coded/repair objects for the FLUTE-based instantiation shall conform to the CMMF bitstream format. In addition, the coded/repair objects are aligned with ALC Payload formats as defined in IETF RFC 5775 [17], in a sense that it includes an encoding that allows the receiver to recover a list of FEC Payload IDs and the corresponding encoding symbols from the CMMF bitstream information. The FEC Payload ID semantics is the Source Block Number (SBN) and Encoding Symbol ID (ESI) assigned to each symbol.

Coded/repair objects are a restricted CMMF bitstream with the following constraints:

- The following subatoms shall be present in the following order:
  - Exactly one instance of `bitstream_header()` as defined in clause 5.2.4 and clause 6.1.4.
  - Multiple instances of `block_header()` as defined in clause 5.2.5 and clause 6.1.5, one instance per number of source blocks.
  - Exactly one instance of `multi_block_packet_group()` as defined in clause 5.2.25 and clause 6.1.22.
- The `bitstream_header()` shall be restricted as follows:
  - `content_source_size` shall be set to the value of the `Transfer-Length` size as included in the FEC-OTI information as defined in clause D.1.4 and carries the FEC-OTI Information `Transfer-Length` size according to IETF RFC 5052 [15].
  - `content_source_type` shall be set to `000b`.
  - `code_type` shall be set to either 1, 5, or 6 and carries the FEC Encoding ID value according to IETF RFC 5052 [15].
  - `b_rfc5052` shall be set to 1.
  - The information in `rfc5052_information()` as defined in clause 5.2.20 and clause 6.1.4.6 shall be set as follows (using the size headers in Table D.1):
    - the `encoding_symbol_length` shall be set to the value of the `FEC-OTI-Encoding-Symbol-Length` size as included in the FEC-OTI information as defined in clause D.1.4 and carries the `FEC-OTI-Encoding-Symbol-Length` according to IETF RFC 5052 [15].

NOTE 1: CMMF in general permits to have different encoding symbol length for each block. For IETF RFC 5052 [15] based `code types` this is not permitted and hence the encoding symbol length is signalled in the bitstream header.

- If `code_type` is set to 1 (Raptor) or set to 6 (RaptorQ), no additional parameters are set. The scheme-specific FEC-OTI parameters "number of sub-blocks (N)" and the "symbol alignment parameter (Al)" are assumed to be set to 1, i.e. sub-blocking shall not be used in the context of CMMF.
- If `code_type` is set to 5 (Reed-Solomon), two additional fields are added and set as follows:
  - The `maximum_source_block_length` shall be set to the `FEC-OTI-Maximum-Source-Block-Length (B)` as defined in IETF RFC 5510 [14].
  - Maximum number of encoding symbols `max_n` size shall be set to the shall be set to the `FEC-OTI-Max-Number-of-Encoding-Symbols (max_n)` as defined in IETF RFC 5510 [14].

- `block_count_minus1` shall be set to:
  - For `code_type` set to 1 and 6: FEC-OTI information "number of source blocks (Z)" minus 1 and carries the scheme-specific value number of source blocks (Z) as defined in IETF RFC 5053 [16] and IETF RFC 6330 [13], respectively.
  - For `code_type` set to 5: the number of source blocks (N) - 1 as defined in clause 9 of IETF RFC 5510 [14].

NOTE 2: With the above information, the FEC decoder following the respective IETF RFC may be instantiated by creating the encoded FEC Object Transmission Information as an octet field consisting of the concatenation of the encoded Common FEC Object Transmission Information and the encoded Scheme-specific FEC Object Transmission Information.

- The following flags shall not be set:
  - `b_content_source_split.`
  - `b_content_block_separate_sources.`
  - `b_profile_information_present.`
  - `b_block_cc_encrypted.`

- The `block_header()` instances shall be restricted as follows:
  - There shall be exactly `block_count` instances of `block_header()`, where `block_count = block_count_minus1 + 1`.
  - `block_header()` instances shall have ascending `block_index` values from `0` to `block_count - 1`.
  - Some values shall be set to the values described by the mapping in Table D.2. This includes the values for:
    - `block_symbol_size` shall be set to the `encoding_symbol_length` as defined in the `rfc5052_information()` in the `bitstream_header()`.
    - `block_num_symbols` shall be set based according to the source block partitioning rules of the FEC scheme.
    - `block_max_symbol_index (b_block_max_symbol_index_present` shall be set to `TRUE)`.
  - Other values shall be set to the values derived from the source blocking algorithm from the FEC scheme. This includes the values for:
    - `block_size` shall be set to `block_num_symbols × block_symbol_size` for all blocks except of the last one, i.e. `block_index` set to `block_count - 1`. The last one is set taking into account the remaining octets to fill up the transfer length.
  - `block_mask` shall be set to `B0000Bb`, where values marked as B (bit 0: sufficient symbols present and bit 5: block integrity present) can be either 0 or 1 depending on the FEC scheme and CMMF encoder settings.
  - The following flags shall not be set:
    - `b_block_content_source_index_present.`
    - `b_block_composite_sources.`
    - `b_addl_block_coding_info_present.`

- The `multi_block_packet_group()` shall be restricted as follows:
  - `mbpg_index` shall be set to 0.
  - `mbpg_start_block_index` shall be set to 0.

- `mbpg_num_blocks` shall be set to `block_count`.
- `mbpg_num_symbols` shall be set to the number of symbols contained in the transport object, i.e. `ceil(content_source_size/encoding_symbol_length)`.
- The `mbpg_header()`shall be restricted as follows:
  - `mbpg_symbol_arrangement` shall be set to either 2 or 3.
  - The symbols are arranged according to clause D.2.4.3.2 with the:
    - `mbpg_index_difference` shall be set to the number of symbol groups *C*.
    - `mbpg_first_symbol_group_subset_index` shall be set to the index of the first symbol within the symbol group. If the value of `mbpg_symbol_group_subset_index` is 0, then `b_mbpg_is_symbol_group_subset` shall be set to FALSE.
    - `mbpg_first_symbol_index` shall be set to the symbol group that is included in the multi-block packet group, *c*.
- `b_mbpg_integrity_present` shall be set to 0.
- `b_mbpg_header_ext_present` shall be set to 0.

D.2.4.3.2 Mapping of FEC Payload ID information to transport blocks

D.2.4.3.2.1 General

The information in this clause relates to the Encoding of CMMF as defined in clause 4.2.1 and is specific when using CDP-based instantiation.

The following parameters are assumed to be available:

- A total number of source symbols *Kt*, available in `mbpg_num_symbols`.
- The source object is partitioned into `block_count` number of source blocks, referred to as *Z*.
- For each source block $z = 0, \ldots, Z-1$:
  - Assuming that source blocks are covering the source object starting `block_index` z=0 at the beginning, and each of the source blocks has the same symbol size.
  - The source symbols for each source block z are defined as $x_{z,i}$ with source symbols index $i=0, .., Ks[z]-1$ and $Ks[z]$ the source block size in symbols of source block z indicated by `block_num_symbols` in clause 4.2.1.
  - The encoded symbols for each source block z are defined as $y_{z,j}$ with encoding symbol index $j = 0, .., Ke[z] - 1$ and $Ke[z]$ the number of encoding symbols assigned to source block z and $Ke[z] \geq Ks[z]$.

NOTE: For some codes/FEC schemes $Ke[z]$ may be quite restrictive (for example for Reed-Solomon codes), for other codes with a fountain property, such as Raptor codes, many symbols may be generated.

  - For systematic codes, the first $Ks[z]$ encoding symbols are identical to the source symbols, i.e. $y_{z,j} = x_{z,j}$ for $j = 0, \ldots, Ks[z]$.

The encoding symbols of all source blocks are then distributed across one or multiple symbol groups.

Several *symbol groups* (as defined in clause 4.2.2) are formed, each including one or multiple symbols from one or multiple source blocks. As defined in clause 4.2.2, symbols groups may be distinct, i.e. each encoding symbols is added exactly once to exactly one *symbol group*. The arrangement of symbols in a symbol group may follow a specific pattern arrangement. Symbol groups may also have assigned a property of *completeness*, i.e. the source object can be recovered from all symbols in the symbol group.

Assume the definition of *C* symbols groups, indexed with *c*=0,…, *C*-1. Each symbol groups *c* includes *Kc*[*c*] encoding symbols, indexed from *k* = 0, …, *Kc*[*c*] - *1*. The symbols in the symbol group are referred to as $w_{c,k}$ for *k =0*, …, *Kc*[*c*] and *c*=0,…, *C*-1. Symbols in a symbol group may come from different encoded blocks.

Based on these *symbol groups*, *multi-block packet groups* are formed. *Multi-block packet groups* represent a subset of symbol groups with a consecutive set of symbols of exactly one symbol group and fully defined by the *index of the first symbol* within the symbol group, and the *number of symbols* included in the *multi-block packet group*.

The information on which encoding symbols are included in the *multi-block packet group* as well their order is provided in a *multi-block packet group header* by the `mbpg_symbol_arrangement` field.

Figure D.2 provides an overview on the formation of transport objects based on a source object. Note the differences between the partitioning algorithm presented here compared to that described in clause 4.2.1.

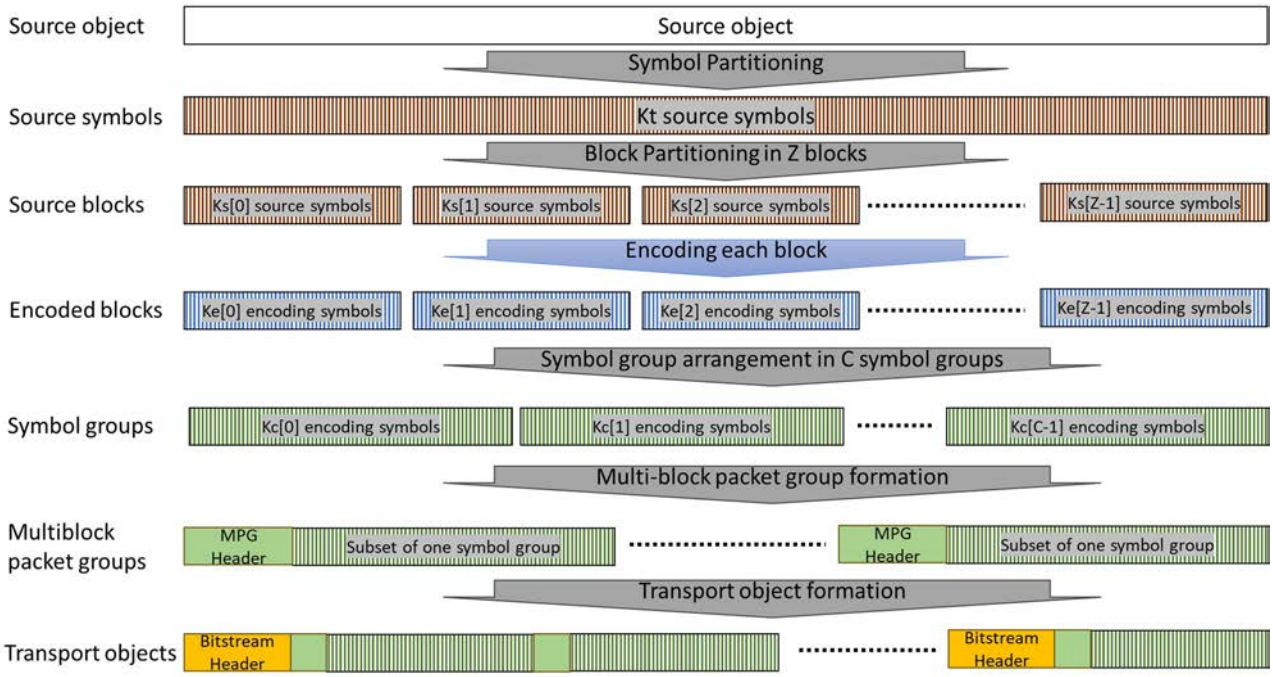

Figure D.2: Formation of transport objects from source objects

On the symbol group arrangements, an arbitrary sequence of coded symbols for different source blocks may be collected in a symbol group. In this case, MBPG header needs to include the list of pairs of source block *z* and symbol identifier *i* assigned to each position *k* in the symbol group *c*.

The present document also defines symbol group arrangements that apply certain patterns, and for which only a few parameters need to be added to map the position *k* in the symbol group *c* and the position of source block *z* and symbol identifier *i*.

For the symbol group arrangement `mbpg_symbol_arrangement` with value set to 3 or 2, the arrangements are provided in clause D.2.4.3.2.2 and clause D.2.4.3.2.3, respectively.

In Figure D.2 the formation of multi-block packet groups from symbol groups is accomplished by extracting a contiguous range of symbols from a symbol group, starting from `mbpg_first_symbol_index` in symbol group `mbpg_symbol_group`, c.

D.2.4.3.2.2 Symbol group arrangement 3: encoding symbol interleaving

This pre-defined symbol arrangement is motivated by a scenario for which typically the source symbols are not included in the transport object, either because the code is non-systematic, or only repair symbols are included. In addition, the symbols on each transport object are distinct. This symbol group arrangement corresponds to an `mbpg_symbol_arrangement` value of 0011b.

In this example, transport objects may be generated with encoding symbols included. On top of this:

1) the transport object includes sufficient information to recover the included source object;

2) if only a subset of several transport objects is accessible, then an initial byte range of a subset of transport objects is sufficient to recover the source object, and the required number of total bytes is small.

The only parameter for this symbol arrangement is the number of symbol groups *C*.

In order for the arrangement to be effective, it is assumed that the source blocks are of equal or at least similar size. In the case of the CDP-based instantiation, this condition is fulfilled.

For defining the symbol group arrangement, the following is defined:

- Total symbol count *TOTAL* added to symbol groups, initialized to 0.
- Next position in symbol group *c* that can add a symbol, *POS*[*c*], initialized to 0.
- Next symbol index in source block *z* that has not yet been added to any transport object, ESI[*z*], initialized to the index of the first non-source symbol, i.e. 0 for non-systematic code, and *Ks*[*z*] for systematic codes.
- Counter for number of symbols included in container *c* from source block *z*, *SYMBOLS*[*o*,*z*], initialized to 0.

In addition, a completeness check is carried out for the entire arrangement across different symbol groups. Only if each symbol group includes a sufficient set of symbols to recover all included source blocks, adding symbols is stopped.

The arrangement as defined in clause 6.1.23.1.

Figure D.3 (aligned with Figure 13) provides an example for encoding symbol arrangement with *Kt=29*, *Z=3*, *Ks[0] = Ks[1]=10, Ks[2]=9, Ke[0]=Ke[1]=30, Ke[2]=27* across *C=3* symbol groups. The bold ESI number shows the symbols that makes a source block complete for each symbol group. Symbols are added to each symbol group until all symbol groups are complete.

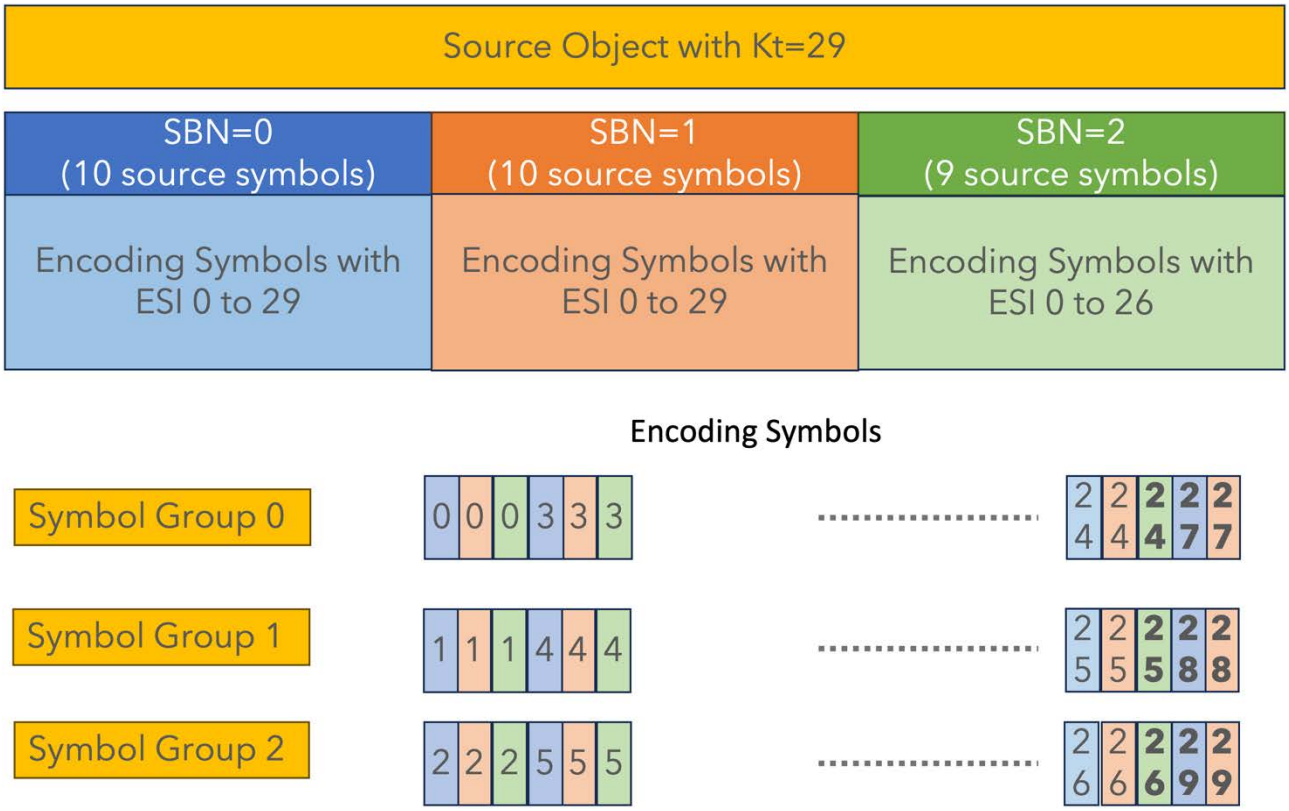

Figure D.3: Symbol arrangement for arrangement 3

D.2.4.3.2.3 Symbol group arrangement 2: source symbol interleaving

This arrangement is motivated by a scenario for which the source symbols are included in the transport object. However, not all transport objects include all source symbols, but a number of transport objects include a distinct set of source symbols. If all transport objects are accessible then reading from the beginning each transport objects, the initial data of the source object can be aggregated. This symbol group arrangement corresponds to an `mbpg_symbol_arrangement` value of 0010b.

This configuration supports for example that transport objects are generated with source symbols included. In addition, the following properties hold:

1) the transport object includes sufficient information to recover the included source object;

2) if only a subset of the transport objects is accessible, then an initial byte range of the subset of transport objects is sufficient to recover the source object, and the total number of bytes is small.

To abstract the procedure, again symbol groups are used. These symbol groups may then be included into transport objects, either completely or partially.

The only parameter for this arrangement is the number of symbol groups *C*.

For defining the symbol group arrangement, the following is defined:

- Total symbol count *TOTAL* added to symbol groups, initialized to 0.
- Next position in symbol group *c* that can add a symbol, *POS*[*c*], initialized to 0.
- Next symbol index in source block *z* that has not yet been added to any transport object, ESI[*z*], initialized to 0.
- Counter for number of symbols included in container *c* from source block *z*, *SYMBOLS*[*o*,*z*], initialized to 0

The arrangement as defined in clause 6.1.23.1.

Figure D.4 provides an example for source and repair symbol arrangement with *Kt=29*, *Z=3*, *Ks[0] = Ks[1]=10*, *Ks[2]=9*, *Kr[0]=Kr[1]=21 and Kr[2]=22* across *C=3* symbol groups. The bold ESI number shows the symbols that makes a source block complete for each symbol group. Symbols are added to each symbol group until all symbol groups are complete. Obviously more symbols until completion may be added.

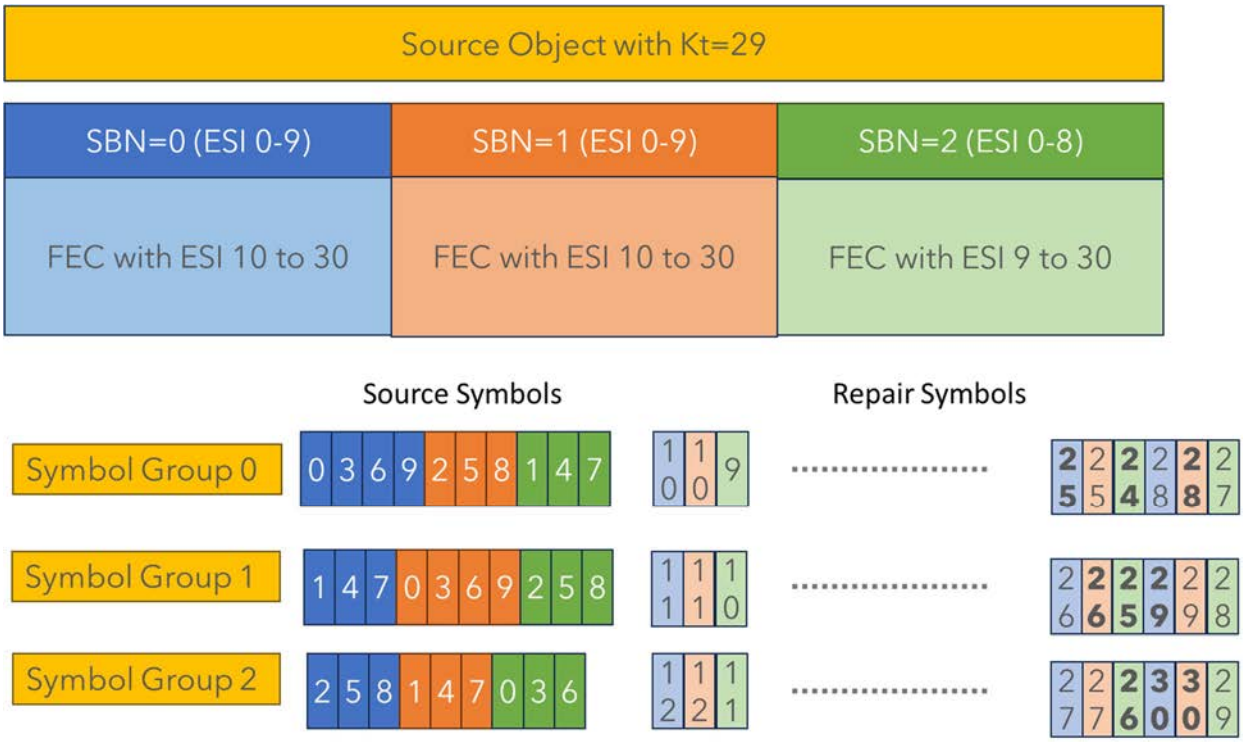

Figure D.4: Symbol arrangement for arrangement 2

D.2.5 Examples

D.2.5.1 Single file - source and partial encoding object

The following example is provided for a single file, where the file is an MP4 file compatible to some MIME type and in addition, for this file, three partial repair objects are provided that each contain 500 symbols of a single source block. The files are only partial and the information is provided in the extended FDT. The FDT is complete, i.e. no new information is provided.

```
<?xml version="1.0" encoding="UTF-8"?>
<FDTInstance xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns="urn:ETSI:CMMF:2023:FDT"
             xsi:schemaLocation="urn:ETSI:CMMF:2023:FDT extendedFDT.xsd"
             Expires="2024-05-30T09:30:10Z"
             Complete="true"
             ContentType="video/mp4 codecs='avc1.42c01e,mp4a.40.29' profiles='iso8'"
             FEC-OTI-FEC-Encoding-ID="6"
             FEC-OTI-Encoding-Symbol-Length="64">
    <File ContentLocation="https://example.com/efd1.mp4"
          TOI="0"
          Content-Length="64000">
        <EncodedObjects type="source"
                        complete="true">https://example.com/efd1.mp4</EncodedObjects>
        <EncodedObjects type="cmmf"
                        symbolArrangement="2"
                        sAParameters="3,0,0"
                        complete="true">https://example-cdn1.com/efd1-0.cmf</EncodedObjects>
        <EncodedObjects type="cmmf"
                        symbolArrangement="2"
                        sAParameters="3,1,0"
                        complete="true">https://example-cdn2.com/efd1-1.cmf</EncodedObjects>
        <EncodedObjects type="cmmf"
                        symbolArrangement="2"
                        sAParameters="3,2,0"
                        complete="true">https://example-cdn3.com/efd1-2.cmf</EncodedObjects>
    </File>
</FDTInstance>
```

D.2.5.2 Multiple files - self-contained objects including source symbols

In the following example, two files are provided, both encoded with RaptorQ and symbol length 64 bytes. In this case, only self-contained objects are provided, and these symbols are spread such that symbols 0, 3, 6 … are contained in one file, 1, 4, 7, … are contained in another file, and 2, 5, 8, …. are yet included in another file. At the same time, each encoded object is self-contained and can represent the files.

```
<?xml version="1.0" encoding="UTF-8"?>
<FDTInstance xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns="urn:ETSI:CMMF:2023:FDT"
             xsi:schemaLocation="urn:ETSI:CMMF:2023:FDT extendedFDT.xsd"
             Expires="2024-05-30T09:30:10Z"
             Complete="true"
             FEC-OTI-FEC-Encoding-ID="6"
             FEC-OTI-Encoding-Symbol-Length="64">
   <File ContentLocation="https://example.com/efd1-video.mp4"
        ContentType="video/mp4 codecs='avc1.42c01e' profiles='iso8'"
         TOI="0"
         Content-Length="64000">
        <EncodedObjects type="cmmf"
                        symbolArrangement="2"
                        sAParameters="3,0,0"
                        complete="true">https://example-cdn1.com/efd1-video-0.cmf</EncodedObjects>
        <EncodedObjects type="cmmf"
                        symbolArrangement="2"
                        sAParameters="3,1,0"
                        complete="true">https://example-cdn2.com/efd1-video-1.cmf</EncodedObjects>
        <EncodedObjects type="cmmf"
                        symbolArrangement="2"
                        sAParameters="3,2,0"
                        complete="true">https://example-cdn3.com/efd1-video-2.cmf</EncodedObjects>
   </File>
   <File ContentLocation="https://example.com/efd1-audio.mp4"
        ContentType="audio/mp4 codecs='mp4a.40.29' profiles='iso8'"
         TOI="1"
         Content-Length="4800">
        <EncodedObjects type="cmmf"
                        symbolArrangement="2"
                        sAParameters="3,0,0"
                        complete="true">https://example-cdn1.com/efd1-audio-0.cmf</EncodedObjects>
        <EncodedObjects type="cmmf"
                        symbolArrangement="2"
                        sAParameters="3,1,0"
                        complete="true">https://example-cdn2.com/efd1-audio-1.cmf</EncodedObjects>
        <EncodedObjects type="cmmf"
                        symbolArrangement="2"
                        sAParameters="3,2,0"
                        complete="true">https://example-cdn3.com/efd1-audio-2.cmf</EncodedObjects>
   </File>
</FDTInstance>
```

D.2.6 Potential receiver operation

Once a CMMF Receiver receives the Configuration Information in the form of an EFDT, it may access multiple coded versions of the same source object and operation can proceed as described in clause 4.3.6.A typical operation of the receiver is to collect symbols from different locations as quickly as possible in order to be able to recover the included objects while avoiding downloading unnecessary symbols.

The arrangements of symbols allow to minimize the network resource usage.

More details are for further study.

History

Document history		
V1.1.1	October 2024	Publication