OpenMAX™ Content Pipe Specification

Version 1.0

March 22, 2011

Table of Contents

OPENMAX™ CONTENT PIPE SPECIFICATION 1

TABLE OF CONTENTS 5

1 OVERVIEW 7

1.1 PURPOSE OF THIS DOCUMENT 7
1.2 ABOUT THE KHRONOS™ GROUP 7
1.3 VERSION NUMBER 7
1.4 HISTORY OF CONTENT PIPE DEVELOPMENT 7
1.5 BACKWARD COMPATIBILITY 8
1.6 ACKNOWLEDGEMENTS 8

2 CONTENT PIPE 10

2.1 RATIONALE 10
2.2 CONCEPT 10
2.3 IMPLEMENTATION 10
2.4 DEFINITION 12
2.4.1 Content Pipe versioning type 14
2.4.2 Content Pipe handle type 14
2.5 CONTENT ACCESS AND MANIPULATION 14
2.5.1 Get a Content Pipe 15
2.5.2 Content Pipe Methods 15
2.5.3 Positions 16
2.6 STREAMING SUPPORT 17
2.7 BASIC TYPES 18
2.8 ENUMERATIONS 19
2.8.1 CPA_RESULTTYPE 19
2.8.2 CPA_ORIGINTYPE 21
2.8.3 CPA_ACCESSTYPE 22
2.8.4 CPA_CHECKBYTERESULTTYPE 23
2.8.5 CPA_EVENTTYPE 24
2.9 TYPE DEFINITIONS 25
2.9.1 CPA_VERSIONTYPE 25
2.9.2 CPA_POSITIONINFOTYPE 25
2.9.3 CPA_CALLBACKTYPE 26
2.10 METHOD FOR ACQUIRING A CONTENT PIPE 27
2.10.1 CPA_GetContentPipe 27
2.11 CONTENT PIPE METHODS 27
2.11.1 CPA_GetApiVersion 27
2.11.2 CPA_ReleaseContentPipe 28
2.11.3 CPA_SetConfig 28
2.11.4 CPA_GetConfig 29
2.11.5 CPA_Open 29
2.11.6 CPA_Create 30
2.11.7 CPA_Close 30
2.11.8 CPA_CheckAvailableBytesToRead 31
2.11.9 CPA_CheckAvailableBytesToWrite 31
2.11.10 CPA_SetPosition 32
2.11.11 CPA_GetPositions 33
2.11.12 CPA_GetCurrentPosition 33

2.11.13 CPA_Read 34
2.11.14 CPA_ReadBuffer 35
2.11.15 CPA_ReleaseReadBuffer 36
2.11.16 CPA_Write 36
2.11.17 CPA_RegisterCallback 37

3 EXAMPLE USE CASES 38

3.1 OpenMAX IL Playback/Parser Use Case 38
3.2 OpenMAX IL Recording/Combiner Use Case 38
3.3 OpenMAX AL Playback Use Case 39
3.4 OpenMAX AL Record Use Case 39
3.5 OpenSL ES Playback Use Case 40
3.6 OpenSL ES Record Use Case 40

1 Overview

1.1 Purpose of this Document

This document details the Content Pipes API (CP) 1.0.0. Developed as an open standard by the Khronos Group, Content Pipes is a C-language interface for reading and writing content data, and could be used together with other Khronos API such as OpenMAX IL API or OpenMAX AL API.

1.2 About the Khronos™ Group

The Khronos Group is a member-funded industry consortium focused on the creation of open standard, royalty-free APIs to enable the authoring and accelerated playback of dynamic media on a wide variety of platforms and devices. All Khronos members can contribute to the development of Khronos API specifications, are empowered to vote at various stages before public deployment, and may accelerate the delivery of their multimedia platforms and applications through early access to specification drafts and conformance tests. The Khronos Group is also responsible for other open APIs such as OpenGL® ES, OpenKODE™, OpenSL ES™ and OpenVG™.

1.3 Version number

The version number of this document is of format a.b.c. It is divided into three parts:

Part	Name	Description
a	Major	An update of the Major number means a non-backwards compatible change in the interface. Not all functionality that was there in the previous version is guaranteed to work.
b	Minor	An update of the Minor number means a backwards-compatible change in the sense that the existing **clients** of the interface will continue working when the **implementation** of the interface is updated. Clients expecting new version of the implementation may or may not work with an older version of the implementation.
c	Step	An update of the Step number means no changes in the interface itself including its semantics, but instead the documentation has been improved.

1.4 History of content pipe development

Content pipe was originally specified in OpenMAX IL 1.x, and also in the OpenMAX AL draft specification. Content pipe was originally specified in OpenMAX IL 1.x, and also in the OpenMAX AL draft specification. To simplify the specification maintenance process, it was decided by IL and AL groups, for future releases of specification, to separate the content

pipe out of both IL specification and the draft AL specification and make it into one independent.

1.5 Backward Compatibility

There will be at least two versions of content pipe implementation, one according to OpenMAX IL 1.x specification that has no special version number for content pipe, and the other one according to this specification which has content pipe version number. In order to facilitate implementation migration, in this specification all type definition and methods are prefixed with "CPA_" to represent the "advanced" content pipe defined in this specification, to be distinguished from the legacy content pipe. Please note that this version of content pipe is not backward compatible with the legacy content pipe specified in OpenMAX IL 1.x spec.

The core method for retrieving the legacy content pipe defined in OpenMAX IL 1.x is defined as:

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_GetContentPipe (
      OMX_OUT OMX_HANDLETYPE  *hPipe,
      OMX_IN OMX_STRING szURI );
```

For the new advanced content pipe in this specification, a corresponding method is defined as:

```
CPA_RESULTTYPE CPA_GetContentPipe (CPA_OUT CPA_HANDLE *hPipe);
```

This method could be used together with OpenMAX IL and other APIs such as OpenMAX AL API. In OpenMAX IL usage, this method is placed at IL core.

1.6 Acknowledgements

Content Pipe Application Programming Interface Specification version 1.0.0 is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Chris Grigg, Beatnik

Roger Nixon, Broadcom

Timothy Granger, Broadcom

Frédéric Gabin, Ericsson

Håkan Gårdrup, Ericsson

Harald Gustafson, Ericsson

Marcus Lorentzon, Ericsson

Zhengrong Yao, Ericsson

Jean-Michel Trivi, Google

Erik Noreke, Independent

Graham Wacey, Imagination

Kevan Ahmadi, Imagination

Neeraj Agrawal, Imagination
Dave Murray, Incoras
Derek O'Gorman, Incoras
Adrian Burian, Nokia
Jarmo Hiipakka, Nokia
Juan Rubio, Nokia
Matti Paavola, Nokia
Ossi Kalevo, Nokia
Robert Palmer, Nokia
Yeshwant Muthusamy, Nokia
Acorn Pooley, Nvidia
Isaac Richards, Nvidia
Jim Van Welzen, Nvidia
Scott Peterson, Nvidia
Dusan Veselinovic, PacketVideo
Tom Longo, Qualcomm
Juan Rubio, Symbian
Robert Palmer, Symbian
Giulio Urlini, STMicroelectronics
Philippe Tribolo, ST-Ericsson
Pierre-Yves Taloud, ST-Ericsson
Sebastien Le Duc, ST-Ericsson
Thierry Vuillaume, ST-Ericsson
Gary Totney, Texas Instruments
Sripal Bagadia, Texas Instruments

2 Content Pipe

2.1 Rationale

Streaming media processing requires efficient data flow in and out of a media processing object.

For instance, in the playback use case a container format parser/demuxer typically pulls source data in a manner that assumes reads on a local file. Likewise, in the recording use case, a container format combiner/muxer typically pushes final data in a manner that assumes writes on a local file. Such “file access” is usually synchronous and includes some high frequency reads/writes of small size as well as random access.

In some cases, the content from which source data is pulled or to which final data is pushed is not local or is not from a file. The conventional approach to this use case, often referred to as “data streaming”, leverages queues of large input or output buffers of linear data transferred asynchronously. This approach is at odds with the "file access" model around which many parsers and combiners are designed. If conventional streaming is used then reconciling the two transfer models involves additional memory copies, waiting, and complexity.

2.2 Concept

We eliminate the inconsistency of these models by constructing a data access abstraction interface for pulling source data and pushing final data that lends itself to the needs of parsers and combiners. Rather than restricting ourselves to “file access” and the connotations it implies we use a more generalized notion of “content piping”.

A “content pipe” is an abstraction for any mechanism of accessing content data (i.e. pulling content data in or pushing content data out). This abstraction is not tied to any particular implementation. A pipe may be implemented, for example, as a local file, a remote file, a broadcast stream, memory buffers, intermediate data from derived from persistent data, etc. A pipe needn’t be limited to a single method of providing access. For instance a single pipe may provide via both local files and remote files, or through multiple transport protocols. A system may include one or many pipes.

2.3 Implementation

Since content pipe functions are synchronous, the implementation of the pipe interface is local even if the content itself is remote. This may entail a local agent acting as a broker between asynchronously pushed buffers from remote content and a pipe client (e.g. a parser) that must synchronously pull in data of varying sizes. Such an agent would maintain both the complex/elastic connection between the remote content and a local cache (which entails careful synchronization) as well as the simple/rigid connection between the local cache and the parser (which as a pull interface lacks complex synchronization).

Note that the synchronous pull based transfer implied by content pipe interface implies neither that the physical connection to the content nor the propagation of the data beyond the client be synchronous and pull-based. For example, consider the example of an OpenMAX IL parser component reading from either a remote file or a local one. The parser is provided the interface it requires, the mechanism to satisfy the pipe is completely abstracted and may actually use asynchronous data transfers, and the downstream data

transfer is completely unaffected. This is shown in Figure 1. In this document, the IL components that directly connect to the content pipe (e.g., the parser and muxer in Figure 1) are also referred to as content pipe users. Figure 2 shows an example of content pipe usage in OpenMAX AL and OpenSL ES (they share the same object model). Here, the content pipe serves as either the data source (for reads) or data sink (for writes) of a media object. The media object utilizes the methods of the content pipe interface to read or write data.

Section 2.5 contains detailed exemplar use cases of content pipes in OpenMAX IL, OpenMAX AL and OpenSL ES.

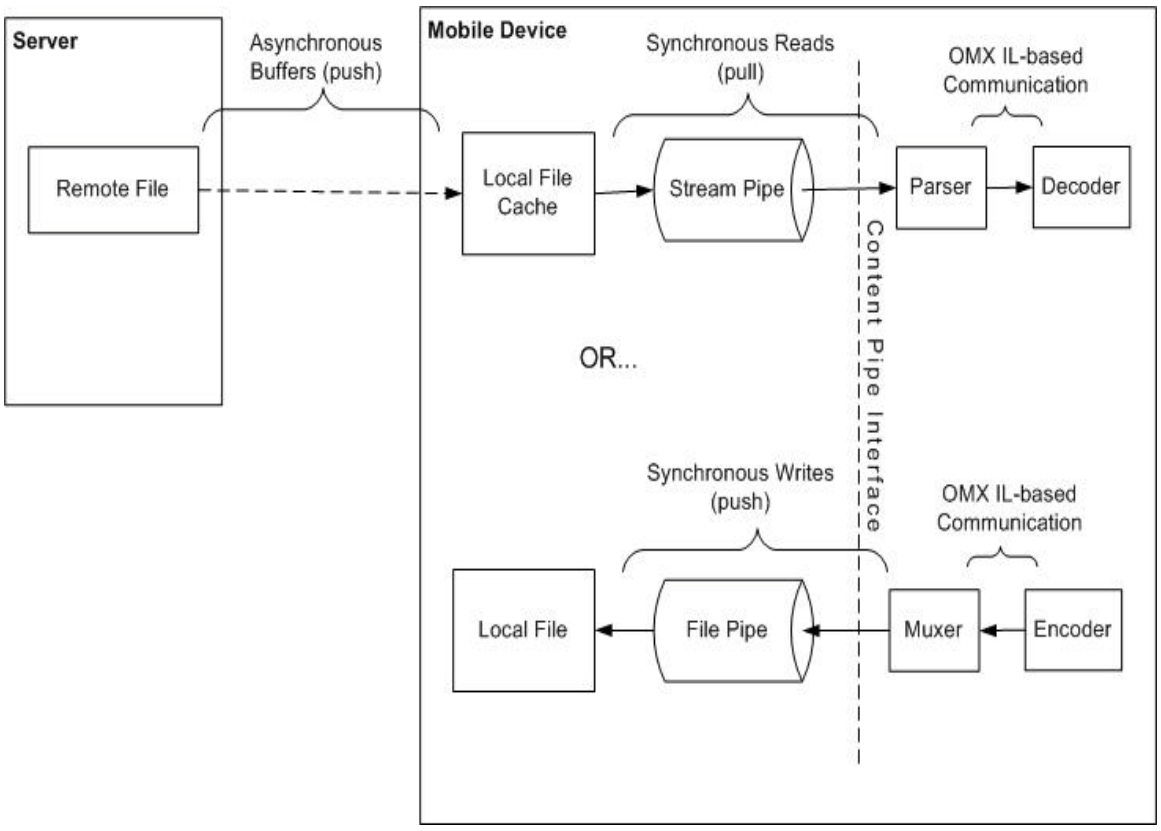

Figure 1: Content Pipe Operation Example - OpenMAX IL

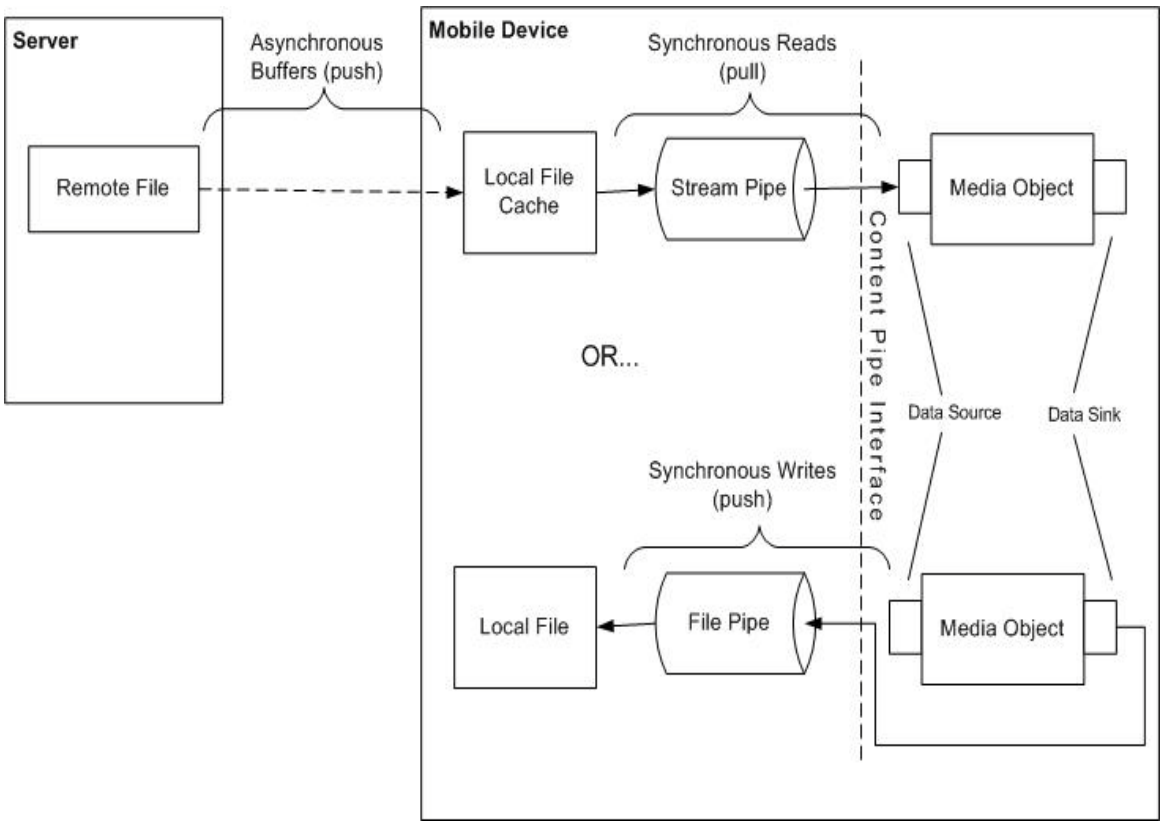

Figure 2: Content Pipe Operation Example - OpenMAX AL/OpenSL ES

2.4 Definition

The content pipe interface structure is defined as:

```
typedef struct CPA_PIPETYPE
{
CPA_VERSIONTYPE nApiVersion;

CPA_RESULTTYPE (*CPA_ReleaseContentPipe)(
      CPA_INOUT CPA_HANDLE* hPipe );

CPA_RESULTTYPE (*CPA_SetConfig)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_STRING szKey,
      CPA_IN CPA_PTR value);

CPA_RESULTTYPE (*CPA_GetConfig)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_STRING szKey,
      CPA_OUT CPA_PTR value);
```

```
CPA_RESULTTYPE (*CPA_Open)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_OUT CPA_HANDLE* hContent,
      CPA_IN CPA_STRING szURI,
      CPA_IN CPA_ACCESSTYPE eAccess);

CPA_RESULTTYPE (*CPA_Create)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_OUT CPA_HANDLE * hContent,
      CPA_IN CPA_STRING szURI);

CPA_RESULTTYPE (*CPA_Close)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_INOUT CPA_HANDLE* hContent);

CPA_RESULTTYPE (*CPA_CheckAvailableBytesToRead)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_U32 nBytesRequested,
      CPA_OUT CPA_CHECKBYTESRESULTTYPE* peResult);

CPA_RESULTTYPE (*CPA_CheckAvailableBytesToWrite)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_U32 nBytesRequested,
      CPA_OUT CPA_CHECKBYTESRESULTTYPE* peResult);

CPA_RESULTTYPE (*CPA_SetPosition)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_POSITIONTYPE nOffset,
      CPA_IN CPA_ORIGINTYPE eOrigin);

CPA_RESULTTYPE (*CPA_GetPosition)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_OUT CPA_POSITIONINFOTYPE* pPosition);

CPA_RESULTTYPE (*CPA_GetCurrentPosition)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_OUT CPA_POSITIONINFOTYPE* pPosition);

CPA_RESULTTYPE (*CPA_Read)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_OUT CPA_BYTE* pData,
      CPA_INOUT CPA_U32* pSize);

CPA_RESULTTYPE (*CPA_ReadBuffer)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_OUT CPA_BYTE** ppBuffer,
      CPA_INOUT CPA_U32* pSize,
      CPA_IN CPA_BOOL bForbidCopy);
```

```
CPA_RESULTTYPE (*CPA_ReleaseReadBuffer)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_BYTE* pBuffer);

CPA_RESULTTYPE (*CPA_Write)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_BYTE *pData,
      CPA_INOUT CPA_U32* pSize);

CPA_RESULTTYPE (*CPA_RegisterCallback)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_CALLBACKTYPE ClientCallback,
      CPA_IN CPA_PTR ClientContext);

} CPA_PIPETYPE;
```

The access macros for content pipe API version is defined as follow:

```
#define CP_GetApiVersion( hPipe ) \
    ((CP_PIPETYPE*)hPipe)->nApiVersion
```

2.4.1 Content Pipe versioning type

The first field of the content pipe type class contains version information.

`CPA_VERSIONTYPE nApiVersion;`

2.4.2 Content Pipe handle type

A content pipe handle type is defined to identify different classes of objects used in content pipe implementation. One handle could refers to one instance of content pipe interface, another handle might refer to one opened content (a file, a stream).

```
typedef void* CPA_HANDLE;
```

2.5 Content Access and Manipulation

Access to content pipe functionality can be conceptually divided into two layers.

The method `CPA_GetContentPipe` is used to acquire an instance of a content pipe interface that can be created and destroyed. Each instance of content pipe interface is identified by one handle.

The acquired instance of content pipe interface (with one handle) provides methods for accessing and manipulating the actual content data (a file, a stream), such as opening, creating and closing actual content, the actual content is identified by one content handle.

In such a manner, one instance of content pipe interface may open and close different content streams during its life cycle. However, only one content stream can be active at any given point in time.

2.5.1 Get a Content Pipe

The `CPA_GetContentPipe` method is used to acquire one instance of content pipe interface.

```
CPA_RESULTTYPE CPA_GetContentPipe (CPA_OUT CPA_HANDLE *hPipe);
```

2.5.2 Content Pipe Methods

A content pipe has the following methods.

```
CPA_VERSIONTYPE CPA_GetApiVersion(
      CPA_IN hPipe );

CPA_RESULTTYPE (*CPA_ReleaseContentPipe)(
      CPA_INOUT CPA_HANDLE* hPipe );

CPA_RESULTTYPE (*CPA_SetConfig)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_STRING szKey,
      CPA_IN CPA_PTR value);

CPA_RESULTTYPE (*CPA_GetConfig)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_STRING szKey,
      CPA_OUT CPA_PTR value);

CPA_RESULTTYPE (*CPA_Open)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_OUT CPA_HANDLE* hContent,
      CPA_IN  CPA_STRING szURI,
      CPA_IN  CPA_ACCESSTYPE eAccess);

CPA_RESULTTYPE (*CPA_Create)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_OUT CPA_HANDLE* hContent,
      CPA_IN  CPA_STRING szURI);

CPA_RESULTTYPE (*CPA_Close)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_INOUT CPA_HANDLE* hContent);

CPA_RESULTTYPE (*CPA_CheckAvailableBytesToRead)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_U32 nBytesRequested,
      CPA_OUT CPA_CHECKBYTERESULTTYPE* peResult);

CPA_RESULTTYPE (*CPA_CheckAvailableBytesToWrite)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_U32 nBytesRequested,
      CPA_OUT CPA_CHECKBYTERESULTTYPE* peResult);

CPA_RESULTTYPE (*CPA_SetPosition)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_POSITIONTYPE nOffset,
      CPA_IN CPA_ORIGINTYPE eOrigin );
```

```
CPA_RESULTTYPE (*CPA_GetPosition)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_OUT CPA_POSITIONINFOTYPE* pPosition);

CPA_RESULTTYPE (*CPA_GetCurrentPosition)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_OUT CPA_POSITIONINFOTYPE* pPosition);

CPA_RESULTTYPE (*CPA_Read)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_OUT CPA_BYTE* pData,
      CPA_INOUT CPA_U32* pSize);

CPA_RESULTTYPE (*CPA_ReadBuffer)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_OUT CPA_BYTE** ppBuffer,
      CPA_INOUT CPA_U32* pSize,
      CPA_IN CPA_BOOL bForbidCopy);

CPA_RESULTTYPE (*CPA_ReleaseReadBuffer)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_BYTE* pBuffer);

CPA_RESULTTYPE (*CPA_Write)
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_BYTE* pData,
      CPA_INOUT CPA_U32* pSiz );

CPA_RESULTTYPE (*CPA_RegisterCallback)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_CALLBACKTYPE ClientCallback,
      CPA_IN CPA_PTR ClientContext);
```

Because content parsers and muxers operate as though they are accessing files directly, a pipe's data access functions are modeled on conventional file access. These include functions for reading and writing data using client buffers and setting/retrieving the read/write position within the content. A content pipe instance provides methods for file handling.

2.5.3 Positions

All positions are expressed as CPA_POSITIONTYPE values as shown in the following structure.

```
#define CPA_POSITION_NA -1
#ifdef CPA_64BITSSUPPORTED
# define CPA_POSITION_32_MAX 0x000000007FFFFFFF
typedef CPA_S64 CPA_POSITIONTYPE;
#else
# define CPA_POSITION_32_MAX 0x7FFFFFFF
typedef CPA_S32 CPA_POSITIONTYPE;
#endif
```

Platforms that can handle 64 bit arithmetic may set the flag CPA_64BITSUPPORTED. Both the content pipe and the content pipe user must be compiled with the same flag setting..

Implementation on a platform that does handle 64 bit arithmetic (and is compiled with the flag CPA_64BITSUPPORTED set) may still only handle positions of 32 bits. This is for example the case if the standard C library function ftell is used. This implementation may convert the signed 64-bit value to a signed 32-bit value internally but risk loss of precision. The defined values CPA_POSITION_32_MAX may be used together with the result code CPA_OKPOSITIONEXCEED2GB to indicate this loss of precision by the content pipe (see CPA_GetPosition).

A position equaling to CPA_POSITION_NA indicates that the requested position could not be determined or is not applicable.

2.6 Streaming Support

In streaming content pipe use case, the source content may be remotely located and streamed during processing to a position of local accessibility (e.g. a local cache of remote content). A set of functions is defined to accommodate such scenarios.

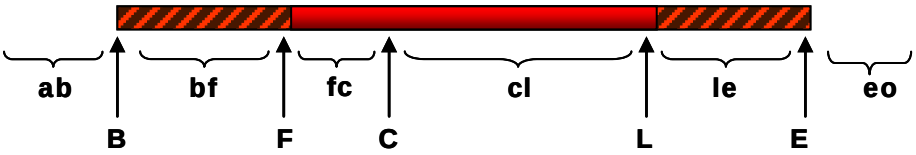

Figure 3: Visualization of a streaming data stream. The striped red area is not accessible because it is not kept or not yet downloaded.

The CPA_GetPositions() method reports five values. The values match different key positions marked out in Figure 2. The internal relations between the different positions in a content pipe implementation must follow the rules B(Begin) ≤ F(First) ≤ C(Current) ≤ L(Last) ≤ E(End). Thus position F may never pass the current read or write position C. The CPA_GetCurrentPosition() method is used to report only the current position. The CheckAvailableBytesToRead() method queries if a given number of bytes are available for reading and the CheckAvailableBytesToWrite() method queries if a given number of bytes are available for writing. This allows the client to check for the availability of enough bytes to satisfy a large section of parsing prior to beginning the parsing. This allows a pipe implementation to stream data to a local cache.

```
CPA_RESULTTYPE (*CPA_CheckAvailableBytesToRead)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_U32 nBytesRequested,
      CPA_OUT CPA_CHECKBYTERESULTTYPE* peResult);


CPA_RESULTTYPE (*CPA_CheckAvailableBytesToWrite)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_U32 nBytesRequested,
      CPA_OUT CPA_CHECKBYTERESULTTYPE* peResult);
```

If the data or space is not immediately available the pipe will call the client via the provided callback when it is. Note that the content pipe will only hold one outstanding callback of each type (BytesToRead and BytesToWrite). A second request for any of the type will cancel

any previously trigged callbacks of that type independent of whether this call trigged a new callback or not. This callback mechanism also includes events for data overflow and a pipe disconnection (e.g. if the connection with a remote source is lost). See the CPA_EVENTTYPE enumeration for details. The callback is registered via the RegisterCallback method:

```
CPA_RESULTTYPE (*CPA_RegisterCallback)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_CALLBACKTYPE ClientCallback,
      CPA_IN CPA_PTR ClientContext);
```

The ReadBuffer method reads a large area of data using the pipe implementation's memory. If a pipe implementation is streaming remote data to a local cache the desired data will already reside in local memory prior to a call on this method. This method avoids the memory copy that would be required if the client provided the memory pointer. Instead, in this method the pipe implementation shall provide the memory pointer ppBuffer.

```
CPA_RESULTTYPE (*CPA_ReadBuffer)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_OUT CPA_BYTE** ppBuffer,
      CPA_INOUT CPA_U32* pSize,
      CPA_IN CPA_BOOL bForbidCopy);
```

This necessitates a ReleaseReadBuffer() method to release a buffer acquired via ReadBuffer

```
CPA_RESULTTYPE (*CPA_ReleaseReadBuffer)(
      CPA_IN CPA_HANDLE hPipe,
      CPA_IN CPA_HANDLE hContent,
      CPA_IN CPA_BYTE* pBuffer);
```

2.7 Basic types

The basic types are defined here:

```
typedef void* CPA_PTR;
typedef char* CPA_STRING;
typedef unsigned char CPA_BYTE;
typedef void* CPA_HANDLE;
typedef unsigned char CPA_U8;
typedef signed char CPA_S8;
typedef unsigned short CPA_U16;
typedef signed short CPA_S16;
typedef unsigned long CPA_U32;
typedef signed long CPA_S32;
```

A short description of each basic type is given in the following table:

Basic Type	Description
CPA_PTR	A void pointer

Basic Type	Description
CPA_STRING	The CPA_STRING type is intended to be used to pass “C” type strings between the Content pipe Client and the Content pipe implementation. The CPA_STRING type is a 8 bit pointer to a zero terminated string. CPA_STRING contains text in UTF-8 format.
CPA_BYTE	The CPA_BYTE type is intended to be used to pass arrays of bytes to and from the content pipe. The CPA_BYTE type is a 8 bit unsigned data field.
CPA_HANDLE	Define the public interface for the content pipe Handle.
CPA_U8	CPA_U8 is an 8 bit unsigned quantity.
CPA_S8	CPA_S8 is an 8 bit signed quantity.
CPA_S16	CPA_S16 is a 16 bit signed quantity.
CPA_U32	CPA_U32 is a 32 bit unsigned quantity.
CPA_S32	CPA_S32 is a 32 bit signed quantity.
CPA_BOOL	Represents a true (CPA_TRUE) or false (CPA_FALSE) value.

For system that support 64 bits arithmetic, the CPA_U64 type and CPA_S64 could be defined accordingly for unsigned and signed quantity.

2.8 Enumerations

2.8.1 CPA_RESULTTYPE

The CPA_RESULTTYPE enumeration defines result codes form content pipe methods and is defined as follow:

```
typedef enum CPA_RESULTTYPE{
      CPA_OK,
      CPA_OKEOS,
      CPA_OKPOSITIONEXCEED2GB,
      CPA_EPOSNOTAVAIL,
      CPA_EUNKNOWN,
      CPA_EACCESS,
      CPA_EAGAIN,
      CPA_EALREADY,
      CPA_EBUSY,
      CPA_ECONNREFUSED,
      CPA_ECONNRESET,
      CPA_EEXIST,
      CPA_EFBIG,
      CPA_EINVAL,
      CPA_EIO,
      CPA_ENOENT,
      CPA_EURINOTSUPP,
      CPA_ENOMEM,
      CPA_ENOSPC,
      CPA_ENO_RECOVERY,
      CPA_EOPNOTSUPP,
      CPA_ETIMEDOUT,
      CPA_EVERSION
} CPA_RESULTTYPE;
```

Result codes may contain both errors and success messages. If the result code indicates an error, other outputs may be invalid. The description of each value is given in Table 2-1:

Table 2-1: Content Pipe result code

Value	Description
CPA_OK	The operation was successful
CPA_OKEOS	End of stream reached. EOS must be returned when more bytes are requested that what is available in the stream. The available bytes must be returned.
CPA_OKPOSITIONEXCEED2GB	Current position exceeds 2 GB and that is not handled by the current implementation.
CPA_EPOSNOTAVAIL	Position not available in the specified content stream.
CPA_EUNKNOWN	Unknown error.
CPA_EACCESS	Operation denied due to violating the content access mode
CPA_EAGAIN	Resource unavailable at the moment but content pipe user can wait for a while and try again.
CPA_EALREADY	Address already in use, or a connection attempt is already in progress for this address.

Value	Description
CPA_EBUSY	Device or resource busy and can not fulfill the request.
CPA_ECONNREFUSED	Connection refused by the content host.
CPA_ECONNRESET	The connection has been reset by the content host. This usually means that the content host program has crashed, or closed the socket unexpectedly.
CPA_EEXIST	The file a content pipe user want to create already exists.
CPA_EFBIG	File too large. The current content pipe implementation might have issue handling it.
CPA_EINVAL	Invalid argument.
CPA_EIO	I/O error.
CPA_ENOENT	The specified address is not available.
CPA_EURINOTSUPP	The URI is not supported with this content pipe.
CPA_ENOMEM	Out of memory.
CPA_ENOSPC	No space left on destination device or address.
CPA_ENO_RECOVERY	A non-recoverable error has occurred.
CPA_EOPNOTSUPP	Operation not supported.
CPA_ETIMEDOUT	Connection to content host has timed out.
CPA_EVERSION	A version error found

2.8.2 CPA_ORIGINTYPE

The CPA_ORIGINTYPE enumeration defines all the origin types used by the CPA_SetPosition method of the CPA_PIPETYPE from which the indicated position is relative.

```
typedef enum CPA_ORIGINTYPE {
      CPA_OriginBegin,
      CPA_OriginFirst,
      CPA_OriginCur,
      CPA_OriginLast,
      CPA_OriginEnd
} CPA_ORIGINTYPE;
```

The description of each value is given in Table 2-2:

Table 2-2: Content Pipe Origin Types

Value	Description. Opened for Read or Write
CPA_OriginBegin	Read/Write: Origin is the beginning of content, specifically the first byte of the content's data stream.
CPA_OriginFirst	Read: Origin is the beginning of available content, specifically the first still available byte of the content's data stream. Write: Not applicable
CPA_OriginCur	Read/Write: Origin is the current position within the content.
CPA_OriginLast	Read: Origin is the end of the available content, specifically the position of the last available byte of the content's data stream. Write: Not applicable
CPA_OriginEnd	Read: Origin is the end of content, specifically the last byte of the content's data stream. Write: Not applicable

2.8.3 CPA_ACCESSTYPE

The CPA_ACCESSTYPE enumeration defines all the access types used by the CPA_Open method of the CPA_PIPETYPE and is defined as:

```
typedef enum CPA_ACCESSTYPE {
      CPA_AccessRead,
      CPA_AccessWrite,
      CPA_AccessReadWrite
} CPA_ACCESSTYPE;
```

The description of each value is given in Table 2-3:

Table 2-3: Content Pipe Access Types

Value	Description
CPA_AccessRead	Access type is read only.
CPA_AccessWrite	Access type is write only.
CPA_AccessReadWrite	Access type is both read and write.

2.8.4 CPA_CHECKBYTERESULTTYPE

The CPA_CHECKBYTERESULTTYPE enumeration defines all possible results of a call to the CPA_CheckAvailableBytesToRead() and CPA_CheckAvailableBytesToWrite() method of the CPA_PIPETYPE. The CPA_CHECKBYTERESULTTYPE enumeration is defined as:

```
typedef enum CPA_CHECKBYTESRESULTTYPE{
      CPA_CheckBytesOk,
      CPA_CheckBytesNotReady,
      CPA_CheckBytesInsufficientBytes,
      CPA_CheckBytesTooLargeRequest,
} CPA_CHECKBYTESRESULTTYPE;
```

The description of each value is given in Table 2-4:

Table 2-4: Result Types for CheckAvailableBytesToRead() and CheckAvailableBytesToWrite()

Value	Description
CPA_CheckBytesOk	Read: There is at least the requested number of bytes available. No callback is triggered. Write: There is space for at least the request number of bytes available. No callback is triggered.
CPA_CheckBytesNotReady	Read: The pipe is still retrieving bytes and presently lacks sufficient bytes. Client will be called when sufficient bytes are available. Callback is triggered and previously trigged read callbacks are canceled. Write: The pipe is still processing data and presently lacks sufficient space. Client will be called when they are sufficient space is available. Callback is triggered and previously trigged write callbacks are canceled.
CPA_CheckBytesInsufficientBytes	Read: The pipe has reached the end of stream and the available bytes are less than those requested. There may still be some data in the pipe. No callback is triggered. Write: The pipes current session lacks sufficient space to store the requested amount of data. This may depend on full storage space or a lost connection. There may still be some data in the pipe that is being processed. No callback is triggered.

Value	Description
CPA_CheckBytesTooLargeRequest	Read: The pipe can never reach the requested number of bytes due to insufficient pre buffer space in the pipe. No callback is triggered. Write: The pipe can never receive a data delivery of the requested size due to insufficient pre buffer space in the pipe. No callback is triggered.

2.8.5 CPA_EVENTTYPE

The CPA_EVENTTYPE enumeration defines events a content pipe may send to its user via a registered client callback method, the registration is done by the CPA_RegisterCallback() method. The CPA_EVENTTYPE enumeration is defined as:

```
typedef enum CPA_EVENTTYPE{
      CPA_EventBytesToReadAvailable,
      CPA_EventBytesToWriteAvailable,
      CPA_EventPipeDisconnected,
      CPA_EventEndOfStream,
} CPA_EVENTTYPE;
```

The description of each value is given in Table 2-5:

Table 2-5: Content Pipe Event Types

Value	Description
CPA_EventBytesToReadAvailable	Bytes requested in the latest CPA_CheckAvailableBytesToRead() call which were formally unavailable are now available. The iParam parameter of the callback contains the number of bytes currently available.
CPA_EventBytesToWriteAvailable	Bytes requested in the latest CPA_heckAvailableBytesToWrite() call which were formally unavailable are now available. The iParam parameter of the callback contains the number of free bytes currently available.
CPA_EventPipeDisconnected	The pipe been disconnected. The iParam parameter of the callback is unused. Trigged CPA_EventBytesToWriteAvailable callbacks have been canceled.
CPA_EventEndOfStream	EndOfStream has been reached and trigged CPA_EventDataAvailable callbacks have been canceled.

2.9 Type definitions

2.9.1 CPA_VERSIONTYPE

The CPA_VERSIONTYPE type indicates the version of a content pipe API and could be queried by content pipe user via a CPA_GetApiVersion( ) method. Note that this is not a versioning of a specific content pipe type's functionality. CPA_VERSIONTYPE is defined as follows:

```
typedef union CPA_VERSIONTYPE
{
    struct
    {
      CPA_U8 nVersionMajor;
      CPA_U8 nVersionMinor;
      CPA_U8 nRevision;
    };
    CPA_U32 nVersionID;
} CPA_VERSIONTYPE;
```

The description of each value is given in Table 2-6:

Table 2-6: Content Pipe Version Types

Type	Value	Description
CPA_U8	nVersionMajor	Major part of version field.
CPA_U8	nVersionMinor	Minor part of version field.
CPA_U8	nRevision	Revision
CPA_U32	nVersionID	Unique ID for the current version.

2.9.2 CPA_POSITIONINFOTYPE

The CPA_POSITIONINFOTYPE describes the current position, the content size and the window for the available content. This type is constructed by a structure with the following members.

```
typedef struct CPA_POSITIONINFOTYPE {
      CPA_POSITIONTYPE nDataBegin;
      CPA_POSITIONTYPE nDataFirst;
      CPA_POSITIONTYPE nDataCur;
      CPA_POSITIONTYPE nDataLast;
      CPA_POSITIONTYPE nDataEnd;
} CPA_POSITIONINFOTYPE;
```

By subtracting nDataFirst from nDataLast, the available number of bytes may be obtained when content pipe is opened for reading. The positions are illustrated in Figure 2.

The description of each letter is given in Table 2-7:

Table 2-7: Position Types

Letter	Type	Value	Description. Opened for Read or Write
B	CPA_POSITIONTYPE	nDataBegin	Read: The beginning of content, specifically the position of the first byte of the content's data stream. Typically zero. Write: The beginning of content, specifically the position of the first writable byte of the content's data stream. Typically zero.
F	CPA_POSITIONTYPE	nDataFirst	Read: The beginning of the available content, specifically the position of the first (still) available byte of the content's data stream. Typically zero for file reading. Write: Nor applicable
C	CPA_POSITIONTYPE	nDataCur	Read/Write: Current position
L	CPA_POSITIONTYPE	nDataLast	Read: The end of the available content, specifically the position of the last available byte of the content's data stream. Typically equaling nDataEnd for file reading. Write: Not applicable
E	CPA_POSITIONTYPE	nDataEnd	Read: The end of content, specifically the position of the last byte of the content's data stream. This may return CPA_POSITION_NA if the position is unknown. Typically equaling the file size for file reading. Write: Not applicable

2.9.3 CPA_CALLBACKTYPE

The CPA_CALLBACKTYPE describes the callback function signature used for callbacks. The context in where the content pipe callback function is executed in is unknown to the content pipe user. Note that CPA_RegisterCallback() is intended for the content pipe user.

```
typedef CPA_RESULTTYPE (*CPA_CALLBACKTYPE)(
      CPA_HANDLE hPipe,
      CPA_EVENTTYPE eEvent,
      CPA_PTR ClientContext) CPA_RESULTTYPE;
```

2.10 Method for acquiring a content pipe

2.10.1 CPA_GetContentPipe

Before any content pipe function could be used, as one starting step one has to acquire one instance of content pipe interface first. The CPA_GetContentPipe() method is used at start to retrieve an instance of content pipe interface. In order to access the content, the full URI information of content are needed when the instance of content pipe interface calls the CPA_Open() and CPA_Create()methods.

In order to deal with different types of URIs, several content pipes implementation might exist on the system. It is the integrator's responsibility to mask the presence of these multiple implementations to the content pipe user, and present a single coherent interface.

After acquiring the instance of content pipe interface, the content pipe user may be able to use CPA_GetConfig() or CPA_SetConfig() methods with a NULL value in hContent parameter for accessing content pipe configuration prior to providing the URI.

The content pipe implementation shall allocate the memory for the instance of content pipe interface.

The result code CPA_EURINOTSUPP must be expected.

```
CPA_RESULTTYPE CPA_GetContentPipe(
      CPA_OUT CPA_HANDLE* hPipe);
```

This is a blocking call. The description of each parameter is given in Table 2-9:

Table 2-8: CPA_GetContentPipe() Parameters

The parameters are as follows.

Parameter	Description
hPipe *[out]*	Handle to the requested instance of content pipe interface. The content pipe implementation shall allocate memory for this instance.

2.11 Content Pipe methods

After the creation of one instance of content pipe interface, the following methods could be used on that instance.

2.11.1 CPA_GetApiVersion

Returns the API version that the current content pipe implements.

```
CPA_VERSIONTYPE CPA_GetApiVersion(
      CPA_HANDLE hPipe);
```

This is a blocking call. The description of each parameter is given in as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.
return *[out]*	Content Pipe API version

2.11.2 CPA_ReleaseContentPipe

The `CPA_ReleaseContentPipe()` method releases all resources allocated when the content pipe was obtained by the content pipe user. It is up the content pipe user to make sure that no content streams of the specified type are still open when releasing the content pipe interface. If the content pipe user still has any open content streams, this call may either fail, close opened resources or proceed the release making all opened content streams inaccessible.

```
CPA_RESULTTYPE (*CPA_ReleaseContentPipe)(
      CPA_HANDLE* hPipe );
```

This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance that has to be released. The handle shall be set to NULL after a successful method call.

2.11.3 CPA_SetConfig

The `CPA_SetConfig()` method is used to set the configuration of the acquired content pipe interface by specifying the key value. The streams of one content pipe interface share the same configuration. The content pipe user must allocate the value memory.

```
CPA_RESULTTYPE (*CPA_SetConfig)(
      CPA_HANDLE hPipe,
      CPA_HANDLE hContent,
      CPA_STRING szKey,
      CPA_PTR value);
```

This is a blocking call. The description of each parameter is given as follows. This method allows Null value in `hContent` parameter.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.

Parameter	Description
hContent *[in]*	Handle of content stream.
szKey *[in]*	A string of parameter name for content pipe configuration.
value *[in]*	Pointer to a configuration parameter value. The memory pointed to is only valid during the call.

2.11.4 CPA_GetConfig

The CPA_GetConfig() method is used to get the content pipe interface's configuration via querying the key value. The streams of one content pipe interface share the same configuration. The content pipe user must allocate the value memory. This method allows Null value in hContent parameter.

```
CPA_RESULTTYPE (*CPA_GetConfig)(
      CPA_HANDLE hPipe,
      CPA_HANDLE hContent,
      CPA_STRING szKey,
      CPA_PTR value);
```

This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.
hContent *[in]*	Handle of content stream.
szKey *[in]*	A string of parameter name for content pipe configuration.
value *[out]*	Pointer to a configuration parameter value. The memory pointed to is only valid during the call

2.11.5 CPA_Open

The CPA_Open() method opens the specified content stream with the specified access type. i.e. open one existing file for reading.

```
CPA_RESULTTYPE (*CPA_Open) (
      CPA_HANDLE hPipe,
      CPA_HANDLE* hContent,
      CPA_STRING szURI,
      CPA_ACCESSTYPE eAccess);
```

This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance that owns the opened content stream.
hContent *[out]*	Pointer receiving the new content handle corresponding to the specified URI opened with the specified access type. The content pipe implementation shall allocate the memory for this pointer.
szURI *[in]*	URI specifying the location of the content stream.
eAccess *[in]*	Desired access to the content.

2.11.6 CPA_Create

The `CPA_Create()` method creates the specified content stream for writing and returns a handle to it. i.e. create one new file for writing.

```
CPA_RESULTTYPE (*CPA_Create)(
      CPA_HANDLE hPipe,
      CPA_HANDLE* hContent,
      CPA_STRING szURI);
```

This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance that owns the created content stream.
hContent *[out]*	Pointer receiving the new content handle corresponding to the specified URI created for writing. The content pipe implementation shall allocate the memory for this pointer.
szURI *[in]*	URI specifying the desired location of the content stream.

2.11.7 CPA_Close

The `CPA_Close()` method closes the specified content pipe data stream handle and sets the `hContent` pointer to NULL.

```
CPA_RESULTTYPE (*CPA_Close)(
      CPA_HANDLE hPipe,
      CPA_HANDLE* hContent);
```

This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.
hContent *[in]*	Handle of the content stream to be closed. The handle shall be set to NULL after a successive call.

2.11.8 CPA_CheckAvailableBytesToRead

The CPA_CheckAvailableBytesToRead() method verifies that the specified number of bytes is available for reading from the current position in the stream. (Specifically the size of area **cl** in Figure 2.) If one content pipe implementation supports the CPA_Read() method or CPA_ReadBuffer() method, it shall also support this method.

```
CPA_RESULTTYPE (*CPA_CheckAvailableBytesToRead) (
      CPA_HANDLE hPipe,
      CPA_HANDLE hContent,
      CPA_U32 nBytesRequested,
      CPA_CHECKBYTESRESULTTYPE * peResult);
```

If current stream is opened in write mode, CPA_EACCES may be returned. This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.
hContent *[in]*	Handle of content stream to check.
nBytesRequested [in]	The desired number of bytes.
peResult *[out]*	Result of check (see definition of CPA_CHECKBYTESRESULTTYPE).

2.11.9 CPA_CheckAvailableBytesToWrite

The CPA_CheckAvailableBytesToWrite() method verifies that space for the specified number of bytes is available for writing from current position in the stream. If one content pipe implementation support CPA_Write() method, it shall also support this method.

```
CPA_RESULTTYPE (*CPA_CheckAvailableBytesToWrite) (
      CPA_HANDLE hPipe,
      CPA_HANDLE hContent,
      CPA_U32 nBytesRequested,
      CPA_CHECKBYTERESULTTYPE * peResult);
```

If current stream is opened in read mode, CPA_EACCES may be returned. This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.
hContent *[in]*	Handle of content stream to check.
nBytesRequested [in]	The desired space in bytes.
peResult *[out]*	Result of check (see definition of CPA_CHECKBYTERESULTTYPE).

2.11.10 CPA_SetPosition

The CPA_SetPosition() method moves the pipe's byte position within a piece of content to the specified location.

```
CPA_RESULTTYPE (*CPA_SetPosition)(
      CPA_HANDLE hPipe,
      CPA_HANDLE hContent,
      CPA_POSITIONTYPE nOffset,
      CPA_ORIGINTYPE eOrigin);
```

SetPosition is expected leave the following result codes depending on the specified position if no other error occurs:

Area in Figure 2	Expected result code
Ab	CPA_EPOSNOTAVAIL
Bf	CPA_OK or CPA_EPOSNOTAVAIL
Fc	CPA_OK
Cl	CPA_OK
Le	CPA_OK
Eo	CPA_OK (If the end position of the file is unknown, there is no way for a SetPosition to tell the difference between area le and eo, thus the same behavior and the same result code is expected.)

This is a blocking call. The returning from this method does not necessarily imply that data from the new position is immediately available. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.
hContent *[in]*	Handle of the content stream.
nOffset *[in]*	Offset of desired byte position relative to the specified origin.
eOrigin *[in]*	Origin from relative to which the offset applies.

2.11.11 CPA_GetPositions

The CPA_GetPositions() method returns the pipe's byte position within a piece of content.

```
CPA_RESULTTYPE (*CPA_GetPositions)(
      CPA_HANDLE hPipe,
      CPA_HANDLE hContent,
      CPA_POSITIONINFOTYPE* pPosition);
```

The implementation is not mandated to support 64 bit position specifier for GetPosition even if the flag CPA_64BITSUPPORTED is set. When any position in pPosition exceeds 2^31 and if 64 bit file handling is not supported, the implementation shall return CPA_OKPOSITIONEXCEED2GB and place CPA_POSITION_32_MAX in one or more fields in pPosition.

This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.
hContent *[in]*	Handle of the content stream.
pPosition *[out]*	Information about the current byte position of the pipe within the specified content, the content size and the window for the available content.

2.11.12 CPA_GetCurrentPosition

The CPA_GetCurrentPosition() method returns only the current position.

```
CPA_RESULTTYPE (*CPA_GetCurrentPosition)(
      CPA_HANDLE hPipe,
      CPA_HANDLE hContent,
      CPA_POSITIONTYPE* pPosition);
```

The implementation is not mandated to support 64 bit position specifier for GetPosition even if the flag CPA_64BITSUPPORTED is set. When any position in pPosition exceeds 2^31 and if

64 bit file handling is not supported, the implementation shall return CPA_OKPOSITIONEXCEED2GB and place CPA_POSITION_32_MAX in one or more fields in pPosition.

This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe [in]	Handle of the content pipe interface instance.
hContent [in]	Handle of the content stream.
pPosition [out]	Information about the current byte position of the pipe within the specified content, the content size and the window for the available content.

2.11.13 CPA_Read

The CPA_Read() method retrieves data of the specified size from the content stream and advances the content pointer by the size of the data. Note that the pipe client provides the pointer to accept the data. This method is therefore appropriate for small high frequency reads.

```
CPA_RESULTTYPE (*CPA_Read)(
      CPA_HANDLE hPipe,
      CPA_HANDLE hContent,
      CPA_BYTE* pData,
      CPA_U32* pSize);
```

If current stream is opened in write mode, CPA_EACCES shall be returned. Read is expected to leave the following result codes depending on the current position if no other error occurs:

Area / Position in Figure 2	Expected result code
ab	CPA_EPOSNOTAVAIL
bf	CPA_EAGAIN or CPA_EPOSNOTAVAIL
fc	CPA_OK
cl	CPA_OK
le	CPA_EAGAIN
L	CPA_OKEOS
eo	CPA_EPOSNOTAVAIL

This is a blocking call. Relevant errors include: CPA_EINVAL, and CPA_EIO. The description of each parameter is given as follows.

Parameter	Description
hPipe [in]	Handle of the content pipe interface instance.
hContent [in]	Handle of the content stream.
pData [out]	Client specified pointer to receive data.
pSize [in/out]	Prior to call: number of bytes to read. After call: number of bytes actually read.

2.11.14 CPA_ReadBuffer

The CPA_ReadBuffer() method retrieves a buffer allocated by the pipe containing the requested number of bytes from the content stream. The content pointer advances by the number of bytes read. Note that the pipe itself provides the pointer to the data. This method is therefore appropriate for large low frequency reads. The client shall call CPA_ReleaseReadBuffer() when done with the buffer to return it to the pipe.

In some cases he requested block might not reside in contiguous memory within the pipe implementation. For instance, if the pipe leverages a circular buffer then the requested block might straddle the boundary of the circular buffer. By default a pipe implementation performs a copy in this case to provide the block to the pipe client in one contiguous buffer. If, however, the client sets bForbidCopy, then the pipe returns only those bytes preceding the memory boundary. Here the client may retrieve the data in segments over successive calls. If bForbidCopy is unset and the result code equals CPA_OK, ReadBuffer must always return a full buffer of the required size. At End Of Stream, the result code CPA_OKEOS is used instead. If no more data is available, the reported pSize equals zero and the result code CPA_OKEOS is delivered. In this case, ppBuffer may point to NULL.

```
CPA_RESULTTYPE (*CPA_ReadBuffer)(
      CPA_HANDLE hPipe,
      CPA_HANDLE hContent,
      CPA_BYTE** ppBuffer,
      CPA_U32* pSize,
      CPA_BOOL bForbidCopy);
```

If current stream is opened in write mode, CPA_EACCES shall be returned. CPA_ReadBuffer() is expected leave the following result codes depending on the current position if no other error occurs:

Area / Position in Figure 2	Expected result code
ab	CPA_EPOSNOTAVAIL
bf	CPA_EAGAIN or CPA_EPOSNOTAVAIL
Fc	CPA_OK
Cl	CPA_OK
Le	CPA_EAGAIN

Area / Position in Figure 2	Expected result code
L	CPA_OKEOS
Eo	CPA_EPOSNOTAVAIL

This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.
hContent *[in]*	Handle of the content stream.
ppBuffer *[out]*	Pointer to receive a pipe supplied data buffer.
pSize *[in/out]*	Prior to call: number of bytes to read. After call: number of bytes actually read.
bForbidCopy *[in]*	If set the pipe shall never perform a copy opting instead to obtain less bytes than what is requested.

2.11.15 CPA_ReleaseReadBuffer

The `CPA_ReleaseReadBuffer()` returns a buffer previously acquired via a call to `CPA_ReadBuffer()`.

```
CPA_RESULTTYPE (*CPA_ReleaseReadBuffer)(
      CPA_HANDLE hPipe,
      CPA_HANDLE hContent,
      CPA_BYTE* pBuffer);
```

This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.
hContent *[in]*	Handle of the content stream.
pBuffer *[in]*	Pipe supplied read buffer being released (i.e. returned to pipe).

2.11.16 CPA_Write

The `CPA_Write()` method writes data of the specified size to the content stream and advances the content pointer by the size of the data. Note that the pipe client provides the pointer to accept the data.

```
CPA_RESULTTYPE (*CPA_Write)(
      CPA_HANDLE hPipe,
      CPA_HANDLE hContent,
      CPA_BYTE* data,
      CPA_U32* pSize);
```

If current stream is opened in read mode, CPA_EACCES shall be returned. This is a blocking call. The description of each parameter is given as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.
hContent *[in]*	Handle of the content stream.
pData *[in]*	Client specified pointer to data.
nSize *[in/out]*	Prior to call: number of bytes to write. After call: number of bytes actually written.

2.11.17 CPA_RegisterCallback

The CPA_RegisterCallback() method registers a client event callback for a given content handle with the pipe.

```
CPA_RESULTTYPE (*CPA_RegisterCallback)(
      CPA_HANDLE hPipe,
      CPA_CALLBACKTYPE ClientCallback,
      CPA_PTR ClientContext);
```

This is a blocking call. The description of each parameter is given in as follows.

Parameter	Description
hPipe *[in]*	Handle of the content pipe interface instance.
ClientCallback *[in]*	Event callback to register.
ClientContext *[in]*	Client context information.

3 Example Use Cases

3.1 OpenMAX IL Playback/Parser Use Case

Consider the OpenMAX IL playback use case where a media processing IL component is responsible for parsing data from pieces of source content. The following steps occur:

1. The IL client specifies the source content to the IL component by URI.
2. The IL client optionally specifies the mechanism for accessing the source content (i.e. the content pipe) to the IL component.
3. If the IL client has not specified a content pipe to the IL component when the IL component transits from LOADED state, the IL component must acquire an instance of the content pipe interface itself (e.g. via an OpenMAX IL Core function or some implementation specific mechanism).
4. At the appropriate time the IL component opens the content stream specified by the IL client using the acquired content pipe interface instance and its method `CPA_Open()`.
5. The IL component performs `CPA_Read()` on the source content using the content pipe interface, parses that content, and plays it.
6. At the appropriate time the IL component closes the content streams using the content pipes `CPA_Close()` method.
7. Finally the content pipe interface instance itself is released if no more content shall be opened (or created) with the current content pipe interface instance.

3.2 OpenMAX IL Recording/Combiner Use Case

Consider the OpenMAX IL recording use case where a media processing IL component is responsible for emitting final data (perhaps muxed and packaged by a “combiner”) to a piece of content. The following steps occur:

1. The IL client specifies the destination content to the IL component by URI.
2. The IL client optionally specifies the mechanism for accessing the destination content (i.e. the content pipe) to the IL component.
3. If the IL client has not specified a content pipe to the IL component when the IL component transits from LOADED state, the IL component must acquire an instance of the content pipe interface itself (e.g. via an OpenMAX IL Core function or some implementation specific mechanism).
4. At the appropriate time the IL component creates the content stream specified by the IL client using the acquired content pipe interface instance and its method CPA_Create().
5. The IL component performs `CPA_Write()`on the destination content stream using the content pipe sending muxed/packaged data to it.
6. At the appropriate time the IL component closes the content streams using the content pipe `CPA_Close()` method.

7. Finally, the content pipe interface instance itself is released if no more content shall be opened (or created) with the current content pipe interface instance.

3.3 OpenMAX AL Playback Use Case

Consider the OpenMAX AL playback use case using the Media Player object to playback media content. The following steps occur:

1. The OpenMAX AL client creates an instance of the content pipe interface using the CPA_GetContentPipe() method.
2. The OpenMAX AL client creates an **XADataLocator_ContentPipe** structure and fills in the pContentPipe member with the handle to the content pipe interface.
3. The OpenMAX AL client creates an **XADataSource** structure and fills in the pLocator member with a pointer to the **XADataLocator_ContentPipe** structure created earlier.
4. The OpenMAX AL client creates a Media Player object using the **XADataSource** as the data source.
5. The OpenMAX AL implementation opens the content pipe using CPA_Open()
6. The OpenMAX AL implementation uses the method CPA_Read() to obtain data during playback.
7. When playback is complete, the OpenMAX AL implementation closes the content pipe using CPA_Close().
8. Finally, if no more content is to be played, the OpenMAX AL client closes content pipe interface using the CPA_ReleaseContentPipe() method.

3.4 OpenMAX AL Record Use Case

Consider the OpenMAX AL recording use case using the Media Recorder object to capture mdeia content. The following steps occur:

1. The OpenMAX AL client creates an instance of the content pipe interface using the CPA_GetContentPipe() method.
2. The OpenMAX AL client creates an **XADataLocator_ContentPipe** structure and fills in the pContentPipe member with the handle to the content pipe interface.
3. The OpenMAX AL client creates an **XADataSource** structure and fills in the pLocator member with a pointer to the **XADataLocator_ContentPipe** structure created earlier.
4. The OpenMAX AL client creates a Media Recorder object using the **XADataSink** as the data sink.
5. The OpenMAX AL implementation creates the content pipe using CPA_Create()

6. The OpenMAX AL implementation uses the method CPA_Write() to write data during recording.
7. When recording is complete, the OpenMAX AL implementation closes the content pipe using CPA_Close().
8. Finally, if no more content is to be captured, the OpenMAX AL client closes the content pipe interface using the CPA_ReleaseContentPipe() method.

3.5 OpenSL ES Playback Use Case

Consider the OpenSL ES playback use case using the Audio Player object.. The following steps occur:

1. The OpenSL ES client creates an instance of the content pipe interface using the CPA_GetContentPipe() method.
2. The OpenSL ES client creates an **SLDataLocator_ContentPipe** structure and fills in the pContentPipe member with the handle to the content pipe interface.
3. The OpenSL ES client creates an **SLDataSource** structure and fills in the pLocator member with a pointer to the **SLDataLocator_ContentPipe** structure created earlier.
4. The OpenSL ES client creates an Audio Player object using the **SLDataSource** as the data source.
5. The OpenSL ES implementation opens the content pipe using CPA_Open()
6. The OpenSL ES implementation uses the method CPA_Read() to obtain data during playback.
7. When playback is complete, the OpenSL ES implementation closes the content pipe using CPA_Close().
8. Finally, if no more content is to played, the OpenSL ES client closes the content pipe interface using the CPA_ReleaseContentPipe() method.

3.6 OpenSL ES Record Use Case

Consider the OpenSL ES recording use case using the Audio Recorder object. The following steps occur:

1. The OpenSL ES client creates an instance of the content pipe interface using the CPA_GetContentPipe() method.
2. The OpenSL ES client creates an **SLDataLocator_ContentPipe** structure and fills in the pContentPipe member with the handle to the content pipe interface.
3. The OpenSL ES client creates an **SLDataSource** structure and fills in the pLocator member with a pointer to the **SLDataLocator_ContentPipe** structure created earlier.
4. The OpenSL ES client creates an Audio Recorder object using the **SLDataSink** as the data sink.

5. The OpenSL ES implementation creates the content pipe using CPA_Create()
6. The OpenSL ES implementation uses the method CPA_Write() to write data during recording.
7. When recording is complete, the OpenSL ES implementation closes the content pipe using CPA_Close().
8. Finally, if no more content is to be captured, the OpenSL ES client closes the content pipe interface using the CPA_ReleaseContentPipe() method.