UHAPI/0300274

API Specification Reader’s Guide

Version 1.0 - 30 July 2004   API Specification

Document Information

Info	Content
Keywords	
Abstract	
Status	Approved

Revision History

Version	Date	Description
1.0	040730	UHAPI initial API Specification Reader's Guide

Disclaimers

Document status	Product status	Definition
Objective data	Development	This document contains data from the objective specification for product development. UHAPI FORUM reserves the right to change the specification in any manner without notice.
Preliminary data	Qualification	This document contains data from the preliminary specification. Supplementary data will be published at a later date. UHAPI FORUM reserves the right to change the specification without notice, in order to improve the design and supply the best possible product

Table of Contents

Introduction

Summary

The purpose of this document is to act as a guide for the reader of a UHAPI specification (referred to as an *API specification* in the rest of the document). It should help the reader in finding his way in the specification and in interpreting the contents of the various sections of the specification in the right way. The document discusses the general structure of an API specification and follows that structure to discuss each section that may occur in the specification. Hence the structure of this document reflects the structure of a UHAPI specification.

The purpose of this document is *not* to provide a general introduction to the main concepts underlying the UHAPI specification approach or to provide a rationale for the approach. These can be found in [1] which is advised as background reading for reading this document. A summary of the main concepts that are discussed in [1] can be found in the “definition of terms” in this chapter.

In explaining the structure and interpretation of an API specification several examples from existing UHAPI specifications are used, in particular from [2] and [3].

Definition of Terms

Term	Description
`API specification`	A UHAPI specification. Defines a collection of software interfaces providing access to coherent streaming-related functionality.
`Interface suite`	A collection of mutually related interfaces providing access to coherent functionality.
`Logical component`	A coherent unit of functionality that interacts with its environment through explicit interfaces only.
`Role`	An abstract class, i.e. a class without implementation defining behavior only.
`Role instance`	An object *playing* a role, i.e. an object displaying the behavior defined by the role,
`Attribute`	An *instance variable* associated with a role. Attributes are used to associate state information with roles.
`Signature`	A definition of the syntactic structure of a specification item such as a type, interface or function in IDL. For C functions, signature is equivalent to *prototype*.
`Specification item`	An entity defined in a specification such as a data type, role, attribute, interface, function, etc.
`IDL`	Interface Definition Language.
`Qualifier`	A predefined keyword representing a property or constraint imposed on a specification item.
`Constraint`	A restriction that applies to a specification item,
`Execution constraint`	A constraint on multi-threaded behavior.
`Base document`	An API specification A that uses definitions from another API specification B is said to be *based on* B. B is called a *base document* of A.
`Model type`	A data type used for specification (modeling) purposes only, such as *set*, *map* and *entity*.
`Model constant`	A constant used for specification (modeling) purposes only.

Term	Description
`Enum element type`	An enumerated type whose values can be used to construct *sets* (bit vectors) of at most 32 values by logical or-ing.
`Enum set type`	A 32-bit integer data type representing sets of enumerated values.
`Set type`	A data type whose values are mathematical sets of values of a specific type. Unlike enum sets, the sets may be infinite.
`Map type`	A data type whose values are *tables* mapping values of one type (the domain type) to values of another type (the range type). Maps are a kind of *generalized arrays*. Unlike arrays, the domain and range types may be arbitrary and possibly infinite types.
`Entity type`	A class of objects that may have attributes associated with them.
`Interface-role model`	An extended UML class diagram showing the roles and interfaces associated with a logical component and their mutual relations.
`Logical component instance`	An incarnation of a logical component, i.e. a configuration of objects displaying the behavior defined by the logical component.
`Provides interface`	An interface that is *provided* by a role or role instance.
`Requires interface`	An interface that is *used* by a role or role instance.
`Specialization`	A role *S specializes* a role *R* if the behavior defined by *S* implies the behavior defined by *R*, i.e. if *S* has more specific behavior than *R*. Specialization is also referred to as *behavioral inheritance*.
`Diversity`	The set of all parameters that can be set at instantiation time of a logical component and that will not change during the lifetime of the logical component.
`Mandatory interface`	A provides interface of a role that should be implemented by each instance of the role.
`Optional interface`	A provides interface of a role that need not be implemented by each instance of the role.
`Configurable item`	A parameter that can be set at instantiation time of a logical component, usually represented by a role attribute.
`Diversity attribute`	A role attribute that represents a configurable item.
`Instantiation`	The process of creating an *instance* (an incarnation) of a role or logical component.
`Initial state`	The state of a role instance or logical component instance immediately after its instantiation.
`Observable behavior`	The behavior that can be observed at the external software and streaming interfaces of a logical component.
`Function behavior`	The behavior of the functions in the provides interfaces of a role.
`Streaming behavior`	The input-output behavior of the streams associated with a role.
`Active behavior`	The autonomous behavior that is visible at the *provides* and *requires* interfaces of a role.
`Instantiation behavior`	The behavior of a role at instantiation time of a logical component.
`Independent attribute`	An attribute whose value may be defined or changed independently of other attributes and entities.
`Dependent attribute`	An attribute whose value is a function of the values of other attributes or entities.
`Invariant`	An assertion about a role or logical component that is *always* true from the external observer’s point of view. In reality, the assertion may temporarily be violated.

Term	Description
`Callback interface`	An interface provided by a client of a logical component whose functions are called by the logical component. A notification interface is an example of this, but there may be other call-back interfaces as well e.g. associated with plug-ins.
`Callback-compliance`	The general constraint that the functions in a callback interface should not interfere with the behavior of the caller in an undesirable way, such as by blocking the caller or by delaying it too long.
`Event notification`	The act of reporting the occurrence of events to *interested* objects.
`Event subscription`	The act of recording the types of events that should be notified to objects.
`Cookie`	A special integer value that is used to identify an event subscription. Clients pass cookies to a logical component when subscribing to events and logical components pass them back to clients when notifying the events.
`Event-action table`	A table associating events that can occur to actions that will be performed in reaction to the events; used to specify event-driven behavior.
`Non-standard event notification`	An event notification that is accompanied by other actions (such as state changes of the notifying logical component).
`Client role`	A role modeling the users of a logical component.
`Actor role`	A role (usually a client role) whose active behavior consists of calling functions in interfaces without any a priori constraints on when these calls will occur.
`Control interface`	An interface provided by a logical component that allows the logical component's functionality to be controlled by a client.
`Notification interface`	An interface provided by a client of a logical component that is used by the logical component to report the occurrence of events to the client.
`Specialized interface`	An interface of a role *R* that is inherited from another role and further constrained by *R*.
`Precondition`	An assertion that should be true immediately before a function is called.
`Action clause`	The part of an extended pre- and postcondition specification defining the *abstract action* performed by a function. The abstract action usually defines which variables are modified and/or which out-calls are made by the function.
`Out-call`	An out-going function call of an object on an interface of another object.
`Postcondition`	An assertion that will be true immediately after a function has been called.
`Asynchronous function`	A function with a delayed effect, i.e. the effect of the function will occur some time after returning from the function call.

References

[1] UHAPI: Structure & Specification, UHAPI/03061.

[2] Analog Audio Decoding, UHAPI Specification, UHAPI/0400228.

[3] Video Mixing, UHAPI Specification, UHAPI/02043.

[4] uhIUnknown, UHAPI Specification, UHAPI/02074.

[5] Pin Objects, UHAPI Specification, UHAPI/02246.

[6] Notification, UHAPI Specification, UHAPI/02075.

[7] Qualifiers Quick Reference, UHAPI/02135

Chapter 1 The API Specification

1.1 Introduction

The purpose of this chapter is to discuss some of the main concepts in connection with API specifications and to present the overall structure of an API specification.

1.2 Concepts

1.2.1 Interface Suites

The purpose of an API specification is to define a collection of software interfaces providing access to some coherent and usually streaming-related chunk of functionality. Such a collection of interfaces is referred to as an *interface suite*. The names of interface suites in the UHAPI refer to the functionality associated with the interfaces, e.g. *Dynamic Noise Reduction*, *Video Mixing*, *VBI Slicing*, etc.

Example (Interface Suite)

The *Analog Audio Decoding* interface suite is a collection of interfaces providing access to Analog Audio Decoding functionality. Among other things, this interface suite contains an interface `uhIAnaAdec` for controlling audio decoding and an interface `uhIAnaAdecSelector` for controlling audio program selection.

End of Example (Interface Suite)

1.2.2 Interfaces versus Functionality

Providing and using interfaces should not be confused with providing and using functionality. Functionality is generally provided by means of a *collection* of interfaces (an interface suite). Generally speaking, the interfaces in this collection are *provides* as well *requires* interfaces (from the point of view of the provider of the functionality).

Example (Interfaces versus Functionality)

Providing Analog Audio Decoding functionality is not the same as providing the interfaces `uhIAnaAdec` and `uhIAnaAdecSelector`. The analog audio decoding and selection functionality is provided not only by means of the *provides* interfaces `uhIAnaAdec` and `uhIAnaAdecSelector` but also by the *requires* interface `uhIAnaAdecNtf` which should be provided by the client of the functionality.

End of Example (Interfaces versus Functionality)

1.2.3 Logical Components

From a conceptual point of view the functionality associated with an interface suite can be viewed as a *black box* that interacts with its environment through the software interfaces in the interface suite and possible other interfaces such as streaming interfaces. Specifying the behavior associated with the interfaces amounts to specifying the external behavior of this black box. Because the black box is conceptual it is also referred to as a *logical component*.

Since interfaces and their associated functionality are inseparable, the terms *interface suite* and *logical component* are often used in an interchangeable way. In a narrow sense, the term *logical component* is sometimes reserved for those interface suites that are direct building blocks of the UHAPI and not just building blocks of other interface suites. Examples of the latter are the *uhIUnknown* [4] and *Pin Objects* [5] interface suites and the generic *Notification* [6] interface suite. These interface suites never occur stand-alone but always in combination with other interface suites.

Example (Logical Component)

When using the terms *Video Mixing logical component* and *Video Mixing interface suite* we usually mean the same thing. The focus in the former is on the Video Mixing *functionality* while the focus in the latter is on the Video Mixing *interfaces*, but these two aspects are closely related.

End of Example (Logical Component)

1.2.4 Logical Components and Implementation Components

The concept of a logical component should not be confused with that of an implementation component. An API specification specifies functionality and the way this functionality can be accessed by means of interfaces. It does not specify how this functionality should be implemented, let alone how it should be packaged. The functionality associated with the interfaces *may* be implemented in a single implementation component, but could just as well be implemented in multiple implementation components, as part of one or more implementation components or even by not using components at all. The difference between logical and implementation components is also reflected in their names.

Example (Implementation Component)

An implementation of the *Video Mixing* logical component is referred to as a *video mixer*. The video mixer need not be a single implementation component but could be a collection of implementation components.

End of Example (Implementation Component)

1.2.5 API Specifications as Contracts

There are two sides to functionality: the side of the *providers* and the side of the *clients* of the functionality. Classically an API is viewed as a single interface that allows a client to access functionality implemented by a provider. The API specification can then be viewed as a bilateral contract between the provider and the client defining the rights and obligations of both. UHAPI specifications are based on a generalized version of the classical contractual paradigm: an API specification is viewed as a *multilateral contract* defining multiple mutually related interfaces. Key to this generalized notion of contract is the concept of *role*.

1.2.6 Roles

A role identifies a contractual party, where each role acts as a provider of one or more interfaces, as a client of one or more interfaces, or as both. Each role has contractual rights and obligations associated with it. When developing code that implements or uses the API the developer should make explicit which roles are to be *played* by the code. The rights and obligations specified for these roles in the API specification then define what API-implied requirements have to be satisfied by the code.

Example (Roles)

Three roles identified in the *Analog Audio Decoding* logical component are: `AnaAdec` (decoder), `AnaAdecSelector` (program selector) and `AnaAdecClient` (client of the interfaces). This illustrates that in the specification of a logical component the responsibilities for providing the overall functionality of the logical component may be divided over several roles *played* by the logical component.

End of Example (Roles)

1.2.7 Roles and Implementation Classes

Roles should not be confused with implementation classes although the names of roles may sometimes suggest otherwise. Roles define required/allowed behavior and do not define how that behavior should be realized. The correspondence between roles and implementation classes need not be one-to-one. A single role can e.g. be implemented by multiple implementation classes and a single implementation class can implement multiple roles.

The difference between roles and implementation classes is similar to the difference between logical components and implementation components, i.e. roles are a kind of *logical classes*. The UML term for a role is *abstract class*, i.e. a class representing behavior and having no implementation. This implies that roles are (abstract) classes and not objects. At run-time there may be multiple *instances* of a role, i.e. multiple objects that *play* the role.

Example (Implementation Class)

In an object-oriented implementation the `AnaAdecSelector` role from the *Analog Audio Decoding* logical component is typically implemented by a co-class with the same name. Because an analog audio decoder/selector has multiple program selectors, multiple objects of type `AnaAdecSelector` will be created at run-time, each of which can be viewed as an instance of the `AnaAdecSelector` role.

End of Example (Implementation Class)

1.3 Patterns

An API specification is itself a composition of more fine-grained specifications defining *specification items* such as constants, data types, roles, interfaces and methods. For each type of specification item fixed specification patterns are used which are discussed in the next chapters. Although different, all of these specification patterns have a common *super-structure* defined by a number of sections that occur in all specification patterns (and in fixed order). Being aware of this common structure makes it easier to read and find your way in the specifications. The common sections are:

1. **Signature**

   Contains the IDL definition of the specification item. The text in this section occurs literally in the IDL definition of the API as a whole. The only exception is the **Signature** section of a *model type or constant* (see Section 4.5), which consists of pseudo-IDL that will not be contained in the IDL of the API. Interface specifications do not have a **Signature** section because the contents of this section is essentially the concatenation of the **Signature** sections of the methods defined in the interface.

2. **Qualifiers**

   Provides a list of *qualifiers* which are predefined keywords that can be associated with a specification item. Each qualifier represents a property or constraint that is being imposed on the specification item. The valid qualifiers for each specification item and their meaning are defined in a separate document [7] and can also be found in Appendix A. Qualifiers are similar to *stereotypes* in UML except that multiple qualifiers may be associated with a specification item. They are typically used as abbreviations for standard execution constraints (e.g. the qualifier *single-threaded*) or the specification of standard behavior (e.g. the qualifier *subscribe-function*).

3. **Description**

   Provides a short informal description of the specification item.

4. **(Execution) Constraints**

   Defines restrictions that apply to the specification item. Examples of these are restrictions on the set of allowed values of a data type (e.g. min-max constraints) and *execution constraints* that impose multi-threading constraints on the clients of interfaces. Note that several frequently occurring constraints (such as being *single-threaded*) have been defined as qualifiers, so these constraints are specified in the **Qualifiers** section rather than the **Constraints** section.

5. **Remarks**

   Contains a list of remarks related to the specification item.

Example (Common Specification Pattern)

The specification of the `uhPixFmtType_t` data type from the *Global Types* specification contains all five common sections referred to above:

Signature

```
typedef UInt32 uhPixFmtType_t, *puhPixFmtType_t;
```

Qualifiers

- sub-type

Description

Defines the *union* of the pixel format classes for video as a subrange of the integers. That is, a value of this type is a value of type `uhPixYuvFmt_t`, `uhPixRgbFmt_t` or `uhPixClutFmt_t`. In order to interpret a value of this type correctly the pixel format class should be indicated. For example, the value 0x00000004 can be interpreted in one of three ways:

- as the value `uhPixYuvFmt_Vyuy` of pixel format class `uhPixYuvFmt_t`;
- as the value `uhPixRgbFmt_Xrgb4444` of pixel format class `uhPixRgbFmt_t`;
- as the value `uhPixClutFmt_RgbClut4bpp` of pixel format class `uhPixClutFmt_t`.

Constraints

- Each value of type `uhPixFmtType_t` is a valid value of at least one of the following types: `uhPixYuvFmt_t`, `uhPixRgbFmt_t` or `uhPixClutFmt_t`.

Remarks

- A value of this type is normally used in combination with a value of type `uhPixFmtCls_t` to indicate the pixel format class the value belongs to.

End of Example (Common Specification Pattern)

1.4 Structure of an API Specification

The idea of API specifications as contracts is reflected in the structure of an API specification which consists of the following chapters:

1. **Introduction**

   This section contains the usual front matter such as a summary, list of references, etc.

2. **Concepts**

   This section is meant to introduce the concepts associated with the functionality being specified and to define the vocabulary used in formulating the contractual rights and obligations.

3. **Types & Constants**

   This section defines types and constants that are used in the specification. Among other things, it contains the definitions of data types that occur in the parameter lists of interface functions and the definitions of the specific error codes that can be returned by these functions.

4. **Logical Component**

   This section defines overall aspects of the logical component such as its structure, its diversity and its instantiation. A central position in this section is taken by the "interface-role model" which is a UML class diagram showing all roles and interfaces and their mutual relations.

5. **Roles**

   This section specifies the roles that represent the functionality provided by the API. A model-oriented style of specification is used, where *attributes* are used to represent state information associated with the roles. Interface functions operate on these attributes but roles can also modify them autonomously.

6. **Interfaces**

   This section contains the actual interface specifications, in particular the specifications of the functions that occur in the interfaces. These functions define the part of role behavior that can be controlled externally from the software. The common style of specification is (extended) pre- and postconditions.

7. **Appendices**

   Appendices are used to collect relevant information that does not fit into any of the other sections.

The structure and contents of the above sections are discussed in more detail in the following chapters.

Example (Structure of an API Specification)

The general structure of an API specification is illustrated by the table of contents of the *Color Transient Improvement* logical component. The top level of this structure is the same for all API specifications:

End of Example (Structure of an API Specification)

Chapter 2 The Introduction

2.1 General

The *Introduction* chapter of an API specification contains the usual introductory material. Each of the sections occurring in this chapter is shortly described below.

1. **Summary**

   Contains a short description of the functionality specified in the document.

2. **Definition of Terms**

   Contains a *quick reference* table providing short descriptions of the main terminology, concepts, acronyms, abbreviations, etc. used in the document. The terms in the table are ordered in logical rather than alphabetical order.

3. **References**

   Contains a list of all documents referred to in the specification. The list of referenced documents is separated into two parts: *Base Documents* and *Other Documents*. The *Base Documents* are other API specifications which this API specification is *based on*. These documents define all external types, constants, interfaces, functions, roles, attributes, etc. referred to in the API specification. The *Other Documents* are all other documents referred to such as standards.

Example (Base Documents)

The reference list from the *Analog Video Encryption* logical component is given below. It shows that this logical component does not support notifications, otherwise there would be a reference to the *Notification* API specification in the *Base Documents* list:

Base Documents

[1] uhIUnknown, UHAPI Specification, UHAPI/02074.

Other Documents

[2] Specifications of the Macrovision Copy Protection Process for Authorised Component Suppliers, Macrovision Corporation, Revision 7.1.L1, September 15, 1998.

End of Example (Base Documents)

Chapter 3 The Concepts

3.1 General

The *Concepts* chapter introduces and discusses the functionality defined in the API specification in an informal and intuitive way. It explains the main terms and concepts that play a role and indicates the context in which the functionality will be used. The concepts are typically illustrated using various kinds of pictures and diagrams. The contents of this chapter is *free format* hence the chapter has no predefined structure.

Example (Concepts 1)

The *Concepts* chapter in the *Analog Audio Decoding* specification has three sections explaining the following three concepts: sound standards, decoding and program selection. In addition, it contains the following schematic picture of an instance of the logical component:

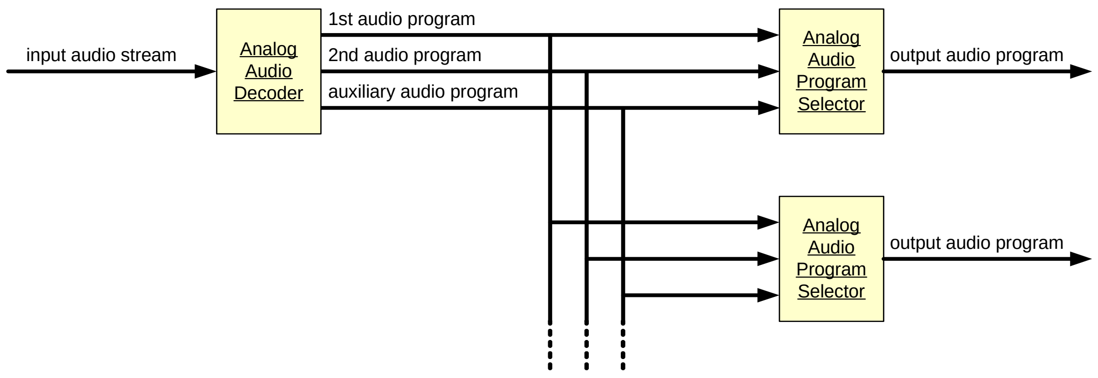

Figure 3-1: Instance of the Analog Audio Decoding Logical Component

End of Example (Concepts 1)

Example (Concepts 2)

The *Concepts* chapter in the *Video Mixing* specification explains concepts such as layers, windows, blanking, color-keying, alpha-blending, etc. Among other things, it contains the following picture showing the video mixing process:

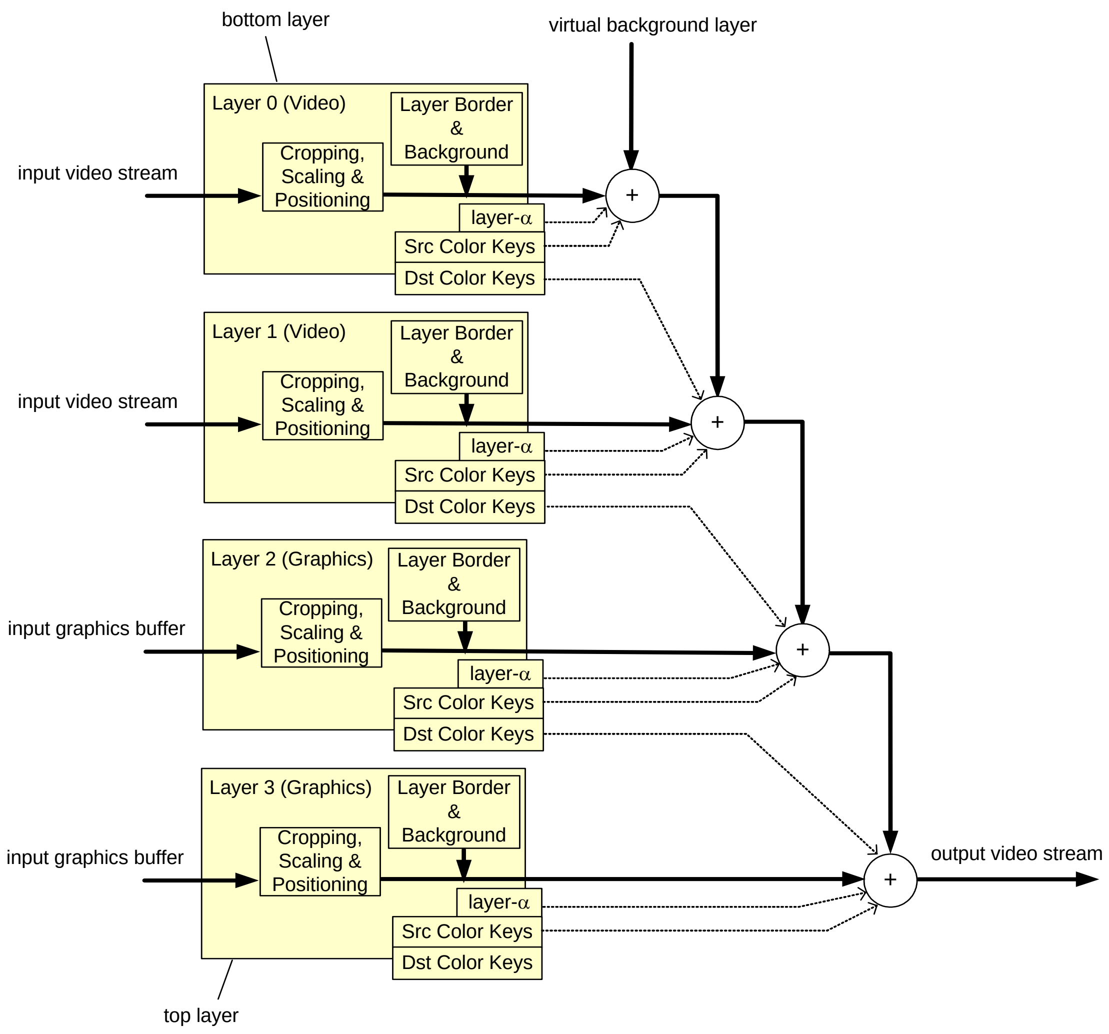

Figure 3-2: The mixing process (in a mixer with two video layers and two graphics layers).

End of Example (Concepts 2)

Chapter 4 The Types & Constants

4.1 General

The *Types & Constants* chapter contains the specifications of all types and constants that play a role in the API specification and that are not already specified in one of the base documents. The chapter is divided into two parts.

The first part deals with the *public types and constants* which are the types and constants that are visible to the application programmer: their definitions occur in the IDL associated with the API. The second part deals with the *model types and constants* which are the types and constants introduced for modeling purposes only: these types and constants will not occur in the IDL of the API.

Public and model types and constants are specified in essentially the same way except that the definitions of model types and constants can use additional data types; see Section 4.5. Each type is specified in a separate subsection; the name of the type is used as the heading of this section. Type specifications are discussed in Section 4.3 and Section 4.4.

Constants are specified in groups of related constants. Each group is defined in a separate subsection; the name of the group (e.g. *Error Codes*) is used as the heading of this section. In case a constant group consists of a single constant, the name of the constant is used as the name of the constant group. Constant specifications are discussed in Section 4.2.

4.2 Constant Specifications

A constant specification defines a group of related constants and consists of the following sections, some of which are optional:

1. **Signature**

   The IDL definition of the constants (see Section 1.3). In the case of *model constants* this may be *pseudo-IDL* as explained in Section 4.5.

2. **Qualifiers**

   The qualifiers associated with the constants (see Section 1.3). An example is the *error-codes* qualifier which states that the constants being defined are error codes returned by functions.

3. **Description**

   A short informal description of the group of constants.

4. **Constants**

   Contains a table providing a short informal description of each constant in the group. In case the group of constants being defined consists of a single constant this section is omitted and the informal description of the constant is contained in the **Description** section.

5. **Remarks**

   Contains a list of remarks related to the constants.

Example (Constant Group Specification)

The section below from the specification of the *Video Mixing* logical component defines the error codes associated with the logical component as a group of constants with name *Error Codes*:

Error Codes

Signature

```
const uhErrorCode_t UH_ERR_VMIX_FADING          = 0x00000801;
const uhErrorCode_t UH_ERR_VMIX_SETTING_WINDOWS = 0x00000802;
```

Qualifiers

- error-codes

Description

Defines the error codes returned by the functions in the interfaces of a Video Mixer.

Constants

Name	Description
`UH_ERR_VMIX_FADING`	The layer is currently fading (warning returned when one queries the layer's blending factor during a fade).
`UH_ERR_VMIX_SETTING_WINDOWS`	The layer is currently moving its windows (warning returned when one queries the layer's windows or scale mode).

End of Example (Constant Group Specification)

Example (Single Constant Specification)

The section below from the specification of the *Analog Audio Decoding* logical component defines a single constant `uhAnaAdec_AllNtfs`. (This constant is used in subscribing to all notifications associated with a notification interface. Each notification interface in each logical component has such a constant associated with it.) Because this section defines a single constant, there is no **Constants** section. Instead, the name and description of the constant are contained in the title and **Description** part of the section, respectively:

uhAnaAdec_AllNtfs

Signature

```
const uhAnaAdec_NtfSet_t uhAnaAdec_AllNtfs = 0x00000007;
```

Qualifiers

None.

Description

Defines the set of all values of type `uhAnaAdec_Ntf_t`. This constant can be used in calls of the `Subscribe` and `Unsubscribe` functions to enable or disable all event notifications, respectively.

End of Example (Single Constant Specification)

4.3 Type Specifications

A type specification defines a data type and consists of the following sections, some of which are optional:

1. **Signature**

   The IDL definition of the data type (see Section 1.3). In the case of a *model type* this may be *pseudo-IDL* as explained in Section 4.5.

2. **Qualifiers**

   The qualifiers associated with the type (see Section 1.3). Examples of frequently used qualifiers are the *enum-element* and *enum-set* qualifiers for enumerated data types; these are explained in Section 4.4.

3. **Description**

   A short informal description of the data type.

4. **Values / Fields / Attributes**

   Contains a table providing a short informal description of each *member* of the data type. In the case of an enumerated type the members are the *values* being enumerated. In the case of a structure type the members are the *fields* of the structure. Entity types are modeling types that have *attributes* as members; they are discussed in Section 4.5. The heading of this section (if present) varies dependent on the kind of data type being defined.

5. **Constraints**

   Defines restrictions that apply to the values of the data type. For example, when defining a structure type there could be a dependency between the values of the fields which can be expressed as a constraint.

6. **Remarks**

   Contains a list of remarks related to the data type.

Example (Type Specification)

The type specification below from the *Video Mixing* logical component defines the data type `uhVmix_NewGfxLayerProp_t`. Because it is a struct type, there is a **Fields** section describing the fields of the structure:

uhVmix_NewGfxLayerProp_t

Signature

```
typedef struct _uhVmix_NewGfxLayerProp_t {
    UInt32 horResolution;
    UInt32 vertResolution;
} uhVmix_NewGfxLayerProp_t, *puhVmix_NewGfxLayerProp_t;
```

Qualifiers

None.

Description

Values of this type contain the properties of the input of a graphics layer. The type is used to notify input graphics property changes to clients.

Fields

Name	Description
`horResolution`	The horizontal resolution of the graphics layer input.
`vertResolution`	The vertical resolution of the graphics layer input.

End of Example (Type Specification)

4.4 Enums and Enum Sets

Enumerated data types are frequently used in APIs. It is also common practice in C to use bitwise or-ing of enumerated values to represent sets of enumerated values. In common programming practice no distinction is made between the enumerated type and the set type which can lead to confusion. In API specifications the distinction between the two is explicitly made. The qualifiers *enum-element* and *enum-set* are used for this.

If the values of an enumerated type are meant to be used as *elements* of sets (bit vectors), the enumerated type gets the qualifier *enum-element*. In that case the values of the enumerated type are defined as powers of two.

Example (Specification of an Enum-Element Type)

The data type `uhAnaAdec_SoundStandard_t` from the *Analog Audio Decoding* logical component defines sound standards as enumerated values that can be bitwise or-ed to constructs sets of sound standards, i.e. values of type `uhAnaAdec_SoundStandardSet_t` (see next example):

uhAnaAdec_SoundStandard_t

Signature

```
typedef enum _uhAnaAdec_SoundStandard_t {
    uhAnaAdec_AmMono      = 0x00000001,
    uhAnaAdec_FmMono      = 0x00000002,
    uhAnaAdec_2Cs         = 0x00000004,
    uhAnaAdec_Btsc        = 0x00000008,
    uhAnaAdec_NicamAmMono = 0x00000010,
    uhAnaAdec_NicamFmMono = 0x00000020
} uhAnaAdec_SoundStandard_t, *puhAnaAdec_SoundStandard_t;
```

Qualifiers

- enum-element

Description

Values of this type represent sound standards.

Values

Name	Description
`uhAnaAdec_AmMono`	AM Mono.
`uhAnaAdec_FmMono`	FM Mono.
`uhAnaAdec_2Cs`	2 Carrier Sound (also known as German stereo).
`uhAnaAdec_Btsc`	BTSC.
`uhAnaAdec_NicamAmMono`	NICAM with AM Mono.
`uhAnaAdec_NicamFmMono`	NICAM with FM Mono.

Remarks

- The absence of a sound standard in the input stream is modeled by the value 0. That is, attributes and variables of type `uhAnaAdec_SoundStandard_t` may have the value 0 which can be interpreted as *no sound standard*.

End of Example (Specification of an Enum-Element Type)

Data types that represent sets of enumerated values are *typedef*-ed as UInt32 (32-bit bit vectors) and get the qualifier *enum-set*. Normal enumerated types whose values are not meant to be bitwise or-ed are defined as usual, without using a qualifier.

Example (Specification of an Enum-Set Type)

In the specification of the *Analog Audio Decoding* logical component the type `uhAnaAdec_SoundStandardSet_t` is used as the type of parameters and attributes representing sets of sound standards and not the enumerated type `uhAnaAdec_SoundStandard_t`. Of course, in C this typing is not enforced by the language but it is intended to improve the clarity of specifications. The definition of `uhAnaAdec_SoundStandardSet_t` is given below.

uhAnaAdec_SoundStandardSet_t

Signature

```
typedef UInt32 uhAnaAdec_SoundStandardSet_t, *puhAnaAdec_SoundStandardSet_t;
```

Qualifiers

- enum-set

Description

Values of this type represent sets of sound standards.

End of Example (Specification of an Enum-Set Type)

4.5 Model Types

Besides the normal data types that can occur in IDL, such as Int32, Bool, enum{...}, struct{...}, etc., UHAPI specifications can also contain a few *abstract data types* that are used for specification purposes only. These data types are defined in the *Model Types & Constants* section of the *Types & Constants* chapter. The naming of model types is more liberal than that of public types and constants. Names of model types usually have no prefixes and suffixes such as uh and `_t`.

Model types are mainly used in the definitions of *attributes* associated with *roles*. Besides the normal IDL data types the following three modeling types are used in API specifications:

1. **Set types**

   A set type defines (possibly infinite) mathematical sets of values of a specific type. The notation `setof{`*`type`*`}` is used to denote the type of mathematical sets of values of type *`type`*. For sets the usual mathematical notations are used.

Example (Set Type)

In the *Connection Management* logical component a use case is modeled as an entity (see item 3 below) with two attributes. One of them is defined in *pseudo-IDL* as follows:

```
const setof{Component} compSet;
```

This attribute represents the set of component instances associated with a use case. This set is constant because it is not possible to dynamically add/remove component instances from a use case.

End of Example (Set Type)

2. **Map types**

   A map type defines *tables* that map values of one type (the *domain type*) to values of another type (the *range type*). Values of this type, called *maps*, can be viewed as *generalized arrays* and identical notation is used for these types. An array can be viewed as a map whose domain type is a subrange of the integers starting at 0; in maps the domain and range types may be any type.

Example (Map Type)

In the *Notification* specification [6] the *Subject* role (which models the provider of notification functionality) has the following attribute:

```
uhX_YNtfSet_t subscription[Observer,UInt32];
```

The `subscription` attribute is a map that maps each instance of the `Observer` role (a subscriber to event notifications) and each *cookie* (a kind of subscription ID) to the set of event notifications that the observer has subscribed to. In the specification text the subscription attribute is treated as a normal two-dimensional array even though the *array* is potentially infinite. So the notation `subscription[`*`x`*`,`*`c`*`]` is used to refer to the set of event notifications subscribed to by observer *x* with cookie *c*.

End of Example (Map Type)

3. **Entity types**

   An entity type defines a collection of objects that may have *attributes* associated with them. In UML terms, an entity type is a *class* with public attributes only and no operations, and an entity is an *instance* of the class, i.e. an *object*. The notation `entity{...}` is used to declare an entity type where the attributes are declared as C-style variables inside the braces.

The notations and the specification pattern used for entities are similar to those used for roles (see Chapter 6). Similar to the specification of role attributes, the attributes of an entity type are divided into *independent* attributes, whose value may in principle vary freely, and *dependent* attributes, whose value can be defined in terms of other values (see Section 6.4). An entity type, like a role, can also be defined as a *specialization* of another entity type.

Example (Entity Type)

In the *Analog Audio Decoding* logical component the type `AudioProgram` is used to model audio programs. It is defined as an entity type with two attributes:

AudioProgram

Signature

```
entity AudioProgram {
   Bool present;
   Bool stereo;
}
```

Qualifiers

None.

Description

Entities of this type represent audio programs that can be contained in streams. The only aspects of audio programs that are modeled are their being present and their being stereo or mono.

Independent Attributes

Name	Description
`present`	Indicates whether the audio program is present in the stream or not.
`stereo`	Indicates whether the audio program is stereophonic or not.

End of Example (Entity Type)

Chapter 5 The Logical Component

5.1 General

The *Logical Component* chapter specifies all aspects of the API functionality that relate to the logical component as a whole, i.e. those aspects that are not specific to a particular role, interface or function. This chapter consists of four sections:

1. **Interface-Role Model**

   Defines the *interface-role model* which can be viewed as a graphical summary of the main contractual entities, in particular the roles and interfaces and their mutual relations.

2. **Diversity**

   Defines the parameters that can be set at instantiation time of the logical component.

3. **Instantiation**

   Specifies the result of instantiating a logical component: the objects that are created and their initial state.

4. **Execution Constraints**

   Defines concurrency-related constraints that apply to the logical component as a whole.

These sections are discussed in more detail below.

5.2 Interface-Role Model

One of the explicit goals of the style of specification used in the API specifications is to improve the level of precision of the API specifications without becoming too formal. Besides by using a contractual approach, this is achieved by using a *model-oriented* style of specification based on a restricted use of UML. Model-oriented specifications define behavior in terms of a *model* which can be seen as a kind of *abstract implementation*. The interface-role model defined in the *Interface-Role Model* section is an enhanced UML class diagram that acts as a graphical summary of the model used to specify the logical component. It shows the main entities in the model such as interfaces and roles while leaving out other entities such as attributes and constraints. The latter are dealt with elsewhere (in a non-graphical way). Each API specification defines such an interface-role model.

Example (Interface-Role Model)

The diagram below is the interface-role model of the *Analog Audio Decoding* logical component. The various ingredients of the interface-role model are explained below. Note that the diagram should be interpreted as a UML class diagram and not as an object diagram. For example, the diagram does not imply that an instance of the logical component consists of a single decoder (`AnaAdec`) and a single selector (`AnaAdecSelector`). As indicated by the aggregation relation in the diagram, there may be multiple selectors.

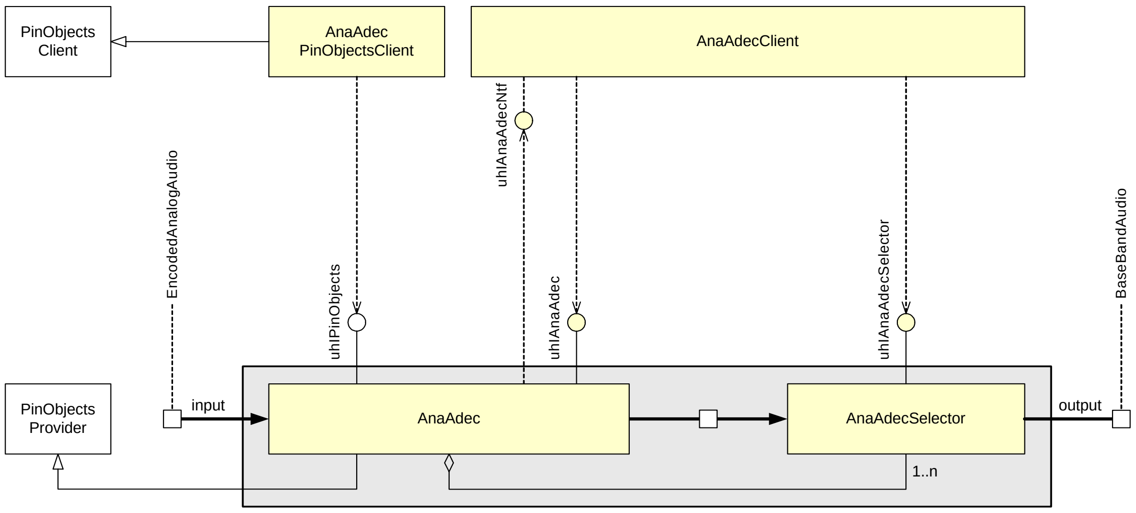

Figure 5-1: Interface-Role Model (Analog Audio Decoding)

End of Example (Interface-Role Model)

The interface-role model consists of the following entities:

1. The *interfaces* defined in the API specification possibly including interfaces defined in the base documents. The interfaces are represented by *lollipops*.
2. The *roles* defined in the API specification possibly including roles defined in the base documents. The roles are represented as (abstract) UML classes.
3. The *provides* and *requires* (or use) relations between the roles and interfaces. The *provides* relation is represented by the stick of a lollipop and the *requires* relation by a UML dependency relation (a dotted arrow).
4. The *specializes* relation between roles, represented by the inverse UML generalization arrow. Role R2 specializing role R1 implies that all contractual rights and obligations that apply to R1 also apply to R2.
5. The *streams* associated with the roles. Streams are represented by small squares.
6. The *input* and *output* relations between roles and streams. The *input* relation is represented by a fat arrow connecting a stream and a role and the *output* relation by a fat line connecting a role and a stream.
7. The *bounding box* of the logical component represented by a grey box. What is inside the box is considered *internal* to the logical component and what is outside is considered *external*.

Figure 5-2 provides a survey of the UML notations used in interface-role diagrams. The small white squares are classes representing streams. A stream may be associated with a role as an input, an output, or both, as indicated by the fat lines and arrows. The small white squares outside the grey box represent the external input and output streams and those inside the grey box represent internal streams introduced for modeling purposes only. In practice, the small white squares representing the streams are sometimes omitted and only the fat lines and arrows are shown. Note that the specialization relation, although shown as a relation between two *client roles*, will most often occur as a relation between the roles inside the grey box.

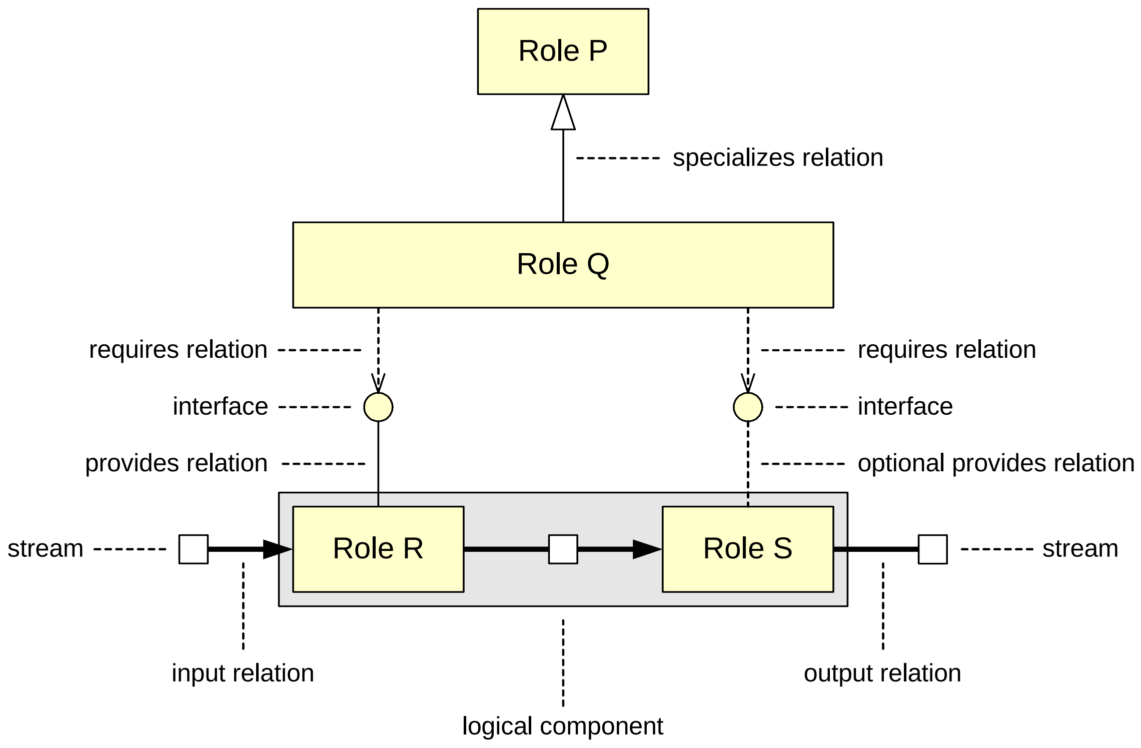

Figure 5-2: Interface-Role Model: UML Notations Used

Roles, interfaces and streams defined in base documents are not normally shown in the interface-role diagram except when they relate to roles that are specialized by roles defined in the API specification. The latter roles and their interfaces are indicated in yellow (very light gray) and the former in white. The roles and interfaces defined in the *uhIUnknown* [4] and *Notification* [6] API specifications are exceptions in that they are not normally shown in the diagram even when being specialized.

Note that the interface-role model provides information on the navigability between the interfaces of the logical component. If two interfaces are provided (directly or indirectly) by the same role, the `QueryInterface` function of `uhIUnknown` can be used to navigate from one interface to another. If two interfaces are provided by two different roles this is not possible.

Example (Interface-Role Models and QueryInterface)

The *Video Mixing* interface-role model (see below) implies that:

- `QueryInterface` can be used to navigate between the following interfaces of a(n instance of a) `VideoLayer`:

  `uhIVmixVidLayer`, `uhIVmixLayer`, `uhIVmixColorKey`, `uhIVmixBorder`, `uhIUnknown`

- `QueryInterface` cannot be used to navigate between the following interfaces:

  `uhIVmixLayer` interface of a `(Gfx/Video)Layer`

  `uhIVmix` interface of a `Mixer`

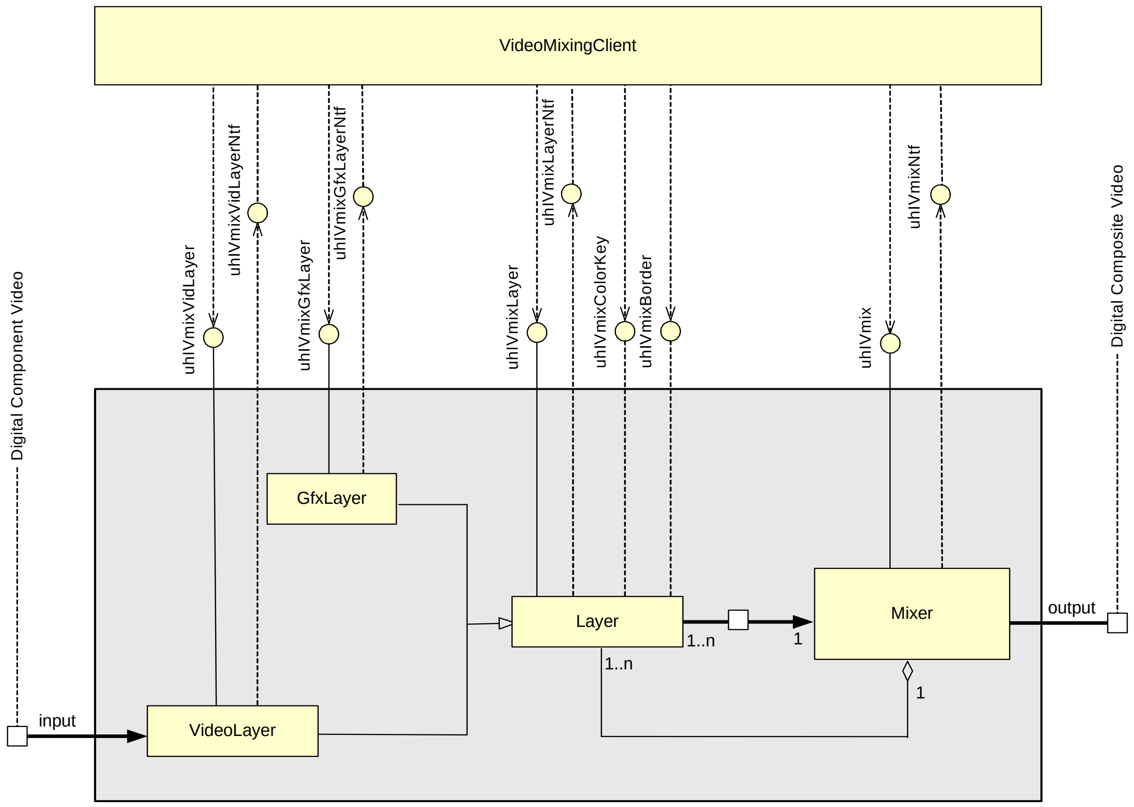

Figure 5-3: Interface-Role Model (Video Mixing)

End of Example (Interface-Role Models and QueryInterface)

5.3 Diversity

The *Diversity* section gives a survey of the diversity associated with the logical component being defined. The diversity of a logical component consists of all parameters that can be set at instantiation time of the logical component and that will not change during the lifetime of the logical component. It is divided into *interface diversity*, defining which interfaces are mandatory/optional when instantiating an instance of the logical component, and *configuration diversity*, defining other parameters that can be set at instantiation time. The latter are modeled by attributes of roles. The interface and configuration diversity are specified in separate subsections. There is a third subsection which is used for specifying constraints on the diversity parameters.

5.3.1 Provided Interfaces

The *Provided Interfaces* section contains a table defining for each interface of the API whether that interface is mandatory or optional. The **Role** column in this table, indicating the provider of an interface, is necessary because the same interface may occur on multiple roles, in particular when two roles specialize the same role defined in a base document.

Note that an interface I_R being *mandatory* on a role *R* means that each instance of *R* has the *obligation* to provide I_R. An interface I_R being *optional* on a role *R* means that each instance of *R* has the *right* to provide I_R but not the obligation. Note that if I_R is optional on *R*, it could be mandatory on a role *S* that specializes *R* but not the other way around. At runtime the presence or absence of an interface can be checked by means of the `QueryInterface` function of the omnipresent `uhIUnknown`.

Example (Provided Interfaces Table)

The *Provided Interfaces* section from the *Analog Audio Decoding* API specification specifies that the logical component (i.e. the roles `AnaAdec` and `AnaAdecSelector`) always provide the interfaces `uhIPinObjects`, `uhIAnaAdec` and `uhIAnaAdecSelector`. A client is free to provide (implement) the notification interface `uhiAnaAdecNtf` or not. In the latter case the only consequence is that he will not be able to subscribe to event notifications.

Provided Interfaces

Role	Interface	Presence
`AnaAdec`	`uhIPinObjects`	mandatory
	`uhIAnaAdec`	mandatory
`AnaAdecClient`	`uhIAnaAdecNtf`	optional
`AnaAdecSelector`	`uhIAnaAdecSelector`	mandatory

End of Example (Provided Interfaces Table)

5.3.2 Configurable Items

The *Configurable Items* section contains a table defining the configuration parameters that can be set when the logical component is instantiated. The parameters are modeled as role attributes (*diversity attributes*) that are defined in the *Roles* chapter. Attributes referred to in this table may be constant as well as variable. If the attribute is variable, the value that is provided at instantiation time is interpreted as the initial value of the attribute. So, the initial value of the attribute is the configuration parameter, and not its current value (which may change).

Note that API specifications only specify *which* parameters can be set at instantiation time and not *how* they are set (e.g. by compile-time variables, properties in a database, parameters of an instantation function, etc.). The mechanism used to set the configuration parameters is implementation-dependent.

Example (Configurable Items Table)

The *Configurable Items* table of the *Analog Audio Decoding* API specification indicates that there are two parameters that can be set at instantiation time: the number of selectors in the audio decoder/selector and the set of audio standards supported by the decoder/selector. In the specification these *parameters* are attributes of roles:

Configurable Items

Role	Attribute
`AnaAdec`	nrSelectors
	supportedStandards

End of Example (Configurable Items Table)

5.3.3 Constraints

The *Constraints* section specifies constraints that apply to the optional interfaces and configurable items. These constraints can be viewed as the *precondition* of the instantiation operation.

Example (Configuration Parameter Constraints)

The constraints that apply to the configurable items from the previous example are specified in the *Analog Audio Decoding* API specification as follows:

Constraints

- `nrSelectors` >= 1
- `supportedStandards` != 0

End of Example (Configuration Parameter Constraints)

5.4 Instantiation

Instantiation refers to the process of creating *instances* of logical components. A logical component instance is not a single object (in the object-oriented sense), but an *aggregate* that may consist of several objects. This aggregate corresponds to the *grey box* in an interface-role model. The API specification makes no assumptions on the mechanisms used to create a logical component instance but specifies what the result of the instantiation of a logical component is. This result is specified in the *Instantiation* section which consists of two subsections, one specifying which objects are created and the second specifying the initial state of these objects.

5.4.1 Objects Created

The *Objects Created* section defines which objects are contained in a new instance of the logical component. This is done by means of a table that specifies for each object: its type (normally a role name), its name and its multiplicity. The multiplicity *n* of an object *x* indicates how many instances of the object occur in the logical component instance. If *n* > 1 then *x* is interpreted as an *array* of objects of size *n*. The individual objects in the array are indicated as usual: *x*[0], *x*[1], ,,,, *x*[*n* - 1]. The entries in the **Object** column in the table act as global names that may be referred to in the rest of the specification.

Example (Objects Created)

The *Objects Created* section from the *Analog Audio Decoding Selection* API specification specifies that a newly created instance of the logical component will consist of 1 instance `dec` of the `AnaAdec` role and `dec.nrSelectors` instances of the `AnaAdecSelector` role (where `nrSelectors` is an attribute of the role `AnaAdec`).

Objects Created

The following objects are created when the logical component is instantiated:

Type	Object	Multiplicity
`AnaAdec`	`dec`	1
`AnaAdecSelector`	`sel`	dec.nrSelectors

End of Example (Objects Created)

5.4.2 Initial State

The *Initial State* section specifies properties of the objects declared in the previous section that are true in the initial state of the logical component instance, i.e. immediately after the instantiation of the logical component. Properties that follow from the diversity constraints specified in Section 5.3.3 are not repeated here, so to get the full specification of the initial state of the logical component instance that information should be added. Note that certain aspects of the initial state may have been deliberately left unspecified, e.g. to create implementation freedom.

Example (Initial State)

The *Initial State* section of the *Analog Audio Decoding* API specification is given below:

Initial State

The following constraints apply to the initial state of a logical component instance:

- `dec.pinIds == { uhAnaAdec_AudioOutBase + i | 0 <= i < nrSelectors }`
- For each `i` in `{ 0 .. nrSelectors - 1 }`
    - `dec.pinObjects[uhAnaAdec_AudioOutBase + i] == sel[i]`
- `dec.allowedStandards == supportedStandards`
- `dec.suggestedStandards == supportedStandards`
- `dec.detectedStandard == 0`
- For each `i` in `{ 0 .. nrSelectors - 1 }`
    - `sel[i].nicamPreference == uhAnaAdec_NicamPreferredOnlyIfRelated`
    - `sel[i].2ndProgramPreferred == False`
    - `sel[i].forcedMono == False`

Remarks

- The constant attributes `pinIds` and `pinObjects` are inherited from the `PinObjectsProvider` role defined in [5] which specifies no values for these attributes. This allows the values of these attributes to be specified above. Note that a (pin objects) client can access the selectors by using the `uhIPinObjects` interface.
- The initial values of the `1stProgram`, `2ndProgram`, `auxProgram` and related attributes of a decoder and the `selectedProgram` attribute of a selector have been deliberately left unspecified.

End of Example (Initial State)

5.5 Execution Constraints

In the *Execution Constraints* section execution constraints that apply to the logical component as a whole are specified. In particular, global constraints with respect to the multi-threading aspects of the logical component are specified here, e.g. that clients may not concurrently access any function in any interface provided by any role that is part of the logical component. These constraints are *not* repeated in the other chapters that follow.

Example (Execution Constraints)

The *Execution Constraints* section of the *Analog Audio Decoding* API specification states that the logical component as a whole is essentially single-threaded except for some special (get-)functions marked as *thread-safe*.

Execution Constraints

The interfaces provided by a decoder-selector may not be accessed concurrently from different threads. The only exceptions are those functions that have the qualifier *thread-safe*.

End of Example (Execution Constraints)

Chapter 6 The Roles

6.1 General

The *Roles* chapter specifies all *new* roles used in the API specification, i.e. all roles that are not already specified in one of the base documents. Each role is specified in a separate section that has the name of the role as its title. A role represents *behavior* associated with one or more interfaces. Which (*provides* and *requires*) interfaces are associated with a role can be seen in the interface-role model.

Roles are a means to define the *observable behavior* of the logical component, i.e. the behavior that can be observed at the external software and streaming interfaces of the logical component. What is observable and what is not is defined by the boundaries of the *grey box* in the interface-role diagram.

The behavior associated with a role can be divided into four aspects (see [1]):

1. *Function behavior:* the behavior of the functions in the interfaces provided by the role.
2. *Streaming behavior:* the input-output behavior of the streams associated with the role.
3. *Active behavior*: autonomous behavior that is visible at the *provides* and *requires* interfaces associated with the role.
4. *Instantiation behavior*: behavior displayed at instantiation time of the logical component.

Only the streaming and active behavior are specified in the *Roles* chapter. Instantiation behavior is specified in the *Logical Component* chapter and function behavior is specified in the *Interfaces* chapter; see Figure 6-1.

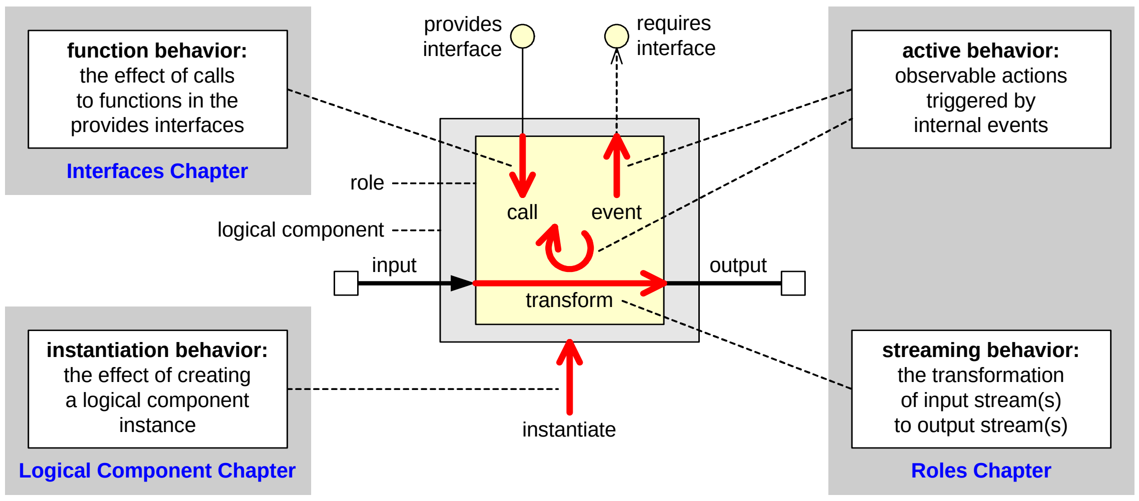
function behavior:
the effect of calls to functions in the provides interfaces
Interfaces Chapter
provides interface
requires interface
active behavior:
observable actions triggered by internal events
role
logical component
call
event
input
output
transform
instantiate
instantiation behavior:
the effect of creating a logical component instance
Logical Component Chapter
streaming behavior:
the transformation of input stream(s) to output stream(s)
Roles Chapter

Figure 6-1: Role Behavior and the Chapters Where it is Specified

Note that roles are pure specification artefacts. When implementing a logical component the only requirement is that the observable behavior of the logical component is consistent with the observable role behavior.

Note also that role behavior is realized dynamically by *instances* of the role, i.e. by objects *playing* the role. So a role should not itself be viewed as an object, but rather as a class that may have several instances at run-time. From a contractual point of view, the specification of the role can be viewed as the set of rights and obligations to be satisfied by the code of that class.

6.2 Role Specifications

A role specification consists of the following sections, some of which are optional:

1. **Signature**

   The *pseudo-IDL* definition of the role which defines the *attributes* associated with the role. Attributes can be viewed as variables representing the *abstract state* of a role instance. They play an essential role in the model-oriented style of specification used in the API specifications. Attributes and the pseudo-IDL used to define them are discussed in more detail in Section 6.3.

2. **Qualifiers**

   The qualifiers associated with the role (see Section 1.3). An example of a frequently used qualifier is *actor* which is explained in Section 6.7.

3. **Description**

   A short informal description of the role.

4. **Independent Attributes**

   Contains a table providing a short informal description of each *independent* attribute. An independent attribute is an attribute whose value may in principle vary freely. More details can be found in Section 6.4.

5. **Dependent Attributes**

   Contains a table providing a short informal description of each *dependent* attribute. A dependent attribute is an attribute whose value can be expressed in terms of other entities, in particular the values of independent attributes. The description of the dependent attribute includes a definition of the value of the attribute; see Section 6.4.

6. **Invariants**

   Provides a list of invariants. An invariant is an assertion about the role that is *always* true from the external observer's point of view. The assertion is typically formulated in terms of the attributes of the role. In reality, the assertion may temporarily be violated.

7. **Instantiation**

   Provides a description of how instances of the role are created. Usually instances of a role are created at instantiation time of a logical component only. In some cases instances of a role can also be created dynamically *after* the instantiation of the logical component, in particular by using interface functions that act as "constructors" of the role. In that case a list of the names of these constructors is provided.

8. **Streaming Behavior**

   A description of the streaming behavior of the role, i.e. the input-output behavior of the streams associated with the role. Streaming behavior is discussed in more detail in Section 6.5.

9. **Active Behavior**

   A description of the active behavior of the role, i.e. autonomous behavior that is visible at the interfaces associated with the role. Active behavior is discussed in more detail in Section 6.6.

10. **Execution Constraints**

    Defines concurrency-related constraints that apply to the role, in particular to the collection of interfaces associated with the role. These constraints are *not* repeated in the specifications of these interfaces.

11.Remarks

Contains a list of remarks related to the role.

6.3 Role Signatures

The *role signature* can be viewed as the *IDL* of the role. It defines the structure of a role in the same way that IDL defines the structure of an interface. More in particular, the role signature defines which roles are *specialized* by the role and which *attributes* are associated with the role. From a contractual point of view a role *S* specializing a role *R* amounts to *S* inheriting all rights and obligations specified for *R*, including all attributes of *R*.

Attributes can be viewed as variables representing the *abstract state* of a role instance. The behavior of the logical component (see Figure 6-1) is specified in terms of this abstract state. Typically, external calls of interface functions will change the values of attributes and thereby influence the streaming and active behavior of a role. Conversely, the values of attributes may change autonomously as a consequence of the streaming and active behavior, which in turn may influence the behavior of other roles.

Attributes are introduced for modeling purposes only and should *not* be confused with implementation variables. They correspond to UML attributes but are not indicated in the interface-role model (which is an extended UML class diagram).

Note that attributes inherited from *base roles* (roles defined in base documents) are not re-declared in the signature. The optional `const` keyword indicates that the value of an attribute is constant during the lifetime of a role instance. It does not imply that the value of the attribute is the same for different role instances.

Example (Base Role Signature)

In the specification of the *Scan Rate Conversion* logical component the state of the component is modeled by the attributes of the `ScanRateConv` role which is defined by the signature below. As can be inferred from the signature the role does not inherit attributes from other roles, otherwise there would be a *specializes* clause (see next example).

Signature

```
role ScanRateConv {
    const UInt32                 nrSuppInputFormats;
    uhScanRateConv_InputFormat_t  suppInputFormats[nrSuppInputFormats]
    scanMode                      actualMode;
    scanMode                      preferredMode[nrSuppInputFormats];
    scanMode                      fallbackMode[nrSuppInputFormats];
    scanModeList                  supportedModes[nrSuppInputFormats];
    uhScanRateConv_InputFormat_t  inputFormat;
    uhScanRateConv_FieldRate_t    slaveFieldRate;
    uhScanRateConv_ScanType_t     slaveScanType;
    Bool                          demoEnabled;
    Bool                          demoAvailable;
    uhScanRateConv_DemoMode_t     demoMode;
    uhScanRateConv_DemoModeSet_t  supportedDemos
    }
```

End of Example (Base Role Signature)

Example (Derived Role Signature)

In the specification of the *Analog Audio Decoding* logical component the state of the decoder-part of the component is modeled by the attributes of the `AnaAdec` role which is defined by the signature below. As implied by the signature below, the `AnaAdec` role *specializes* or *derives from* the `PinObjectsProvider` role defined in [5] which implies that it inherits all attributes of the `PinObjectsProvider` role. These attributes are not repeated in the signature of `AnaAdec`.

Signature

```
role AnaAdec specializes PinObjectsProvider {
    const UInt32                          nrSelectors;
    const uhAnaAdec_SoundStandardSet_t    supportedStandards;
    uhAnaAdec_SoundStandardSet_t          allowedStandards;
    uhAnaAdec_SoundStandardSet_t          suggestedStandards;
    uhAnaAdec_SoundStandard_t             detectedStandard;
    AudioProgram                          1stProgram;
    AudioProgram                          2ndProgram;
    AudioProgram                          auxProgram;
    Bool                                  related;
    uhAnaAdec_AudioProgramInfoSet_t       audioProgramInfo;
    uhAnaAdec_DolbySurroundInfo_t         dolbySurroundInfo;
}
```

End of Example (Derived Role Signature)

6.4 Independent and Dependent Attributes

The attributes associated with a role are divided into two types: those whose value may be defined or changed independently of all other attributes and those whose value is a function of the values of other attributes. These two types are referred to as *independent* and *dependent* attributes. The difference makes sense because in specifying state transitions only the changes to the values of the independent attributes have to be specified; the values of the dependent attributes follow automatically.

Independent attributes can be viewed as *variables* whose value can be modified. The independent attributes are listed in a table containing the name of each independent attribute and a short description of the meaning of the attribute.

Example (Independent Attributes)

The independent attributes of the `ScanRateConv` role from the *Scan Rate Conversion* logical component are described in the *Independent Attributes* section reproduced below:

Independent Attributes

Name	Description
`nrSuppInputFormats`	The number of supported video input formats (see `uhScanRateConv_InputFormat_t`).
`suppInputFormats`	Array containing the supported input formats.
`preferredMode`	Array containing preferred scan modes for every supported input format. An element at index `x` of this array is the preferred scan mode of the input format in the array `suppInputFormats` at index `x`.
`fallbackMode`	Array containing fallback scan modes for every supported input format. An element at index `x` of this array is the fallback scan mode of the input format in the array `suppInputFormats` at index `x`. A fallback scan mode for a certain input format is supported for every use-case. So, it can always be used when the preferred scan mode is unavailable.
`supportedModes`	Array containing lists with supported scan modes for every supported input format. An element at index *x* of this array is the list with supported scan modes of the input format in the array `suppInputFormats` at index `x`.
`demoEnabled`	Indicates whether the demonstration mode is enabled (`True`) or disabled (`False`).
`demoMode`	The current demonstration mode.
`supportedDemos`	Vector with the supported demonstration modes.

End of Example (Independent Attributes)

The value of a dependent attribute can be expressed in terms of other entities, in particular the values of independent attributes. Dependent attributes can be viewed as parameterless *functions* whose value can be computed.

Dependent attributes are listed in a separate table similar to the dependent attributes. The difference is that the table also defines the value of each attribute (as part of the *description* of the attribute).

Sometimes it is known that an attribute is dependent on other entities without knowing what the exact dependencies are. This happens particularly with attributes that model signal properties; their value depends on the contents of the signal. In these cases the value of the attribute is only defined informally.

Dependent attributes are sometimes called *auxiliary* or *convenience* attributes. We can in principle do without them by repeating the expression defining the value of a dependent attribute every time we have to refer to that value, but the appropriate use of dependent attributes has a positive effect on both the size and the maintainability of an API specification.

Example (Dependent Attributes)

The dependent attributes of the `ScanRateConv` role from the *Scan Rate Conversion* logical component are described in the *Dependent Attributes* section reproduced below:

Dependent Attributes

Name	Description
`actualMode`	The actual scan mode that is used by the Scan Rate Converter. Normally, the actual mode is equal to the preferred scan mode for the current input format. However, in some use-cases, the preferred scan mode is unavailable due to resource issues. The actual mode will then be equal to the fallback scan mode.
`inputFormat`	The current video input format.
`slaveFieldRate`	This is the field rate that must be followed in case the field rate of the current scan mode is set to slave mode. The value of this attribute depends on settings in other parts of the platform.
`slaveScanType`	This is the scan type that must be followed in case the scan type of the current scan mode is set to slave mode. The value of this attribute depends on settings in other parts of the platform.
`demoAvailable`	Indicates whether the current demonstration mode is available (`True`) or not (`False`). A demonstration mode can become unavailable when the input format changes to a format that can not be handled by the demonstration mode.

End of Example (Dependent Attributes)

6.5 Streaming Behavior

API specifications are meant to specify *software interfaces*, i.e. interfaces that can be used by software developers to develop applications on top of the Nexperia Home platform. Many of the UHAPI interfaces are streaming-related; they are used to control or provide information about streaming functionality. There is no way such interfaces can be defined without referring to streaming functionality. On the other hand, it is not the purpose of the UHAPI specifications to provide detailed specifications of the streaming itself. The API specifications define software interfaces that are common to all Nexperia Home platform instances. In a concrete platform instance these specifications are augmented with additional information concerning performance, resource usage, streaming algorithms used, etc.

The purpose of the *Streaming Behavior* section is to define the streaming behavior associated with the role at a level of abstraction that is sufficiently concrete to define the effect of the functions in the interfaces and that is sufficiently abstract to make the interface generally applicable to different Nexperia Home platform instances. In certain special cases this may mean that streaming behavior is specified in detail, but in most cases streaming behavior is modeled abstractly with a reference to standards for further details. The description of streaming behavior starts in the interface-role model diagram where the input and output streams are indicated, together with possible internal streams. The input and output streams are *weakly typed*. The description of how an input stream is transformed into an output stream is generally informal, where role attributes may be used to represent stream content.

Example (Streaming Behavior)

From the *Scan Rate Conversion* API specification:

Streaming Behavior

The `ScanRateConv` converts the scan rate of the video stream at the input into the requested output scan rate using the actual scan mode. In most cases, the actual scan mode is equal to the preferred scan mode for the current input format as set by the client. When the preferred scan mode is temporally unavailable because of system limitations, then the fallback scan mode for the current input format will become the actual scan mode.

In case the field rate and / or scan type of the scan mode are slaved, then the converter uses the field rate and / or scan type as prescribed by the platform.

When a demonstration mode is enabled, the actual scan mode is overruled until the demonstration mode is disabled again or when the demonstration is temporally unavailable due to an input format change.

End of Example (Streaming Behavior)

6.6 Active Behavior

Besides autonomously performing streaming functions, instances of roles may also autonomously perform software functions. A typical case is the notification of events detected in input streams. Autonomous behavior is often specified in terms of an *event-action table* that connects events that may occur to actions performed by a role instance. The events can be streaming-related events but also software-related events such as timer events. The actions can be calls to functions in *requires* interfaces or modifications of attributes. Roles which have no active behavior are *passive*; they do not actively influence the behavior of other roles.

Note that even though event notification is conceptually part of the active behavior associated with a role, most event notifications supported by a role are not listed in the event-action table. The reason is that the most common form of event behavior, an event leading to a notification of all *subscribers* of the event, is already specified in the general notification specification [6]. This behavior is not repeated for every single event in the event-action table. The description of the event and the conditions under which the event is raised are described in the specification of the corresponding callback function. Some events are special in that they may have additional effects besides notifying clients. Only these non-standard event notifications together with other autonomous behavior of role instances are specified in the *Active Behavior* section.

Example (Active Behavior)

The active behavior of the `GfxLayer` role from the *Video Mixing* API specification consists of three ingredients:

1. The active behavior inherited from the `Layer` role.
2. The standard notification of `GfxLayer`-specific events.
3. Special event-driven behavior.

Only the third ingredient is specified in the *Active Behavior* section by means of an event-action table:

Active Behavior

Besides the active behavior inherited from `Layer`, a `GfxLayer` has standard notification behavior (see `uhIVmixGfxLayerNtf` for the notified events) and the following event-driven behavior:

Event	Action
A new output composition starts.	The `GfxLayer` performs the following actions: • `activeBuffer = nextBuffer` • `activeClut = bufferClut` • `Layer::activeColorKey = bufferColorKey`

End of Example (Active Behavior)

6.7 Actor Roles

In the interface-role model of a logical component roles are introduced not only for the objects that provide the functionality of the logical component but also for the objects that use the functionality, i.e. the *clients* of the logical component. The reason is that, from the contractual point of view, not only the providers but also the users of the logical component may have to satisfy certain obligations. The latter obligations are part of the contract and are associated with *client roles* that model the users of the logical component. Examples of client obligations are that a client has to provide a notification interface or that it should not call certain functions under certain conditions.

From the specification point of view, client roles are usually much simpler than the provider roles of a logical component. A client role is typically characterized by having no attributes and no streaming behavior while its active behavior consists of providing stimuli only, i.e. of calling functions in interfaces of the logical component (as indicated in the interface-role model), without any a priori assumptions on when these calls occur. The qualifier *actor* is used to indicate this type of role and the streaming and active behavior as well as the attribute sections are omitted in the specification. Furthermore, no assumptions are made on how instances of an actor role are created.

Example (Actor Role)

The client of *Video Mixing* functionality is modeled in the API specification by the `VideoMixingClient` role which is defined as follows:

VideoMixingClient

Signature

```
role VideoMixingClient {}
```

Qualifiers

- actor

Description

A `VideoMixingClient` represents a user of the interfaces `uhIVmix`, `uhIVmixLayer`, `uhIVmixColorKey`, `uhIVmixBorder`, `uhIVmixVidLayer` and `uhIVmixGfxLayer`.

End of Example (Actor Role)

Chapter 7 The Interfaces

7.1 General

The *Interfaces* chapter contains specifications of all interfaces that are part of the API. The bulk of the specification of an interface consists of the specifications of the individual functions that constitute the interface. Each interface is specified in a separate section that has the name of the interface as its title. The structure and contents of an interface specification are discussed in Section 7.2.

Each function in an interface is specified in a separate subsection of the interface specification. The structure and contents of a function specification are discussed in Section 7.3. The style of specification used is *extended pre- and postconditions* which is further explained in Section 7.4.

There are two types of interfaces: *control interfaces* and *notification interfaces*, which are dealt with slightly differently at the specification level. The differences between these two types of interface specifications are discussed in Section 7.5.

Some frequently occurring types of functions are dealt with in a special way. This includes some standard functions relating to event subscription (discussed in Section 7.6) and asynchronous functions (discussed in Section 7.7). Finally, in Section 7.8 we explain the “bullet notation” that is frequently used in function specifications.

Note that the specifications of the interfaces and the functions contained in them do not specify the full behavior of a logical component; see Figure 6-1.

7.2 Interface Specifications

An interface specification defines either a *new* interface or an interface (typically from a base document) that is *specialized* by the logical component. In the case of a new interface, the name of the section defining the interface is equal to the unqualified interface name, e.g. `uhIAnaAdec`. In the case of a specialized interface, the name of the role providing the interface is added as a prefix to the interface name, using “::” as a separator. For example, “`AnaAdec::uhIPinObjects`” can be read as “the interface `uhIPinObjects` as specialized by the role `AnaAdec`”. The reason for this convention is that there may be multiple roles specializing the same interface. Each of these specializations is described in a separate section.

An interface specification consists of the following sections, some of which are optional.

1. **Qualifiers**

   The qualifiers associated with the interface (see Section 1.3). An example of a frequently used qualifier is *callback* which marks the interface as a *callback interface*; the typical example of a callback interface is a notification interface (see Section 7.5).

2. **Description**

   A short informal description of the interface.

3. **Interface ID**

   The globally unique identifier associated with the interface.

4. **Execution Constraints**

   Defines concurrency-related constraints that apply to the interface, in particular to the collection of functions in the interface. These constraints are *not* repeated in the specifications of these functions.

5. **Remarks**

   Contains a list of remarks related to the interface.

6. **Function specifications**

   Specifications of the individual functions in the interface, each in a separate subsection. The standard functions inherited from the `uhIUnknown` interface [4] (`QueryInterface`, `AddRef` and `Release`) are omitted.

There is no separate section defining the *signature* (IDL representation) of the interface, because the signature can be generated automatically from the name of the interface, the interface ID and the signatures of the functions in the interface.

The above template is used for new as well as specialized interfaces. The only difference for specialized interfaces is that there are no specifications for those functions that are inherited *as is* from the base interface. Only those functions whose specification has changed by the specialization get a separate specification that overrides the old specification.

Example (New Interface Specification)

The specification of the `uhIAnaAdec` interface of the *Analog Audio Decoding* logical component is given below. The real information on the interface is in the function specifications at the end of the interface specification.

uhIAnaAdec

Qualifiers

None.

Description

This interface of a decoder allows the client to control the decoding of audio programs, to obtain information on the decoder input stream and to subscribe to events notified by the decoder. The client can control the decoding by restricting the sound standards to be used by the decoder and by suggesting the use of particular sound standards to speed up the detection of the actual sound standard.

Interface ID

```
uuid ( 5697C761-5E73-11d6-9A8B-00065B6400D5 )
```

<function specifications>

End of Example (New Interface Specification)

Example (Specialized Interface Specification)

The `uhIPinObjects` interface provided by the `AnaAdec` role of the *Analog Audio Decoding* logical component is not a new interface because `AnaAdec` inherits it from the `PinObjectsProvider` role from [5]. In the title of the section defining the interface it is therefore explicitly indicated that we refer to `uhIPinObjects` as provided by the `AnaAdec` role (and not the `PinObjectsProvider` role). No function specifications are overridden, so the specification of `uhIPinObjects` read:

AnaAdec::uhIPinObjects

Qualifiers

None.

Description

This interface of a decoder is a specialization of the `IPinObjects` interface as defined in [5]. The specialization is defined by the specific values of the pinIds and `pinObjects` attributes of the decoder. These attributes are inherited from `PinObjectsProvider` and their values are set at instantiation time.

Interface ID

See [5].

End of Example (Specialized Interface Specification)

7.3 Function Specifications

Except for one detail (explained in Section 7.4), the structure of a function specification is fairly standard. The style of specification can be characterized as *extended pre- and postconditions*. The specification of a function is subdivided into the following sections, some of which are optional:

1. **Signature**

   The IDL definition of the function which is essentially the C prototype of the function with some extra information, e.g. indicating which parameters are in-parameters and which ones are out-parameters.

2. **Qualifiers**

   The qualifiers associated with the function (see Section 1.3). These qualifiers are often execution-related. For example, the "single-threaded" qualifier indicates that the number of threads that may concurrently execute the method is at most one.

3. **Description**

   A short informal description of the function.

4. **Parameters**

   Short informal descriptions of the parameters of the function, arranged in a table.

5. **Return values**

   Short informal descriptions of the values that can be returned by the function, arranged in a table. In UHAPI control interfaces the return values are almost always error codes; functions in notification functions do not normally return values. If the function returns only standard error codes, the table is omitted and the keyword "Standard" is used instead.

6. **Precondition**

   Defines an assertion that should be true immediately before execution of the function starts. Contractually speaking, it is the obligation of the *caller* of the function to make sure this is the case.

7. **Action**

   Describes the "abstract action" performed by the *callee* when the function is called. This action typically indicates which role attributes are modified by the function and/or which out-calls are performed by the function. For further details see Section 7.4.

8. **Postcondition**

   Defines an assertion that is true immediately after execution of the function finishes. Contractually speaking, it is the obligation of the *callee* to make sure this is the case.

9. **Remarks**

   Contains a list of remarks related to the function.

Example (Function Specification)

The specification of the `BlankOutput` function of the `uhIVmix` interface of the *Video Mixing* logical component contains examples of all 9 types of sections that constitute a function specification:

BlankOutput

Signature

```
uhErrorCode_t BlankOutput (
    [in] Bool      blank,
    [in] uhColor_t blankColor );
```

Qualifiers

- synchronous
- single-threaded

Description

Blanks or unblanks the output of the video mixer. The blanking of the video mixer is independent of the blanking of individual layers. When the output is blanked the specified (RGB) blanking color is shown full screen. The parameter `blankColor` has no effect in case of an unblank (except that `GetOutputBlanked` will return this value).

Parameters

Name	Description
`blank`	Indicates whether the output has to be blanked (`True`) or unblanked (`False`).
`blankColor`	The blanking color.

Return Values

Standard.

Precondition

- `True`

Action

- if (*`blank`* `== True`)
  - Blank the screen with *`blankColor`* (on the next field).
  - Modify `mixer.blanked, mixer.blankColor, mixer.actualBlankColor`.
- if (*`blank`* `== False`)
  - Modify `mixer.blanked, mixer.blankColor`.
  - Unblank the screen (on the next field).

Postcondition

- `mixer.blanked ==` *`blank`*`.`

- `mixer.blankColor ==` *`blankColor`*`.`
- if `(`*`blank`*` == True)`
  - `mixer.actualBlankColor ==` *`blankColor`*`.`

Remarks

- This function is synchronous and lasts up to the next field to complete (when the screen is actually (un)blanked).
- Note that the screen is only unblanked if no auto blanking is currently active.

End of Example (Function Specification)

7.4 Preconditions, Actions and Postconditions

An inherent restriction of classical pre- and postcondition specifications is that they can only be used to specify constraints on the states *before* and *after* the execution of a function. They cannot be used to specify constraints on what should happen *between* these two states, i.e. during the execution of the function. A common approach is to allow *anything* to happen during the execution as long as the postcondition is not violated. This approach leads to underspecification of functions because we normally do not want a function to modify arbitrary variables or call arbitrary functions during its execution, even if the postcondition is met in the end. Furthermore, we sometimes want to express that certain observable actions *should* occur during the execution of a function, which is hard or even impossible to express in a postcondition. Typical examples of such actions are out-calls (such as callbacks) and synchronization actions (blocking).

The above problem is solved in API specifications by using extended pre- and postcondition specifications that allow observable behavior during the execution of a function to be specified in terms of an *abstract action*. The action is described in a way similar to ordinary code but using more abstract *non-deterministic* constructs that leave a lot of implementation freedom. The action is generally simple, though it can be used to specify more complex behavior as well, such as out-calls and synchronization.

A typical construct used in formulating abstract actions is **Modify *x***, where *x* is some attribute (see the example above). **Modify *x*** should be interpreted as "change the state of the logical component in such a way that no state variable other than *x* is modified". The new value of *x* is typically specified in the postcondition, where we do not have to specify that other variables have not been modified. In order to refer to the *old* value of *x* in the postcondition, the notation *x*' is used. So, in the example above, when the notation mixer.blanked' would have been used in the postcondition, it would refer to the value of `mixer.blanked` immediately before the call of the `BlankOutput` function. The **Modify** clause for an out-parameter of a function is omitted because the right to modify the value of the parameter is implicit in the *[out]* attribute associated with the parameter.

Each extended pre- and postcondition specification of a function can be interpreted as a *mini-contract* between the caller and callee of the function, as indicated in Figure 7-1.

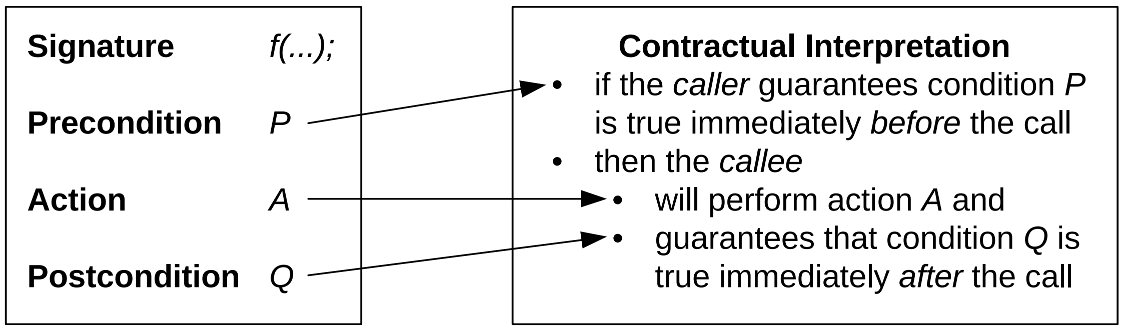
Signature f(...);
Precondition P
Action A
Postcondition Q
Contractual Interpretation
• if the *caller* guarantees condition *P* is true immediately *before* the call
• then the *callee*
• will perform action *A* and
• guarantees that condition *Q* is true immediately *after* the call

Figure 7-1: Contractual Interpretation of Extended Pre- and Postcondition Specifications

7.5 Control and Notification Interfaces

Generally speaking, a logical component provides two types of functionality: (streaming) control and (event) notification. The control functionality is provided by means of one or more *control interfaces* that are provided by the logical component and that allow a client to influence the behavior of the logical component by calling functions. The notification functionality is provided by one or more *notification interfaces* that are provided by the client and that allow the logical component to notify the occurrence of events to the client by calling functions.

Although the same layout is used for defining control and notification interfaces, they are dealt with differently at the specification level. The main reason for this is that, from the point of view of a logical component, a control interface is a *provides* interface and a notification interface is a *requires* interface. The specification of a control interface will define precisely what the effect of each function in the interface is. The specification of a notification interface, on the other hand, will usually not define what the effect of a notification function is; that is up to the client implementing the interface. Instead, the specification will describe *when* the function will be called, i.e. which event will cause the function to be called by the logical component.

Notification interfaces can be recognized by their name ending in `Ntf`. Typically, the name of a notification interface is equal to the name of the corresponding control interface extended with `Ntf`.

Example (Notification Interface)

The notification interface associated with the control interface `uhIAnaAdec` of the *Analog Audio Decoding* logical component is `uhIAnaAdecNtf`. Its specification is given below, where the specifications of the functions in the interface have been omitted. The fact that this is a notification interface can be derived from the name of the interface as well as the *callback* qualifier used in the specification.

uhIAnaAdecNtf

Qualifiers

- callback

Description

This is the notification interface to be provided by each client that wants to subscribe to notifications of events that occur in the decoder.

Interface ID

```
uuid( 5697C762-5E73-11d6-9A8B-00065B6400D5 )
```

<*notification function specifications*>

End of Example (Notification Interface)

Example (Notification Function)

The specification of the `OnInputChanged` notification function from the `uhIAnaAdecNtf` interface of the *Analog Audio Decoding* logical component is shown below. Notice that the function has no return value, that its precondition and postcondition are True and that its *Action* is specified as *Any callback-compliant action*. The latter means that the client providing the interface is free to implement it any way he likes provided that the interface satisfies the general constraints for callback functions, such as not blocking the caller of the function.

Rather than describing the (unknown) effect of the function, the *Description* part of the function specification defines *when* the function will be called, i.e. which event will trigger the function. Of course, the function in the client interface will only be called when the client has subscribed to the `InputChanged` event.

OnInputChanged

Signature

```
Void OnInputChanged (
    [in] UInt32 cookie,
    [in] uhAnaAdec_SoundStandard_t soundStandard,
    [in] uhAnaAdec_AudioProgramInfo_t progInfo );
```

Qualifiers

None.

Description

This function is called by a decoder whenever it has detected one of the following two changes in its input stream:

- loss of synchronization, either spontaneously (in particular due to bad reception conditions) or at the beginning of a (re-)tuning operation;
- detection of the sound standard after re-synchronization.

Parameters

Name	Description
`cookie`	The value that was passed by the client at subscription time.
`soundStandard`	The sound standard detected by the decoder. In the case of loss of synchronization, the value of this parameter will be 0.
`progInfo`	Information about the audio programs in the input stream of the decoder-selector.

Return Values

None.

Precondition

- `True`

Action

- Any callback-compliant action.

Postcondition

- `True`

Remarks

- Sound standard and audio program information are combined in this function, thus providing synchronous, consistent information on the input stream.

End of Example (Notification Function)

7.6 Subscribe, Unsubscribe and OnSubscriptionChanged

In order to make a logical component notify the occurrence of an event to a client, the client should first *subscribe* to the event. Each notification interface therefore has an associated control interface that among other functions contains two standard functions `Subscribe` and `Unsubscribe`. There is also one standard function `OnSubscriptionChanged` in each notification interface which is used to notify the client that its subscription has changed (because subscription is asynchronous). Because the specifications of these three functions have a standard pattern, their specifications are abbreviated using special qualifiers; see the examples below. For their full specification and other details of the notification functionality, see [6].

Example (`Subscribe` and `Unsubscribe` Functions)

The specification of the `uhIAnaAdec` interface of the *Analog Audio Decoding* logical component contains the specifications of the `Subscribe` and `Unsubscribe` functions shown below. The qualifiers *subscribe-function* and *unsubscribe-function* indicate that they are standard functions defined in the *Notification* API specification. These functions (with different types for the first and third parameters) occur in all control interfaces that have an associated notification interface.

Subscribe

Signature

```
uhErrorCode_t Subscribe (
    [in] uhIAnaAdecNtf *pINotify,
    [in] UInt32 cookie,
    [in] uhAnaAdec_NtfSet_t notifs );
```

Qualifiers

- subscribe-function

Unsubscribe

Signature

```
uhErrorCode_t Unsubscribe (
    [in] uhIAnaAdecNtf *pINotify,
    [in] UInt32 cookie,
    [in] uhAnaAdec_NtfSet_t notifs );
```

Qualifiers

- unsubscribe-function

End of Example (`Subscribe` and `Unsubscribe` Functions)

Example (`OnSubscriptionChanged` Function)

The specification of the `uhIAnaAdecNtf` interface of the *Analog Audio Decoding* logical component contains the specification of the `OnSubscriptionChanged` notification function shown below. The qualifier *onsubscriptionchanged-function* indicates that it is a standard function defined in the *Notification* API specification. This function (with a different type for the second parameter) occurs in all notification interfaces.

OnSubscriptionChanged

Signature

```
[async] Void OnSubscriptionChanged (
    [in] UInt32 cookie,
    [in] uhAnaAdec_NtfSet_t notifs );
```

Qualifiers

- onsubscriptionchanged-function

End of Example (`OnSubscriptionChanged` Function)

7.7 Asynchronous Functions

An asynchronous function can be viewed as a function with a *delayed effect*. The behavior of such a function can be separated into a synchronous part, i.e. the behavior between call and return of the function, and an asynchronous part, i.e. the behavior after returning from the function. Completion of the asynchronous action is usually reported by means of a notification allowing the caller of the function to synchronize with completion of the asynchronous effect of the function.

Because asynchronous functions occur frequently, a special extension of the pre-action-post format is used for specifying them. The synchronous part of an asynchronous function is specified in the same way as a synchronous function (except for the qualifier *asynchronous*). The asynchronous part is specified by means of an additional action-postcondition pair specifying the effect of the asynchronous action performed by the function.

The following notation is used in the *Asynchronous Action* clause to indicate the (asynchronous) act of notifying all subscribers to a specific event:

```
Notify OnEvent(*,...)
```

The asterisk refers to the observer-specific cookie and the ellipses to the event-specific data that is being passed.

The absence of the synchronous *Action* and *Postcondition* clauses in an asynchronous function specification are equivalent to the synchronous action being *None* and the synchronous postcondition being `True`. In that case the function does not have an immediate effect but a delayed effect only.

The asynchronous action that is the consequence of calling an asynchronous function is part of the active behavior of the logical component (see Section 6.5). Because this action is already specified in the asynchronous function specification, its specification is not repeated in the specification of the active behavior of the role that provides the function.

Example (Asynchronous Function)

The `SelectUseCaseX` function from the *Connection Management* logical component is an example of an asynchronous function. The pre- and postcondition part of its specification is given below. From the specification we can derive that calling `SelectUseCaseX` will have the immediate (synchronous) effect of changing the state of the connection manager from `UseCaseSelectionCompleted` to the state `UseCaseSelectionInProgress`. The connection manager will then asynchronously perform the transition to another use case. On completion it will change the state back to `UseCaseSelectionCompleted` and notify all subscribers to the `OnUseCaseSelected` event.

Precondition

- `connmgr.state == UseCaseSelectionCompleted`
- `(connmgr.currentUseCase,`*`UseCaseX`*`) in connmgr.allowedTransitions`
- The client has released all interfaces of components that are not part of *`UseCaseX`*.

Action

- Modify `connmgr.state`

Postcondition

- `connmgr.state == UseCaseSelectionInProgress`

Asynchronous Action

- For all `comp in connmgr.currentUseCase.compSet` and not(`comp in` *`UseCaseX`*`.compSet):`
  - destroy `comp`
- For all comp in *`UseCaseX`*`.compSet` and not(`comp in connmgr.currentUseCase.compSet):`
  - create `comp`
- `connmgr.currentUseCase = UseCaseX`
- `connmgr.state = UseCaseSelectionCompleted`
- Notify `OnUseCaseSelected(*)`

Asynchronous Postcondition

- connmgr.state == UseCaseSelectionCompleted

End of Example (Asynchronous Function)

7.8 Bullet Notation

Throughout the API specification bullets and indentation are used to structure possibly complex assertions and expressions and restrict the number of parentheses as much as possible. The items in a bullet list that represents an assertion (such as an invariant, precondition or postcondition) are conceptually connected by logical and operators. The items in a bullet list that represents an action are conceptually connected by sequential composition operators (semicolons, in C terms). Start and end of indentation logically introduce an opening and closing bracket, respectively.

Example (Bullet Notation)

A precondition such as:

Precondition

- *Assertion*$_1$
- If(*Assertion*$_2$)
  - *Assertion*$_3$
  - *Assertion*$_4$

is equivalent to:

Precondition

- *Assertion*$_1$ && If(*Assertion*$_2$) { *Assertion*$_3$ && *Assertion*$_4$ }

Likewise, an action clause such as:

Action

- *Action*$_1$
- *Action*$_2$
- Modify
  - *Variable*$_1$
  - *Variable*$_2$

is equivalent to

Action

- *Action*$_1$; *Action*$_2$; Modify { *Variable*$_1$, *Variable*$_2$ }

End of Example (Bullet Notation)

Appendix A Qualifiers Quick Reference

A.1 Type & Constant Qualifiers

Qualifier	Description
error-codes	Indicates that this is the definition of a group of constants that act as error codes.
enum-element	Indicates that this is the definition of an enumerated type whose values represent elements of sets. The values of this type are powers of two (*flags*) and can be used for constructing sets by means of logical OR-ing.
enum-set	Indicates that this is the definition of a type whose values represent sets of enumerated values (*bitvectors*). This type is *typedef*-ed as `UInt32`.
sub-type	Indicates that this is the definition of a type whose values are a subset of the values of another type. The former type is *typedef*-ed in terms of the latter type and the subset conditions (e.g. upper and lower bounds) are specified as constraints.

A.2 Role Qualifiers

Qualifier	Description
root	Indicates that this is the *root role* of the logical component. When querying the connection manager in a platform instance for an instance of this component, it will return a pointer to one of the interfaces provided by this role (provided an instance of this component exists in the current use case).
actor	Indicates that the active behavior of the role consists of providing stimuli only, i.e. of calling functions, without any a priori assumptions on when these calls occur. Typical examples are roles that model clients of control interfaces. Roles of this type typically have no attributes and no streaming behavior. The streaming and active behavior are omitted in the specification.

A.3 Interface Qualifiers

Qualifier	Description
callback	Indicates that all functions in the interface are callback functions, typically notification functions. The difference between a callback function and a *normal* function is that the API specification does not specify what the effect should be of the function (this is up to the client implementing the interface) but only specifies when and with what parameters the function is called. Callback functions usually return no values and have in-parameters only.
model-interface	Indicates that this is an interface introduced for modeling purposes only. The interface does not occur in the IDL of the API.

A.4 Function Qualifiers

Qualifier	Description
synchronous	Indicates that the effect of the function will have been completed on returning from the function.
asynchronous	Indicates that the effect of the function will not have been completed when returning from the function; the function has a *delayed effect*. Completion is usually reported by means of a notification.
thread-safe	Indicates that the function may be accessed concurrently by multiple threads.
single-threaded	Indicates that the function may be accessed by at most one thread at a time.
subscribe-function	Indicates that this is a standard `Subscribe` function as specified in the Notification API specification. Only the signature of the function is given.
unsubscribe-function	Indicates that this is a standard `Unsubscribe` function as specified in the Notification API specification. Only the signature of the function is given.
onsubscriptionchanged-function	Indicates that this is a standard `OnSubscriptionChanged` function as specified in the Notification API specification. Only the signature of the function is given.
model-function	Indicates that this is a function introduced for modeling purposes only. The function does not occur in the IDL of the API.

Date of release: 30 July 2004