CELF Meeting
January 26th

UHAPI

John Vugts
Agenda

- Introduction
- UHAPI Scope
- UHAPI Characteristics
- UHAPI Concepts
- UHAPI Specification structure
- Overview of available Logical Components
- Walk through Logical Component
- Example Application code snippets
- Summary
UHAPI Scope
UH Scope core and non core AV

- Applications
- Middleware services
- Core AV
- Customer AV
- Native OS Services

APIs:
- MIDDLEWARE API
- UNIVERSAL HOME API
- OS & CONNECTIVITY API
UHAPI is an API to the MW

- Focus on runtime control by the Middleware (ISV).
- It does not specify e.g. streaming interfaces.
- Specification structure deals with diversity.
UHAPI Characteristics
UHAPI Characteristics (1)

- **API family for AV functionality (analog, digital)**
- **Designed with a middleware view in mind**
  - Domain model supports a large set of middlewares
- **Provides a consistent, orthogonal, coherent set of interfaces**
  - Well defined (syntax and semantics)
  - Easy to understand / debug by the users (functional interface)
  - Minimizes support costs
    - Fewer people involved (aspects touched) if one aspect changes
    - Reduced maintenance costs
  - Subset of provided interfaces can be used to manage diversity
  - Dependency between middleware and platform is very explicit

- **Binary Interface**
  - Binary releases
  - Enables partial downloads (dynamic binding)
**UHAPI Characteristics (2)**

- Hardware and implementation independent interface
  - Allows freedom in implementation and evolution
  - Support both HW and SW streaming
  - Support both on and off chip peripherals
  - Does not expose the physical software component architecture

- Processor independent

- Used processor transparent to client
  - Support efficient RPC implementation
UHAPI Characteristics (3)

- **Uses standard mechanisms**
  - Notification (runtime binding)
  - Error handling
  - Connection management (simple to program)

- **Uses standard COM like mechanisms**
  - IUnknown
    - QueryInterface
    - AddRef & Release
  - uhCom_CreateInstance
  - v-tables
  - GUIDs

- **“Interface grouping” and diversity mechanisms**
  - Interfaces
  - Roles (group of interfaces)
  - Logical Components (group of roles)
  - Platform instance (group of logical components) “Product”
UHAPI Concepts
Concepts Agenda

- Logical components
- Connection management
- Framework v.s. platform instance
  - Diversity elements
- Interface technology
  - 3rd party binding
- Interface navigation
- Notifications
- Error handling
  - Strong typing
- Execution architecture
Concepts Agenda

- Logical components
  - Logical v.s. physical
- Connection management
  - 3rd party binding
- Framework v.s. platform instance
  - Diversity elements
- Interface technology
- Interface navigation
- Notifications
- Error handling
- Execution architecture
Logical v.s. Physical components

- **Logical components (part of the UHAPI spec):**
  - Specification entities.
  - Used to compose a logical model.
  - Terminology familiar to the clients of the UHAPI.
  - Related to interface suites.
  - One control aspect for the MW (e.g. Tuner, ATSC dec, Video feat.)

- **Physical components (NOT part of the UHAPI spec):**
  - Implementation entities.
  - Used to implement the functionality required.
  - Terminology familiar to domain experts
  - Mapped onto logical components, but not necessarily one to one.
  - One implementation aspect for the platform (e.g. SW streaming FW).

- **This to enable HW independence**
Logical v.s. Physical components

Logical components exist only in user documentation. It includes behavior specification.

The specified behavior is in the Physical components.

Physical Components located in “comps” directory of your SDE.
Concepts Agenda

- Logical components
  - Logical v.s. physical

- Connection management
  - 3rd party binding

- Framework v.s. platform instance
  - Diversity elements

- Interface technology

- Interface navigation

- Notifications

- Error handling

- Execution architecture
Connection Management

- A UHAPI client selects the streaming setup of the platform instance by selecting a use-case.

```c
err = gpConnMgr->SelectUseCase(uhConnMgrSdkDemo_UcSingleWindow);
```

- The connection manager creates all streaming components and connects them (initializes system).

- A client initially obtains a logical component interface.

```c
err = gpConnMgr->GetInstance(uhConnMgrSdkDemo_CVmix, UHIID_uhIVmix, &gpVmix);
```

- It abstracts the client from the Physical Components that need to be connected
Connection Management

- It relieves the client from difficult and HW specific task of setting up components (priorities, buffer sizes etc.)
- It handles transients in initializing the platform.

- The connection manager is platform instance specific.
- Use cases are NOT defined by UHAPI
- Systems will exist that have only one use case (initialize).

```c
err = gpConnMgr->SelectUseCase(uhConnMgrSdkDemo_UcSingleWindow);
err = gpConnMgr->GetInstance(uhConnMgrSdkDemo_CVmix, UHIID_uhIVmix, &gpVmix);
```
Concepts Agenda

- Logical components
  - Logical v.s. physical

- Connection management
  - 3rd party binding

- Framework v.s. platform instance
  - Diversity elements

- Interface technology
- Interface navigation
- Notifications
- Error handling
- Execution architecture
Universal Home Application Programming Interface

**UHAPI is an API framework to the MW**

- **Platform independent code**
- **Platform dependent code**

**Platform Instance specification**

- Platform Instance specification A
- Platform Instance specification B

- Products A
- Products B

Philips Digital Networks - NextWorks

UHAPI Forum
UHAPI Specification Diversity

- **Platform Instance document specifies:**
  - Which logical components (instances) are supported
  - Which use cases are supported
  - Diversity options of the Logical Components are specified:
    - Availability of optional interfaces
    - Parameter ranges
    - Available resources (e.g. number of section filters)
    - Other e.g. which standards are supported
  - Resource usage figures (CPU cycles, memory, memory bandwidth, algorithms used, ...)

- This all supports different hardware, and make the interfaces hardware independent.
Portability: Isolate platform dependencies

SelectUseCaseYyyy(p1, ...);
GetInstance(UHCMG_DEMUX1, UHIID_uhIDmxCapability, &pDmx);
Bind(idx, UHIID_uhIDmxCapability, pDmx);
SetFilterParam(...)

Client software

Connection manager
Tuner
DMX
Video Decoder
Mixer

1
2
3
4
Diversity, platform independence

// Get current brightness value
err = gpVfeatBcs->GetBrightness(&gBrightness);
assert (err == UH_OK);

// Get brightness range
err = gpVfeatBcs->GetBrightnessRange(&gBrightnessMin, &gBrightnessMax);
assert (err == UH_OK);

// Interpret the value
// (gBrightness - gBrightnessMin) / (gBrightnessMax - gBrightnessMin)

// Increase brightness by 10%, assume at least 10 steps
assert(gBrightnessMax - gBrightnessMin + 1 >= 10);
gBrightness = gBrightness + (gBrightnessMax - gBrightnessMin) / 10;

// Clip the value
if (gBrightness > gBrightnessMax)
gBrightness = gBrightnessMax;

// Set the value
err = gpVfeatBcs->SetBrightness(gBrightness);
assert (err == UH_OK);
Concepts Agenda

- Logical components
  - Logical v.s. physical
- Connection management
  - 3rd party binding
- Framework v.s. platform instance
  - Diversity elements
- Interface technology
- Interface navigation
- Notifications
- Error handling
- Execution architecture
A client subscribes a notification interface using the `Subscribe` methods on the corresponding control interface (dynamic binding).

Subscriptions are changed using bit masks.

Multiple clients can subscribe to a control interface.

A client can subscribe its notification interface at multiple control interfaces. A cookie can be used to distinguish between the various notifications.

Notification interfaces have a separate method for each event.

This enables efficient use of HW in CE devices.
Concepts Agenda

- Logical components
  - Logical v.s. physical
- Connection management
  - 3rd party binding
- Framework v.s. platform instance
  - Diversity elements
- Interface technology
- Interface navigation
- Notifications
- Error handling
- Execution architecture
Execution Architecture

- **Logical Components are in principle single threaded**
  - Allows for simple implementation
  - In general it is not required to make everything thread safe
  - Where required it is multi-thread safe

- **Get methods are by default thread-safe even when the interface they are part of is not thread-safe.**
  - Easy to implement (stop scheduling or interrupts)

- **Access to an interface is by default single-threaded.**
  - To keep it as simple as possible

- **Notification methods are serialized**
  - This makes the client implementation simpler (single threaded)

- **Notification methods are not allowed to block.**
Execution Architecture (2)

- **Thread safeness can be specified on:**
  - Method level
  - Interface level
  - Role level
  - Logical Component level

- **Single client view requires client to synchronize, but this is required anyway**
  - What would be the semantics of multiple threads calling a set method?
Speciﬁcation views

Interfaces Chapter

function behavior: the effect of calls to functions in the provides interfaces

Logical Component Chapter

instantiation behavior: the effect of creating a logical component instance

Roles Chapter

active behavior: observable actions triggered by internal events

streaming behavior: the transformation of input stream(s) to output stream(s)
Overview of available Logical Components
## UHAPI 1.0 contents

<table>
<thead>
<tr>
<th>General documents (7)</th>
<th>Decoders/Encoders (5)</th>
<th>Audio Processing Components (10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Specification Reader's Guide</td>
<td>ATSC Decoder</td>
<td>Audio Automatic Volume Leveling</td>
</tr>
<tr>
<td>API Naming Conventions</td>
<td>Image Decoding</td>
<td>Audio Bass Enhancements</td>
</tr>
<tr>
<td>Error Handling</td>
<td>SPDIF Decoding</td>
<td>Audio Dynamic Range Control</td>
</tr>
<tr>
<td>Execution Architecture</td>
<td>STC Decoding</td>
<td>Audio Mixing</td>
</tr>
<tr>
<td>Notification</td>
<td>Transport Stream Demultiplexing</td>
<td>Audio Noise Generation</td>
</tr>
<tr>
<td>Qualifiers Quick Reference</td>
<td></td>
<td>Audio Program Selection</td>
</tr>
<tr>
<td>API Evolution Rules</td>
<td></td>
<td>Audio Volume Control</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type specifications (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Types</td>
</tr>
<tr>
<td>Global Types</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>API specifications (50)</th>
<th>Video Processing Components (15)</th>
<th>Generic (8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front End Components (12)</td>
<td>Ambient Level</td>
<td>Analog AV Output</td>
</tr>
<tr>
<td>Analog Audio &amp; Video Demodulation</td>
<td>Analog Video Decoding</td>
<td>SPDIF-out</td>
</tr>
<tr>
<td>Analog AV Input</td>
<td>Analog Video Encoding</td>
<td>Connection Management</td>
</tr>
<tr>
<td>Anti Aging</td>
<td>Analog Video Encryption</td>
<td>Fatal Error Handling</td>
</tr>
<tr>
<td>Analog Audio Decoding</td>
<td>Basic Video Featuring</td>
<td>I am Alive</td>
</tr>
<tr>
<td>Channel Decoding</td>
<td>Black Bar Detection</td>
<td>Pin Objects</td>
</tr>
<tr>
<td>RF Amplification</td>
<td>Color Transient Improvement</td>
<td>Unknown</td>
</tr>
<tr>
<td>Out Of Band Tuning &amp; Demodulation</td>
<td>Dynamic Noise Reduction</td>
<td>URL Source</td>
</tr>
<tr>
<td>Signal Strength</td>
<td>Histogram Modification</td>
<td></td>
</tr>
<tr>
<td>Tuning</td>
<td>Noise Measurement</td>
<td></td>
</tr>
<tr>
<td>Hdmiln</td>
<td>Scan Rate Conversion</td>
<td></td>
</tr>
<tr>
<td>SPDIF-in</td>
<td>Sharpness Enhancement</td>
<td></td>
</tr>
<tr>
<td>VBI Slicing</td>
<td>Sharpness Measurement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Video Color Enhancement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Video Mixing</td>
<td></td>
</tr>
</tbody>
</table>
Walk through Logical Components

ATSC Decoder
Video Mixer
Example Application code snippets

C++ example
Main usage is in plain C
Any language is possible
Explicit dependencies on interfaces

```c
#include "uhIVmix.h"
#include "uhIVmixLayer.h"
#include "uhIVmixVidLayer.h"
#include "uhIVmixGfxLayer.h"
```
Bring up the platform

```cpp
err = uhPlatformInit();
assert (err == UH_OK);

// Create a connection manager instance
std::cout << "Creating Connection Manager Instance" << std::endl;
err = uhComCreateInstance(UHCLSID_uhConnMgrSdkDemo, UHIID_uhConnMgrSdkDemo,
&gpConnMgr);
assert (err == UH_OK);

// Select the "single window" Use Case
std::cout << "Selecting Single Window Use Case" << std::endl;
err = gpConnMgr->SelectUseCase(uhConnMgrSdkDemo_UcSingleWindow);
assert (err == UH_OK);
```
Get Video Mixer, and set the background

// Get the current Use Case's Vmix interface
err = gpConnMgr->GetInstance(uhConnMgrSdkDemo_CVmix, UHIID_uhIVmix, &gpVmix);
assert (err == UH_OK);

// Set the background colour of the Vmix
colour.redOrY = 30;
colour.greenOrU = 144;
colour.blueOrV = 255;
err = gpVmix->SetBgColor(colour);
Set the windows of the Video Mixer

```c
// Select entire input
inputRect.ul.x = 0;
inputRect.ul.y = 0;
inputRect.lr.x = gVidLayerMaxWidth;
inputRect.lr.y = gVidLayerMaxHeight;
err = gpVmixLayerV->SetSrcWindow(inputRect);
assert (err == UH_OK);
...

err = gpVmixLayerV->SetLayerWindow(layerRect);
assert (err == UH_OK);

err = gpVmixLayerV->SetDstWindow(destRect);
assert (err == UH_OK);

err = gpVmixLayerV->ActivateNewWindowSettings(True, 1000);
assert (err == UH_OK);
```
Summary
Summary

- UHAPI is complete, consistent and supports entire CE products
- UHAPI documentation is of high quality
- UHAPI is platform independent
- Member companies add tools and training
- Download UHAPI at www.uhapi.org
Questions?

Thanks for your attention!