

**SOFTWARE ARCHITECTURE DOCUMENT**

AXIS

CONFIDENTIAL

pages

**Table of Contents**

[1. Architecture Overview 3](#_Toc309662746)

[1.1. Scope of the Document 3](#_Toc309662747)

[1.2. Goals and Objectives 3](#_Toc309662748)

[1.3. Problem Statement 3](#_Toc309662749)

[1.4. Product Overview 4](#_Toc309662750)

[1.5. Glossary 4](#_Toc309662751)

[1.6. Acronym List 7](#_Toc309662752)

[1.7. Technology Overview 8](#_Toc309662753)

[1.8. Architecture Implementation Approach 9](#_Toc309662754)

[1.9. Implementation Notes 10](#_Toc309662755)

[1.9.1. Remote call contract 10](#_Toc309662756)

[1.9.2. Profile implementation 10](#_Toc309662757)

[1.9.3. Service implementation 10](#_Toc309662758)

[1.9.4. Service implementation binding for interpreted languages 10](#_Toc309662759)

[2. Views 11](#_Toc309662760)

[2.1. Layered View 11](#_Toc309662761)

[2.1.1. View Description 11](#_Toc309662762)

[2.1.2. Architecture Background 14](#_Toc309662763)

[2.2. Connectivity View 18](#_Toc309662764)

[2.3. Control View 19](#_Toc309662765)

[3. Tooling 20](#_Toc309662766)

[4. Deployment 20](#_Toc309662767)

[5. Implementation Notes 21](#_Toc309662768)

[5.1. Content of the SDK 21](#_Toc309662769)

[5.2. Content of Toolchain 21](#_Toc309662770)

[5.3. OEM-Developed Components 21](#_Toc309662771)

[5.4. Community-Developed Components 21](#_Toc309662772)

[5.5. Open Source and Proprietary Items 21](#_Toc309662773)

[6. Appendixes 21](#_Toc309662774)

[7. Change History 22](#_Toc309662775)

[8. Approve History 22](#_Toc309662776)

# Architecture Overview

## Scope of the Document

This document describes the overall design and architecture of the Accessory Control and Connectivity for Infotainment Systems (AXIS), the technology for seamless, multimodal data exchange between mobile, automotive, and home infotainment and multimedia devices.

## Goals and Objectives

Development goals for the AXIS connectivity technology:

* Follow KISS (Keep it Simple and Straightforward) principles without sacrificing functionality or ease of use
* High portability between target platforms
* Network abstraction over various physical media
* High levels of response time and throughput for multimedia and real-time streams
* Maximum functional availability by adding common and OEM-specific profiles
* Security and sanity process isolation between operating systems
* Contract-style interfaces between the parts to simplify profile and application development

Target platforms currently supported by AXIS:

* Linux
* Linux-based automotive head units
* Android-based devices
* Apple iOS-based devices

## Problem Statement

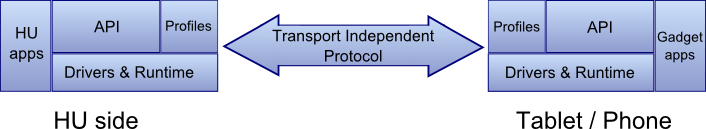
With the current level of automotive infotainment usage, portable user electronics such as mobile phones and tablet PCs have an important value-added opportunity to be used as both terminal HMI devices and sources of content that is displayed on pre-installed vehicle systems. However, currently there are no commonly accepted or fully institutionalized standards for performing the seamless transfer of multimedia, infotainment, and navigation information across the mobile devices. Without a process for seamless information exchange, multiple IVI products are incompatible with each other and integration with mobile gadgets is limited.

Technologies such as DLNA, Bluetooth, and VNC can be adopted for in-car use and have been successfully applied for media and information transfer. Therefore, joining these approaches under one managing framework that provides transparent access and seamless usability produces an excellent synergy between the various types of in-vehicle infotainment (IVI) and mobile devices — and AXIS solves this two-way symmetrical integration challenge.

AXIS is a portable application development framework that provides secure, OEM-controlled, two-way connectivity and system abstraction for:

* Access to content via screen and controls
* Audio / Video sharing
* Internet connectivity sharing
* Telephony services sharing
* Navigational systems sharing
* API access for app developers

Figure : AXIS Framework

****

Primary Use Cases for Accessory Control and Connectivity:

* Use a mobile device (smartphone or tablet) as the application source for a head unit
* Use a mobile device to display from a head unit, including screen-less and low-cost head units such as 1-DIN modules
* Use the head unit screen to display mobile device content (screen sharing)
* Synchronize head unit and mobile device content
* Share media content between multiple devices, including smartphones, tablets, personal navigation devices (PND), and rear seat entertainment (RSE) devices
* Integrate remote navigational services
* Integrate rear and side view cameras, include both wired and wireless cameras
* Integrate aftermarket in-vehicle equipment with wired connectivity minimal adoption requirements

## Product Overview

AXIS is a software development kit (SDK) with development tools for in-vehicle platforms. It gives developers the tools needed to synchronize data, applications, and multimedia content between mobile devices and the head unit.

AXIS includes:

* Profile-oriented protocol that performs control functions and data exchange
* In-band and out-of-band multimedia data exchange functionality
* Sample applications
* Development toolchain

AXIS includes the following user-oriented use cases:

* Remote HMI management, including raster screen exchange and remote rendering with provisioning of controls and behaviors
* Seamless switching between local and remote sinks for audio and video playback with playback control from any side
* Internet connectivity sharing
* Use of telephony service for dialing from the remote device (headset and handset options)
* Navigational systems function sharing
* Remote API access

## Glossary

**Application pairing** — process started at the time of registering all services by application. Including:

* Select appropriate profiles on both sides (service pairing phase 1)
* Select shadow applications on the server side
* Run shadow applications with loading shadow profiles
* Check consistency of loaded parts across network
* report process success or failure

**AXIS application** — application linked to the AXIS library that uses the AXIS queue for AXIS-provided communications. AXIS applications serve data streams on both the data provision and the data consumption sides.

**AXIS daemon** — background software process, consisting of the carrier abstraction layer (CAL) and the set of repositories, which performs the following operations:

* Device pairing
* Raw data exchange
* Channel multiplexing and scheduling
* Profiles binding
* Service and profile management
* Axis.queue management
* Profile update and synchronization
* Updating of profiles implementations

**Axis\_gen** — utility for code generation from the interface definition language (IDL)

**Axis\_lbgen** — utility for generating language-dependent glue code for the language binding generator (LBG) service

**Axis.queue** — UNIX domain socket used for all command communication with AXIS daemon

**Carrier** — serial communications link for TCP/IP, Bluetooth, and USB

**Carrier abstraction layer (CAL)** — the part of the AXIS daemon responsible for low-level transmission over different carriers..

**Carrier Adapter** — entity that encapsulates the low-level I/O for a given carrier and defines the low-level handshake strategy between the client-side and server-side AXIS daemons

**Client-side** — AXIS application operations performed by the client in a client-server relationship

**Client-side application** — AXIS application that runs on the client and can use many server instances that are loaded by multiple services

**Connectivity agent** — the entity that finds and establishes a physical connection

**Connectivity manager** — creates and manages the CAL infrastructure when a physical connection established

**Control stream** — abstract stream of data designed for RPC transmission that is used for typed, contracted data and signal delivery.

**Demarshaller** — component responsible for restoring typed data back to its original data unit

**IDL-based interface** — contract defined by IDL-file and used for restricted data transmission between profile pairs

**IDL compiler** — code generator used to create profile and PAD glue code

**Interface definition language (IDL)** — DSL designed for profile-to-profile remote call description, where the interface definition is specific to a given profile

**Foreign Function Interface** — interfaces designed for glue code written in different languages, for example the java native interface (JNI) is a foreign function interface

**Frame** — atomic element of transmission

**Handshaking** — process of negotiation that dynamically sets the carrier parameters established between two devices before normal communication over the network begins

**Language-binding generator** — code generator used to create language-specific glue code between a service and the AXIS application

**Logical channel** — duplex transmission channel associated with a profile instance

**Logical connection** — all profile channels associated with given profile instance

**Logical data unit** — unstructured block of information suitable for atomic transmission

**Marshaller** — component responsible for assembling typed data into one logical data unit for further transmission

**Maximum transmission unit** — size (in bytes) of the largest protocol data unit that the layer can pass forward

**Multimedia stream** — binary stream of raw data with no types or entities recognizable on the AXIS infrastructure level and is used primarily to transfer large amounts of media data in real time

**Packet assembler/disassembler (PAD)** — component responsible for splitting a data stream into LDU’s for transmission and then to restore the data on the receiving end

**PAD level** — packet assembler/disassembler level between the RPC and AXIS channels

**Physical channel** — multiplexing abstraction of simplex physical channel, implemented as mark on every frame.

**Physical channel#0** — single duplex physical channel selected for administration and logical pairing

**Physical frame** — atomic unit designed for transmission over physical connection

**Physical MTU** — maximum transfer unit size for a given physical connection

**Physical pairing** — process responsible for establishing the physical connection and handshaking

**Per profile channel** (**PPC)** — UNIX socket created for a given channel and a given profile instance

**PPC#0** — UNIX socket associated with the profile’s initial channel

**Profile** — a named and versioned set of modules (dynamic libraries) with a predefined API, responsible for remote method invocation and streaming content exchange. The profile converts application-level operations into a form of API invocation of the process of formatted data exchange via RPC and PAD. Each profile has two role-specific implementations deployed on paired devices that are mutually connectable for data exchange according for the data exchange protocol (contract) defined by the given profile specification.

**Profile API** — is the named contract that specifies the methods whereby a requesting application receives typed data. The profile API contains method specifications and the definition of how these methods are to be used. Multiple profiles that implement the same API are polymorphs that can execute the same services on a selective basis.

**Profile API repository** — database containing profile APIs

**Profile implementation** — representation of a profile for selected role, version, and hardware architecture in the form of a dynamic linked module

**Profile instance** — a copy of the profile in the application address space that is bound to the corresponding remote side for executing a dedicated task

**Profile repository** — contains the profiles map, a simple folder with XMLs and subfolders named by platform

**Profiles update process** — process built on top of the profile repository that is responsible for updating, uploading, and downloading profiles

**Protocol data unit** — minimum data chunk with atomic transmission property over a physical carrier

**Remote procedure call** — method used to restrict profile interaction to only strong-typed messages and is also used to avoid finite automatons or protocol-based solutions

**RPC level** — automatically generated code based on interface specification (IDL)

**RPCGEN** — interface pre-compiler for Sun Microsystems RPC that is designed to generate C code from the IDL

**Server-side** —AXIS application loaded by AXIS daemon on the server in response to a client-side application

**Server-side application** — AXIS application performed by the server in a client-server relationship. A server-side application may implement one more services and can serve many client-side applications by dynamically creating or destroying service instances for client-side applications.

**Service** — is the named list of profile API IDs that are dedicated to direct request from applications. The Service groups profile APIs into a single entity in the service repository to allow atomic initialization

**Service instance** — a group of profiles that implement a given service

**Service level agreement (SLA)** — part of QoS, the SLA is the guaranteed minimum quality of service required for contracted service work. It specifies the level of service that is guaranteed based on mutual agreement for measures of service performance, such as acceptable levels of throughput, latency, or data rate.

**Service repository** — contain services (map of services). Actually is simple folder with XML files.

**Synchronization and update manager (SUM)** — component built over the profile repository that is responsible for the profile update process

**System Control Profile** — holder of the Physical Channel#0 and processor of administrative requests

## Acronym List

**API** — application programming interface

**ASP** — audio streaming profile

**CAL** — carrier abstraction layer

**DLL** — dynamic link library

**DLNA** — digital living network alliance

**DSL** — data structure language

**FFI** — foreign function interface

**GUUID** — globally universally unique identifier

**HFP** — hands free protocol

**HMI** — human machine interface

**IDL** — interface definition language

**IPC** — interprocess communication

**JNI** — java native interface

**LBG** — Language Binding generator

**LDU** — logical data unit

**MSQ** —

**MTU** — maximum transmission unit

**PAD** — packet assembler/disassembler

**PDU** — protocol data unit

**PND** — portable navigation device

**PPC** — per profile channel.

**PUP** — profile update profile

**QoS** — quality of service

**RFSP** — remote file system profile

**RPC** — remote procedure call

**RPCL** — RPC language, common standard of SUN ONC and internet community

**RSE** — rear seat entertainment

**RTP** — real-time transport protocol

**SAS** — streaming audio service

**SCM** — system control manager

**SCP** — stream control profile

**SDK** — software developer’s kit

**SLA** — service level agreement

**SUM** — synchronization and update manager

**SWIG** — simplified wrapper and interface generator

**VNC** — virtual network computing

**XDR** — external data representation, standard to internet community

## Technology Overview

Figure : AXIS Architecture

pyramid

The AXIS architecture is a layered approach with strict rules of interaction between layers.

At the lowest level is the carrier abstraction layer (CAL) responsible for media-independent data transmission.

Above the CAL level, data is transferred in two ways: via remote procedure call (RPC) or via asynchronous data streaming. The RPC is oriented on control data transfer (AXIS control) and asynchronous data streaming is mostly applicable for media data transportation (AXIS streaming).

Over the data transfer level is the profile level. AXIS handles profiles as polymorphic chunks of code responsible for higher abstraction and for hiding network data exchange. Profiles are grouped into services. A service is a named set of profiles that are suitable for atomic initialization and access, and fully cover a complete use case.

At the top, the application level implements business logic, applied user interactions, and is controlled by third party applications. The AXIS SDK only provides connectivity support for the application layer, but it doesn’t define the application structure, design, or functionality.

An example of the steps in the data exchange process:

1. The user has a mobile device with a folder containing mp3 files to play on the car’s audio system
2. The user starts the media player on the mobile device
3. The mobile device media player connects with the AXIS daemon and requests streaming audio service (SAS)
4. The requested service contains two profiles: the audio streaming profile (ASP) and the stream control profile (SCP) that define the play, pause, stop, and seek interfaces
5. The AXIS daemon sends the request to to the head unit side and verifies the existence of requested profiles and service on both the mobile device and the head unit. The AXIS daemons on the mobile device and the head unit collaboratively select the best profiles for a given service.
6. When the remote application with required service support is selected, the media player on the head unit executes SAS.
7. If two profile implementations for ASP are available, such as one that uses RTP streams and one that uses Bluetooth hands free transport, the AXIS daemons will select the most suitable profile. In this case, Bluetooth pairing HFP would be selected.

## Architecture Implementation Approach

Figure : AXIS Implementation View

**axis_deployment2**

## Implementation Notes

### Remote call contract

Interaction between profiles on both sides of the communication is defined using the Interface Definition Language (IDL). The RPC level checks every data chunk for validity of the method signatures and data types. IDL supports the following data types: structures, discriminated unions, arrays, and ASCIIZ strings. In addition, the RPC level performs automated byte-order conversions that conform to the target CPU architecture.

The RPC level contains only the code generated by the IDL compiler that ia a part of the AXIS development toolchain. This design allows for the implementation of profile “genders” that can be developed separately and provided by different vendors, but remain compatible because of the contract on data exchange.

### Profile implementation

The profile installation consists of code and the manifest. The code is fully packed in the form of dynamically linked library modules (one per client and server for each of supported architectures) with predefined function names (using the \_\_cdecl naming convention) that are suitable for dynamic loading. The manifest contains the profile API description that specifies the content of the DLL in order for the infrastructure to manage it properly.

The Profile supports polymorphism, allowing a single profile API to be applied to multiple profile implementations.

The AXIS runtime infrastructure is responsible for profile selection using optimization rules provided by the manifest and the applications.

A RFSP (remote file system profile) can be used as an example of polymorphism. Each profile contains a single profile API. One profile may use a secure channel, and another profile may not. The RFSP can be implemented with multiple profile APIs so that in the case of valid X.509 certificates with common certifiers, AXIS can select the secure profile API, or in the case of a slow hardware system, it can select a plain transfer. AXIS selects the profile API that is best suited to the connection.

### Service implementation

A service is implemented as an auto-generated wrapper around used profiles and has its own API, which is consolidated from profile APIs. A service is linked with an application at compile time and contains the necessary code to support profile polymorphism. Profile polymorphism is implemented by using dynamic implementation (DLL) selection that is loaded at run time.

### Service implementation binding for interpreted languages

For interpreted languages, the use of static linkage is not possible. In this case, the service code is precompiled as a dynamic library that includes FFI and wrappers to the target language.

# Views

## Layered View

### View Description

#### Global view

Figure : Layered View of the AXIS Stack

**global_axis**

#### AXIS application

An AXIS-compatible application is a regular program on the target system and OS that is linked to the AXIS core library that provides access to the AXIS connectivity service functions. An AXIS application may import services (use services as a client) or export services (provide services as a server).

An AXIS application may have two roles: client or server. Client applications are typically launched by a user and connect to the AXIS daemon proactively. Server applications are typically run by the AXIS daemon on demand to handle and respond to a client application request. The client and server roles of an AXIS application are purely logical and are not predefined. AXIS applications are not limited to only exporting or only importing services, regardless of the role.

#### AXIS service

An AXIS service is a named entity for a group of profile API’s. An AXIS service provides atomic initialization of multiple profiles that implement one block of user functionality. In simple cases, an AXIS service may contain only one profile.

An AXIS service provides a consistent API (including target language wrappers) for contained profile APIs. An AXIS service implementation is generated automatically from the profile API and an XML description of the service. Currently, the supported languages are C, C++, and Java; but more languages are easily added by using FFI generators such as SWIG.

An AXIS service may be linked to an application statically (languages with static linking support such as C and C++) or dynamically (Java and scripting languages).

#### AXIS profile

An AXIS profile is a key feature of the AXIS system. From an implementation standpoint, the profile is a self-described, dynamically linked module with predefined interfaces for its upper level consumers and with open interfaces for remote connectivity.

AXIS profile instances exist as polymorph objects with the same profile API in two genders: client and server. For example, a Remote File System Profile (RFSP) on the server side implements local file system requests, while on the client side it serves as an interface for file system calls.

There are two kinds of interfaces in a profile:

* High-level profile API that is focused on application data exchange and control message flow. This API is used to access the profile through the services layer from applications.
* Low-level, RPC-based interface that is described by IDL and is suitable for remote call abstraction.

Figure : AXIS Profile Features

profile_comp.emf

A profile API defines and describes interfaces and data structures that are used for the interaction between an application and the profile. Different profile implementations of same profile API use the same API calls, arguments and method signatures.

A profile API is consists of four headers (in terms of C language): two on the server side and two on the client side. The client and server each have two headers defined: one for direct methods (calls from application to profile) and one for callback methods (calls from profile to application).

When an application loads a profile (as DLL), dynamic interface binding is performed by the AXIS service. This process is possible, because the profile API is identified to the application at compile time.

The communication interface between profile instances on the remote side of the link are defined by the IDL language description. To simplify the profile development, AXIS uses RPC abstraction with asynchronous callbacks.

The IDL file is a contract on data types and remote method invocations. Using IDL, the contract allows safe and secure communication between profiles written by different vendors. In addition to IDL-defined synchronous and asynchronous methods, profiles use the concept of logical channels for low latency, unstructured data streams. Logical streams can be created or destroyed by a profile at any time; every logical channel is processed by the pair of callbacks on each side for fast incoming data processing. Stream-oriented data is passed with using a special scheduler for real-time media data that is given higher priority.

A profile implementation has two roles, server and client, that share a common IDL.

#### RPC

The RPC level is responsible for the transformation of architecture-dependent typed data into data chunks, and vice versa. All RPC code is generated by the IDL compiler and is directly included into the profile implementation.

The data marshaller is based on the XDR protocol (RFC 4506), and generated packets are mostly compatible with XDR specifications. The only exception prohibits recursive declarations for system stability.

#### PAD

PAD (Packet assembler/disassembler) is responsible for the breakdown of un-typed data chunks into physical frames that require tagging and assembly into the logical data chunks that are suitable for use by RPC.

PAD also performs the remote synchronous call emulation over an asynchronous packet exchange mechanism and is responsible for remote callback dispatching.

The PAD implementation contains heavy-thread and asynchronous queue management. AXIS implements message exchange between the components with call blocking emulation, instead of regular function calls, for proper inter-process communication.

#### CAL

AXIS operates over the carrier abstraction layer, which may use wired Ethernet, Wi-Fi, Bluetooth, USB or almost any serial bus interface. CAL is a low-level API abstraction that enables AXIS over any sort of carrier that provides a reliable serial link.

The CAL component is responsible for physical connection management, including handshaking, exception processing, handover, and logical channel multiplexing. CAL operates with single serial connection that performs multiplexing of logical channels.

Every channel maintains its own scheduling rules for priority and exceptions. There are three types of channels with different schedulers: command, real-time, and critical. Data passed on critical channels is delivered with high priority. Real-time channels are suitable for passing multimedia data with minimal jitter, and real-time packets have internal timers so that the connection can be discarded in case of long delays.

When a connection is lost, CAL first tries to reconnect. If the reconnect attempt is unsuccessful, CAL scans for another carrier to re-establish the connection and notifies the upper layer..Carrier scanning is performed in descending order by expected bandwidth. When a connection is established, CAL passes the event to the upper layer.

CAL maintains QoS information such as bandwidth, latency, jitter, and other metrics. All QoS measurements are accessed from the upper layers.

#### PUP

PUP (Profile Update Profile) designed to populate and update the profile and service repositories. In systems with strict X.509 usage, PUP must maintain all digital signatures for the profiles.

On each successful connection between devices, PUP starts its own RPC-based protocol to synchronize profile implementations according to the target architecture. PUP connections run transparently to other processes and are low priority. In the first phase of handshaking, PUP exchanges information about the device’s architecture. In the second phase, PUP sends the list of available profile implementations with versions. After this, if new profile versions are found for a given architecture, PUP sends the profile code with descriptors, including version, create date, vendor-specific information, and GUUID.

### Architecture Background

.

#### Element Catalog

##### CAL (Carrier Abstraction Level)

##### Responsibility

CAL is responsible for carrier abstraction, connect and disconnect event handling, device discovering and physical pairing

##### Relations

* AXIS daemon
* AXIS daemon manager
* PPC

##### Interfaces

* AXIS manager
* PPC
* External carrier protocol

##### Behavior

Handles connect/disconnect events, chooses the optimal carrier, maintains QoS measurements, and performs PPC serving

##### Constraints

* Operated over point-to-point connection
* Use Bluetooth, Wi-Fi, TCP/IP, USB carriers

##### AXIS Daemon Manager

##### Responsibility

Responsible for repository management, axis,queue management, CAL interaction, and PPC lifecycle management

##### Relations

* AXIS daemon
* CAL
* Repositories
* PPC

##### Interfaces

* Axis.queue
* CAL
* Repositories

##### Behavior

Application loading, profile and services selection, and axis.queue event processing

##### Constraints

* Singleton

##### Profile Update Process (PUP)

##### Responsibility

Responsible for repository update

##### Relations

* AXIS daemon
* CAL
* Repositories

##### Interfaces

* AXIS daemon manager
* CAL
* Repositories

##### Behavior

Permanently running; on every connect it checks the profile, service, and application repositories on remote side and downloads new versions as needed; where X.509 certificates are used, PUP will check digital signatures

##### Constraints

* Singleton

##### Axis.queue

##### Responsibility

Responsible for the interaction between the AXIS daemon and applications

##### Relations

* AXIS daemon manager
* AXIS manager

##### Interfaces

* AXIS daemon manager
* AXIS manager

##### Behavior

Permanent UNIX domain socket

##### Constraints

* Singleton

##### Profile Repository

##### Responsibility

Responsible for storing and retrieving profile code chunks on disk

##### Relations

* AXIS daemon manager
* PUP

##### Interfaces

* AXIS daemon manager
* PUP

##### Behavior

Simple file-system-based database of profile implementations

##### Constraints

* Singleton
* For X.509-enabled systems, all profiles must have valid signatures

##### Service Repository

##### Responsibility

Responsible for storing and retrieving service descriptions (XML files)

##### Relations

* AXIS daemon manager
* PUP

##### Interfaces

* AXIS daemon manager
* PUP

##### Behavior

Simple file-system-based database of service descriptions

##### Constraints

* Singleton

##### Application Repository

##### Responsibility

Responsible for storing and retrieving application paths and application-to-service mapping

##### Relations

* AXIS daemon manager
* PUP

##### Interfaces

* AXIS daemon manager
* PUP

##### Behavior

Simple file-system-based database of application descriptions, including path to run and CLI parameters and application-to-service mapping information

##### Constraints

* Singleton
* For some installation may be constant database.

##### AXIS Manager (part of application)

##### Responsibility

Responsible for hiding the complex axis.queue interaction and for service loading by simple API

##### Relations

* AXIS daemon manager
* Service

##### Interfaces

* AXIS daemon manager
* Service

##### Behavior

Handshaking with axis.queue, service registration and lifecycle management

##### Constraints

* Singleton

##### Service (part of application)

##### Responsibility

Responsible for profile grouping, providing a single API

##### Relations

* AXIS manager
* Application code

##### Interfaces

* AXIS manager
* Application

##### Behavior

Loading profile instances and API grouping

##### Constraints

* One service instance corresponds to 1..n profile instances
* Service code generated by automated way

##### Profile

##### Responsibility

Responsible for profile API implementation and IDL contract

##### Relations

* Service
* RPC

##### Interfaces

* Profile API
* IDL contract

##### Behavior

OEM defined

##### Constraints

* Profile API
* IDL

##### RPC

##### Responsibility

Responsible for data marshalling in an architecture-independent way

##### Relations

* Profile
* PAD

##### Interfaces

* Profile
* PAD

##### Behavior

Converts un-typed data blocks on typed, remote method signatures and vice versa

##### Constraints

* Architecture and byte-order independent

##### PAD (Packet Assembler/Disassembler)

##### Responsibility

Responsible for frame tagging and data chunk assembly

##### Relations

* RPC
* PPC

##### Interfaces

* RPC
* PPC

##### Behavior

Works with PPC from the application side and emulates RPC call blocking via wait-on-response

##### Constraints

* Architecture and byte-order independent

## Connectivity View

Figure : AXIS Interface Connectivity

AXIS_comp

Primary Interfaces:

1. Axis.queue: the interaction between the AXIS daemon and applications
2. PPC: per profile channel sockets
3. Application control

## Control View

Figure : AXIS Typical Use Case MSQ

main

# Tooling

There are two main tools included in AXIS: axis\_rpcgen and axis\_lgen.

Axis\_rpcgen is an IDL compiler that produces the following files in C language for a single IDL input:

* <protocolname\_types>.h. Contain type mapping information
* <protocolname\_types>.c. Contain type mapping and marshalling routines
* <protocolname\_client>.c. Contain definitions of wrappers and callback dispatcher on client side.
* <protocolname\_client>.h. Contain declarations of wrappers and callback dispatcher on client side.
* <protocolname\_server>.c. Contain definitions of wrappers and callback dispatcher on server side.
* <protocolname\_server>.h. Contain declarations of wrappers and callback dispatcher on client side.

These files are linked to the corresponding profile genders:

* Server is linked to <protocolname\_server>.c and <protocolname\_types>.c
* Client is linked to <protocolname\_client>.c and <protocolname\_types>.c

Axis\_lgen is a utility for service code generation with wrappers for the target language. Axis\_lgen uses an XML-file with service description and profile API headers as input to generate wrapper files with profile polymorphism implementation and target language binding (currently only supported by Java).

# Deployment

Figure : AXIS Implementation View

axis_deployment2

Central to the AXIS system is the AXIS daemon, a continually running process in a separate address space that is responsible for most run-time operations. The AXIS daemon maintains the axis.queue UNIX domain socket as the main point of high-level communications between AXIS applications and PPC sockets for direct, low-level data transfer. The AXIS daemon is a singleton and running several AXIS daemon instances on same device is prohibited.

The AXIS daemon contains the CAL component that is responsible for all low-level communication between devices and the PUP component that is responsible for remote updating of profile code. It also maintains the set of repositories (profile and service) and functions as the system resource manager. Each device requires its own AXIS daemon.

The AXIS application entity is responsible for high-level, user-defined operations such as multimedia encoding, play, or navigation, and event processing. The AXIS application and AXIS daemon are interconnected in two ways: through the axis.queue and the PPC.

The PPC consists of dynamically created UNIX domain sockets for fast data transfer between the profile part of the AXIS application and CAL. All frames transferred by the PPC have a size and structure that is compatible with the CAL frames transferred by a physical medium. Thus, frame processing is minimized in all phases of transmission.

AXIS applications are dynamically linked to AXIS services.

# Implementation Notes

## Content of the SDK

The AXIS SDK contains headers and libraries for application linking and the toolchain.

## Content of Toolchain

The toolchain contains two code generators: the axis\_rpcgen and axis\_lgen.

## OEM-Developed Components

OEMs can develop profiles that comply with their own security restrictions and stability requirements. OEMs can also develop applications (mainly on head unit side) according to their own requirements.

## Community-Developed Components

Third parties may develop applications and optional profiles, but the head unit vendor may restrict the use of community-written profiles on their head unit products.

## Open Source and Proprietary Items

Currently, AXIS does not contain any proprietary components. The following open source items are required for AXIS:

* GNU compiler
* GNU libc
* GNU make
* Perl
* OS Linux
* Android SDK
* Android NDK
* SWIG
* Libuuid
* Pthreads library
* Bzip2 library (optional for protocol compression)

# Appendixes

**CONTENTS OF THIS SECTION:** Appendices may be used to provide information published separately for convenience in document maintenance (e.g., charts, classified data, API specifications). As applicable, each appendix is referenced in the main body of the document where the data would normally have been provided. Appendices may be bound as separate documents for ease in handling. If your SAD has no appendices, delete this section.

# Change History

| Version | Date | Status | Change description | Author/Editor |
| --- | --- | --- | --- | --- |
| 0.1 | 1 Nov 2011 | draft | initial | Dmitry Lobashevsky |
| 0.2 | 18 Nov 2011 | Draft | * Updates to correct English grammar and word usage * Updates to formatting for readability * Add missing acronym definitions * Add captions to figures * See also the comment on the glossary and acronym list * See also comment for empty sections:   + 1.8 Significant Driving Requirements   + 2.1.1 View Description   + 2.1.1.1 Global View | Pamela Traphagen |
|  |  |  |  |  |
|  |  |  |  |  |

# Approve History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Approval Date | Issue ID | Approved By |
|  |  |  |  |