

Joint Tactical Radio System (JTRS)

SCA Developer's Guide

Contract No. DAAB15-00-3-0001

Document Number: Rev 1.1

18 June 2002

Prepared for the
Joint Tactical Radio System (JTRS)
Joint Program Office

Prepared by:

Raytheon

Raytheon Company
Radios and Terminals
1010 Production Road
Fort Wayne, In 46808

Joint Tactical Radio System (JTRS)

SCA Developer's Guide

Contract No. DAAB15-00-3-0001

Document Number: Rev 1.1

18 June 2002

Prepared for the
Joint Tactical Radio System (JTRS)
Joint Program Office

Prepared by:

Raytheon

Raytheon Company
Radios and Terminals
1010 Production Road
Fort Wayne, In 46808

Prepared By

Al Gonzalez

Al Gonzalez

Date: 6/21/2002

Randy Hess

Randy Hess

Date: 6/21/2002

Program Manager

James Brown

Jim Brown

Date: 6/21/2002

Software Quality Assurance

Brian Ault

Brian Ault

Date: 6-21-02

1	INTRODUCTION.....	4
1.1	SCOPE.....	4
1.2	TERMINOLOGY	4
2	SCA OVERVIEW	6
2.1	INTRODUCTION.....	6
2.2	ARCHITECTURE DEFINITION METHODOLOGY	7
2.3	ARCHITECTURE OVERVIEW.....	8
2.4	FRAMEWORK INTERFACES.....	10
2.4.1	<i>DomainManager</i>	10
2.4.2	<i>ApplicationFactory</i>	10
2.4.3	<i>Application</i>	10
2.4.4	<i>File</i>	10
2.4.5	<i>FileSystem</i>	10
2.4.6	<i>FileManager</i>	11
2.5	BASE APPLICATION INTERFACES.....	11
2.5.1	<i>Port</i>	11
2.5.2	<i>LifeCycle</i>	13
2.5.3	<i>TestableObject</i>	13
2.5.4	<i>PortSupplier</i>	14
2.5.5	<i>PropertySet</i>	15
2.5.6	<i>Resource</i>	16
2.5.7	<i>ResourceFactory</i>	17
2.6	SERVICE INTERFACES.....	18
2.6.1	<i>Naming Service</i>	18
2.6.1.1	Use of Naming Service	19
2.6.2	<i>Log Service</i>	20
2.6.2.1	Use of Log Service	20
2.6.3	<i>CORBA Event Service</i>	21
2.6.3.1	Use of Event Service.....	22
3	APPLICATION PROGRAM INTERFACE (API) OVERVIEW	24
3.1	GENERIC PACKETS.....	25
3.2	PHYSICAL API.....	25
3.2.1	<i>Physical Real Time</i>	25
3.2.2	<i>Physical Non-Real Time</i>	25
3.3	MEDIUM ACCESS CONTROL (MAC) API	26
3.4	LOGICAL LINK CONTROL API.....	26
3.5	I/O API.....	26
4	DOMAIN PROFILE COMPONENTS	27
4.1	SOFTWARE PACKAGE DESCRIPTOR.....	28
4.2	PROPERTIES DESCRIPTOR.....	29
4.3	SOFTWARE COMPONENT DESCRIPTOR.....	30
4.4	SOFTWARE ASSEMBLY DESCRIPTOR.....	31
4.5	PROFILE DESCRIPTOR.....	32
5	DESIGN PROGRESSION.....	33
5.1	IDL MODELING.....	33
5.2	IDL GENERATION.....	34
5.3	IDL COMPILATION	35
5.4	CLIENT/SERVER COMPILATION	36
5.5	REVERSE ENGINEERING.....	36

5.6	CREATION OF SERVANT IMPLEMENTATION CLASSES	37
5.7	SERVANT CODE GENERATION	38
5.8	XML GENERATION	40
6	WAVEFORM DEVELOPMENT	40
6.1	FUNCTIONAL ALLOCATION TO API DESIGN	40
6.1.1	<i>Introduction</i>	41
6.1.2	<i>Identifying Application Functionality</i>	42
6.1.2.1	Data and Control from Attached Device.....	42
6.1.2.2	Data and Control to Attached Device	42
6.1.2.3	Control from User Interface (UI).....	42
6.1.3	<i>Functional API Mapping</i>	43
6.1.4	<i>Mapping Remaining Needs</i>	44
6.2	BUILDING API LAYER DEFINITIONS.....	45
6.2.1	<i>Introduction</i>	45
6.2.2	<i>Waveforms with Physical Layer</i>	45
6.2.2.1	Using Concrete Service Group	46
6.2.2.2	Instantiating Generic Building Block	46
6.2.2.3	Extending A Service Group	47
6.2.2.4	Constructing A New Service Group	48
6.2.2.5	Combining Service Groups to Form A New Interface	48
6.2.2.6	Completing the Waveform Interface.....	50
6.2.3	<i>Waveforms with MAC Layer</i>	51
6.2.3.1	Explanation of the Task	51
6.2.3.2	Solution Using MAC Building Block(s).....	51
6.2.4	<i>Waveforms with Link Layer</i>	53
6.2.4.1	Explanation of the Task	53
6.2.5	<i>AssemblyController</i>	56
6.3	REFINING API DEFINITIONS WITH IMPLEMENTATION DESIGN	57
6.3.1	<i>Use of Interfaces</i>	57
6.3.1.1	Implementing A Provides Port.....	58
6.3.1.2	Implementing A Uses Port.....	58
6.3.1.3	Implementing Multiple Ports	58
6.3.1.4	Implementing getPort().....	58
6.3.2	<i>Model for Physical Layer</i>	59
6.3.2.1	Unified Design	59
6.3.2.2	Partitioned Design	60
6.3.3	<i>Model for MAC Layer</i>	61
6.3.4	<i>Model for Link Layer</i>	62
6.3.5	<i>Model for AssemblyController</i>	63
7	DEVICE CREATION	64
7.1	DEVICE INTERFACES.....	64
7.1.1	<i>Device</i>	64
7.1.2	<i>LoadableDevice</i>	65
7.1.3	<i>ExecutableDevice</i>	66
7.1.4	<i>AggregateDevice</i>	67
7.1.5	<i>DeviceManager</i>	67
7.1.6	<i>Defining the Device</i>	69
7.1.6.1	Selecting the Appropriate Device Interface	69
7.1.6.2	Designing a Device Servant	70
7.1.6.3	Device Configuration Descriptor.....	72
7.2	DEVICE PACKAGE DESCRIPTOR.....	73
7.3	DOMAINMANAGER CONFIGURATION DESCRIPTOR.....	75
8	UI DISCUSSION	76
8.1	INTRODUCTION.....	76
8.2	DIRECT CORBA LINKS	76

8.3 NON-DIRECT CORBA LINKS 77

9 APPENDICES 78

9.1 APPENDIX A – XML INTRODUCTION..... 78

9.2 APPENDIX B – IDL FOR XYZ WAVEFORM PHYSICAL LAYER..... 78

9.3 APPENDIX C – HEADER FILES FOR XYZ WAVEFORM PHYSICAL LAYER..... 78

9.4 APPENDIX D – XML FOR A SAMPLE WAVEFORM 78

9.5 APPENDIX E – XML FOR A SAMPLE DEVICE 78

1 Introduction

This Developer's Guide provides design guidelines for developing compliant waveform applications and devices for the Software Communications Architecture (SCA) published by the Joint Tactical Radio System (JTRS) Joint Program Office (JPO). This guide is organized as follows:

- ?? an overview of the SCA and its services
- ?? application program interface (API) overview
- ?? domain profile discussion
- ?? design progression
- ?? waveform development
- ?? device creation
- ?? user interface
- ?? appendices

This guide also contains working examples of IDL (Interface Definition Language) and XML (eXtensible Markup Language). This document is intended to be a living document that is to be updated as necessary to reflect the current version of the SCA.

1.1 Scope

This guide is written for users developing SCA-compliant waveforms and/or devices. A software framework is defined in the SCA, and is comprised of:

- ?? a Portable Operating System Interface (POSIX)-compliant Operating System (OS)
- ?? a distributed computing middleware, CORBA
- ?? a set of open-software interfaces geared toward embedded distributed communications systems, Core Framework (CF)

These three elements make up the SCA Operating Environment (OE).

We have assumed that the reader has some knowledge of the SCA. Further, we assume that the reader has a high degree of experience with object-oriented design, CORBA, UML, and high-level languages. Coding examples are provided using the C++ programming language. A brief definition of some of the concepts and terms used in this document is found in the next section.

1.2 Terminology

There are a number of terms used throughout this document – some of the more essential are explained at this time.

OMG (Object Management Group) – In 1989, the OMG was formed to address the problems of developing portable distributed applications for heterogeneous systems. It is now the world's largest software consortium, with more than 800 members. Two key specifications produced by the OMG, the OMA (Object Management Architecture) and its core, the CORBA specification, provide a flexible architectural framework that accommodates a wide variety of distributed systems.

CORBA (Common Object Request Broker Architecture) provides platform-independent programming interfaces and models for portable distributed-oriented computing applications. CORBA is the middleware of the SCA OE. Its independence from programming languages, computing platforms, and networking protocols makes it highly suitable for the development of new applications and their integration into existing distributed systems. CORBA has associated with it some unique terminology; the most important of which is explained in the following list.

- ?? A CORBA object is a “virtual” entity capable of being located by an ORB and having client requests invoked on it. It is virtual in the sense that it does not really exist unless it is made concrete by an implementation written in a programming language.
- ?? A target object, within the context of a CORBA request invocation, is the CORBA object that is the target of that request.
- ?? A client is an entity that invokes a request on a CORBA object.
- ?? A server is an application in which one or more CORBA objects exist.
- ?? A request is an invocation of an operation on a CORBA object by a client.
- ?? An object reference, also known as an IOR (Interoperable Object Reference) is a handle used to identify, locate, and address a CORBA object.
- ?? A servant is a programming language entity that *realizes* (i.e., implements) one or more CORBA objects. Servants are said to be incarnate CORBA objects because they provide bodies, or implementations, for those objects. Servants exist within the context of a server application. In C++, servants are object instances of a particular class.

IDL (Interface Definition Language) – The OMG IDL is CORBA’s fundamental abstraction mechanism for separating object interfaces from their implementations. OMG IDL establishes a contract between client and server that describes the types and object interfaces used by an application. This description is independent of the implementation language, so it does not matter whether the client is written in the same language as the server.

IDL Definitions are compiled for a particular implementation language by an IDL compiler. The compiler translates the language-independent definitions into language-specific type definitions and APIs (Application Program Interface). These type definitions and APIs are used by the developer to provide application functionality and to interact with the ORB. The translation algorithms for various implementation languages are specified by CORBA and are known as language mappings. CORBA defines a number of language mappings including those for C++, Ada, and Java (along with many others).

An IDL compiler produces source files that must be combined with application code to produce client and server executables¹. Details, such as the names and numbers of generated source files, vary from ORB to ORB. However, the concepts are the same for all ORBs and implementation languages. The outcome of the development process is a client executable and a server executable. Section 5 provides more detail.

UML (Unified Modeling Language) is a standard (modeling) language for writing software blueprints². UML enables system builders to create blueprints that capture their visions in a standard, easy-to-understand way and communicate them to others. It may be used to visualize, specify, construct, and document the artifacts of a software-intensive system. The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies; visually, that class icon might only show a small part of this specification. UML diagrams are used in numerous ways – here, however, we focus on two: to specify models from which an executable system is constructed (forward engineering) and to reconstruct models from parts of an executable system (reverse engineering).

¹ “[Advanced CORBA Programming With C++](#)” (Addison-Wesley Professional Computing) Henning & Vinoski

² “[The Unified Modeling Language User Guide](#)” (Addison Wesley) Grady Booch, James Rumbaugh, Ivar Jacobson, p13

XML (eXtensible Markup Language) is a markup language designed specifically for delivering information over the World Wide Web. XML is used within the SCA to define a profile for the domain in which waveform applications can be managed. XML's definition consists of only a bare-bones syntax³. When you create an XML document, rather than use a limited set of predefined elements, you create your own elements and assign them any names you like – hence the term *extensible*. You can therefore use XML to describe virtually any type of document, from a musical score to a digitally-programmable radio. However, for JTRS, this extensibility is limited to the SCA-defined Document Type Definitions (DTDs). A DTD provides a list of the elements, attributes, notations, and entities contained in a document, as well as their relationship to one another. DTDs specify a set of rules for the structure of a document. The DTD defines exactly what is allowed to appear inside a document. Appendix A provides an introduction to XML. SCA 2.2 Attachment 1 To Appendix D of the SCA contains the JTRS DTDs.

2 SCA Overview

2.1 Introduction

This section provides an overview of the SCA with emphasis on the Core Framework (CF) Base Application, Framework Control, and Framework Services Interfaces. The CF is the essential (“core”) set of open application-layer interfaces and services that provide an abstraction of the underlying software and hardware layers for software application designers. The CF consists of:

- ?? Base Application Interfaces (*Port, LifeCycle, TestableObject, PropertySet, PortSupplier, ResourceFactory, and Resource*) that can be used by all software applications
- ?? Framework Control Interfaces (*Application, ApplicationFactory, DomainManager, Device, LoadableDevice, ExecutableDevice, AggregateDevice and DeviceManager*) that provide control of the system
- ?? Framework Services Interfaces (*File, FileSystem, FileManager*) that support both core and non-core applications, and
- ?? A Domain Profile that describes the properties of hardware devices (Device Profile) and software components (Software Profile) in the system.

The SCA is not a system specification, as it is intended to be implementation independent, but a set of rules that constrain the design of systems. The OE, consisting of the Core Framework, CORBA middleware, and OS, imposes design constraints on waveform and other applications to provide increased portability of those applications from one SCA-compliant radio platform to another. These design constraints include specified interfaces between the Core Framework and application software, and restrictions on waveform usage of Operating System APIs.

The SCA also provides a building block structure (defined in the API Supplement) for defining application software component APIs. The building-block structure for API definition facilitates component-level reuse and allows significant flexibility for developers to define waveform-specific APIs.

³ “[XML Step By Step](#)” (Microsoft Press) Micheal J. Young, p7

2.2 **Architecture Definition Methodology.**

The architecture has been developed using an object-oriented approach depicted with UML diagrams. Color-coding is used to differentiate between architecture elements and applications in diagrams as shown in Figures 2.3-1 and 2.3-2.

	Core Framework (CF) elements
	Commercial-Off-The-Shelf (COTS) components
	Host Applications
	Red Side Network and Link Applications
	Security Applications
	Black Side Network and Link Applications
	Modem Applications
	RF

2.3 Architecture Overview.

The structure of the software architecture is shown in figure 2.3-1. The key benefits of the software architecture are that it:

- 1) Maximizes the use of commercial protocols and products,
- 2) Isolates both core and non-core applications from the underlying hardware through multiple layers of open, commercial software infrastructure, and
- 3) Provides for a distributed processing environment, through the use of CORBA, to provide software application portability, reusability, and scalability.

The CF Module specification includes a detailed description of the purpose of each interface, the purpose of each supported operation within the interface, and interface class diagrams to support these descriptions.

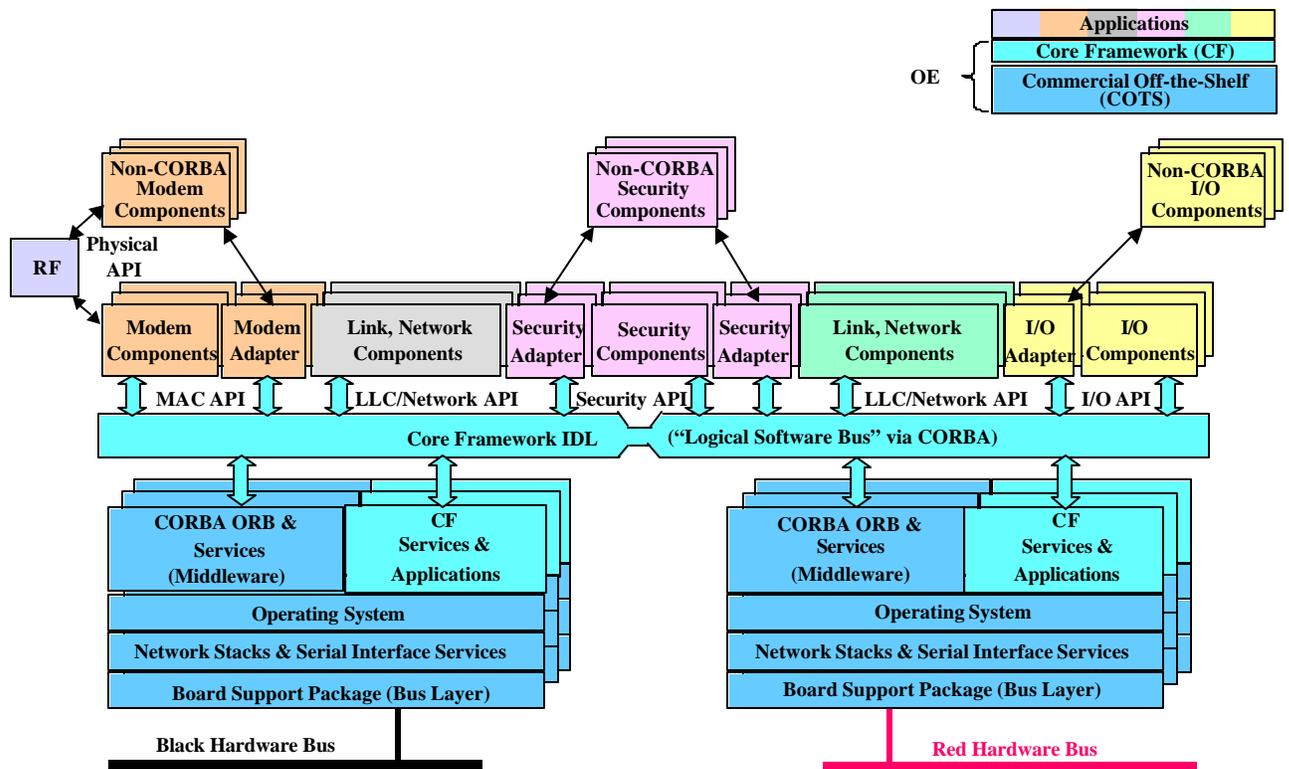


Figure 2.3-1 Software Structure

Figure 2.3-2 depicts the key elements of the CF and the IDL relationships between these elements. The interfaces enclosed by the red box are the ones the waveform developer would "realize" (implement). The Device developers are most concerned with those interfaces enclosed by the blue box. The remaining interfaces are the responsibility of a Core Framework product.

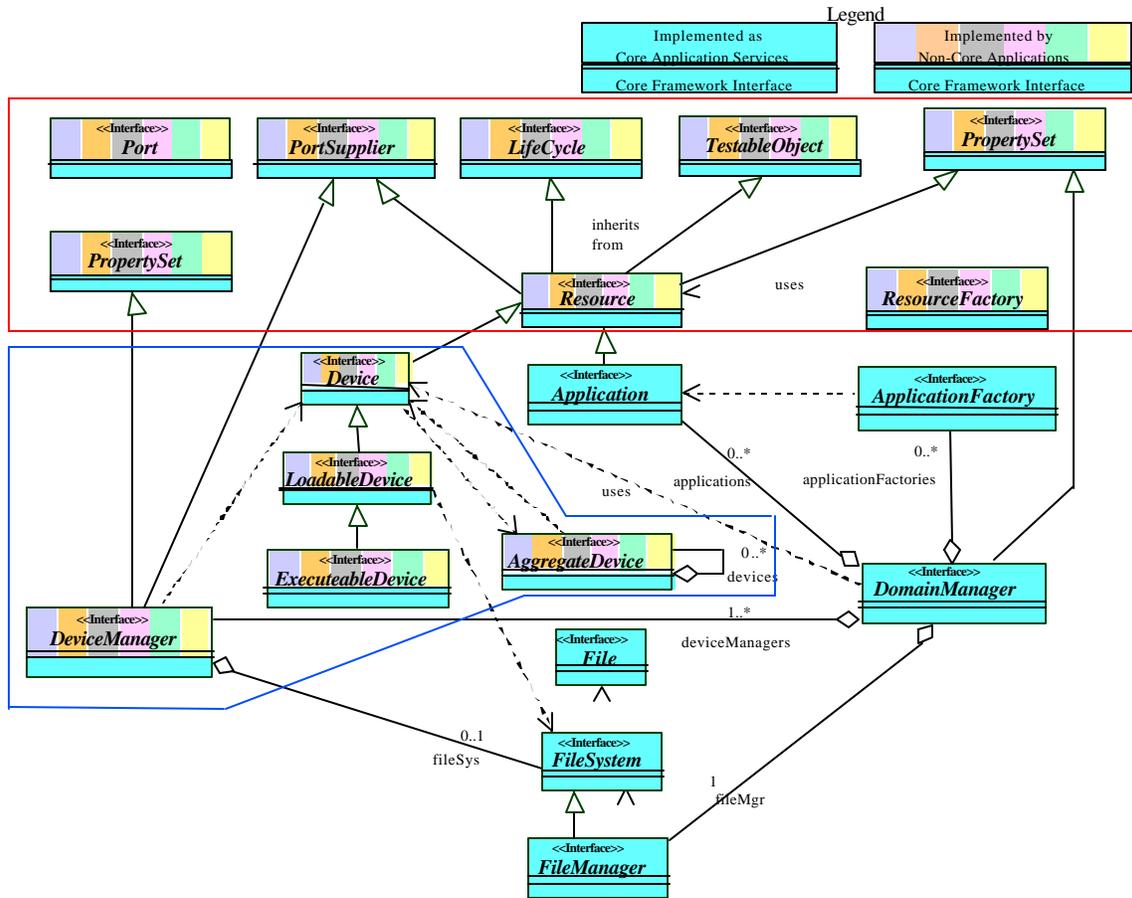


Figure 2.3-2 Core Framework IDL Relationships

2.4 Framework Interfaces

This section gives a brief overview of the Framework Control Interfaces (*DomainManager*, *ApplicationFactory*, and *Application*) that provide control of the system. These interfaces manage the registration/unregistration of applications, devices, and device managers within the domain and the controlling of applications within the domain. The implementation of the *Application*, *ApplicationFactory*, and *DomainManager* interfaces are coupled together and must therefore be delivered together as a complete domain management implementation and service. The Framework Services Interfaces (*File*, *FileSystem*, *FileManager*) that support both core and non-core applications are also discussed in this section.

2.4.1 DomainManager

The *DomainManager* interface is for the control and configuration of the system domain. It is logically grouped into three categories: Human Computer Interface (HCI), Registration, and CF administration.

- ?? The HCI operations are used to configure the domain, get the domain's capabilities (*Devices*, *Services*, and *Applications*), and initiate maintenance functions. These operations are invoked by an HCI-client capable of interfacing to the *DomainManager*.
- ?? The registration operations are used to register/unregister *DeviceManagers*, *DeviceManager Devices*, *DeviceManager Services*, and *Applications* at startup or during run-time for dynamic device, service, and application extraction and insertion.
- ?? The administration operations are used to access the interfaces of registered *DeviceManagers* and the *DomainManager's FileManager*.

2.4.2 ApplicationFactory

The *ApplicationFactory* interface provides the Domain Management interface to request the creation of a specific type of *Application* in the domain. The *ApplicationFactory* interface class is based on the OMG Factory Design Pattern. The Software Assembly Descriptor profile (discussed further in section 6.5) determines the type of *Application* that is created by the *ApplicationFactory*.

2.4.3 Application

The *Application* interface provides the Domain Management interface for the control and configuration of an instantiated application in the domain. The *Application* interface class inherits the IDL interface of *Resource*. A created application instance may contain *Resource* components and/or non-CORBA components. An application is created by the *ApplicationFactory* create operation. The *Application* is a proxy for the application's assemblycontroller and application's software components that were created.

2.4.4 File

The *File* interface provides the ability to read and write a file residing within a CF-compliant, distributed *FileSystem*. A file can be thought of conceptually as a sequence of octets with a current file pointer describing where the next read or write will occur. The file pointer points to the beginning of the file as a result of the execution of the file implementation software.

2.4.5 FileSystem

The *FileSystem* interface defines CORBA operations that enable remote access to a physical file system. The *FileSystem* interface provides the traditional operations associated with file accesses (i.e. remove, copy, directory listing, etc.).

2.4.6 FileManager

Multiple, distributed *FileSystems* may be accessed through a *FileManager*. The *FileManager* interface appears to be a single *FileSystem*, although the actual file storage may span multiple physical file systems. This is called a federated file system. A federated file system is managed using the *mount* and *unmount* operations. Typically, the *DomainManager* or system initialization software invokes these operations. The *FileManager* inherits the IDL interface of a *FileSystem*. Based upon the pathname of a directory or file and the set of mounted *FileSystems*, the *FileManager* delegates the *FileSystem* operations to the appropriate *FileSystem*. For example, if a *FileSystem* is mounted at `/ppc2`, an *open* operation for a file called `/ppc2/profile.xml` would be delegated to the mounted *FileSystem*. The mounted *FileSystem* is given the filename relative to it. In this example the *FileSystem*'s *open* operation would receive `/profile.xml` as the `fileName` argument.

If a client does not need to mount and unmount *FileSystems*, it can treat the *FileManager* as a *FileSystem* by CORBA widening a *FileManager* reference to a *FileSystem* reference. One can always widen a *FileManager* to a *FileSystem* since the *FileManager* is derived from a *FileSystem*.

2.5 Base Application Interfaces

This section discusses the Base Application Interfaces (*Port*, *LifeCycle*, *TestableObject*, *PropertySet*, *PortSupplier*, *ResourceFactory*, and *Resource*) that can be used by all software applications.

In several interface operations, a "*Properties*" parameter is required. In the SCA context, "*Properties*" refers to a CORBA sequence of id, value pairs. Within each pair, the first element is a string naming a property and the second element provides a value for that property.

2.5.1 Port

The *Port* interface provides operations for managing associations between ports. Transferring one object's reference to another object is a common occurrence in distributed programming. In a CORBA environment, a reference to an object can be obtained by using dynamic stringified IORs or the Naming Service. The SCA provides the *Port* interface to distribute the reference for any object to another object. Figure 2.5-1 depicts the *Port* interface.

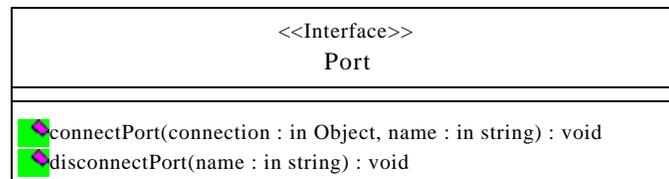


Figure 2.5-1 Port CORBA Interface UML

Before describing how ports are connected, the terms *uses-port* and *provides-port* must be explained. Both are called ports, but only the *uses-port* needs to implement the *Port* interface (due to its need for the *connectPort* operation). A *provides-port* is a type of port that provides a user-defined CORBA interface – it does not need to implement the *Port* interface. The intent, then, is to connect the *uses* and *provides* ports allowing the *uses-port* access to the *provides-port* operations.

When a client desires to connect two ports, several things must take place. A client must obtain the reference to two component *Resource* interfaces, which inherit the *PortSupplier* interface. One of the ports must be a *uses-port* and the other a *provides-port*. The client calls the *getPort* operation (on the *PortSupplier* interface) for each component in the desired connection. The *getPort* operation returns a

CORBA object reference. So, in the case of the uses -port, this object reference must be narrowed to a Port interface. For the provides -port, the CORBA object reference returned from getPort needs no modification.

The client calls the connectPort operation on the uses -Port, passing it a connection id and the provides -port CORBA object reference. A one-way connection has now been established. One-way, here, means client to server, not the typical messaging approach intended for building unreliable signaling mechanisms (the send-and-forget approach).

The ApplicationFactory has the requirements to establish the connections for application components during the instantiation process of the application. . The connections to be established are described in the application's Software Assembly Descriptor (SAD) file.

Figure 2.5-2 is a sequence diagram depicting a simple one-way connection of two ports.

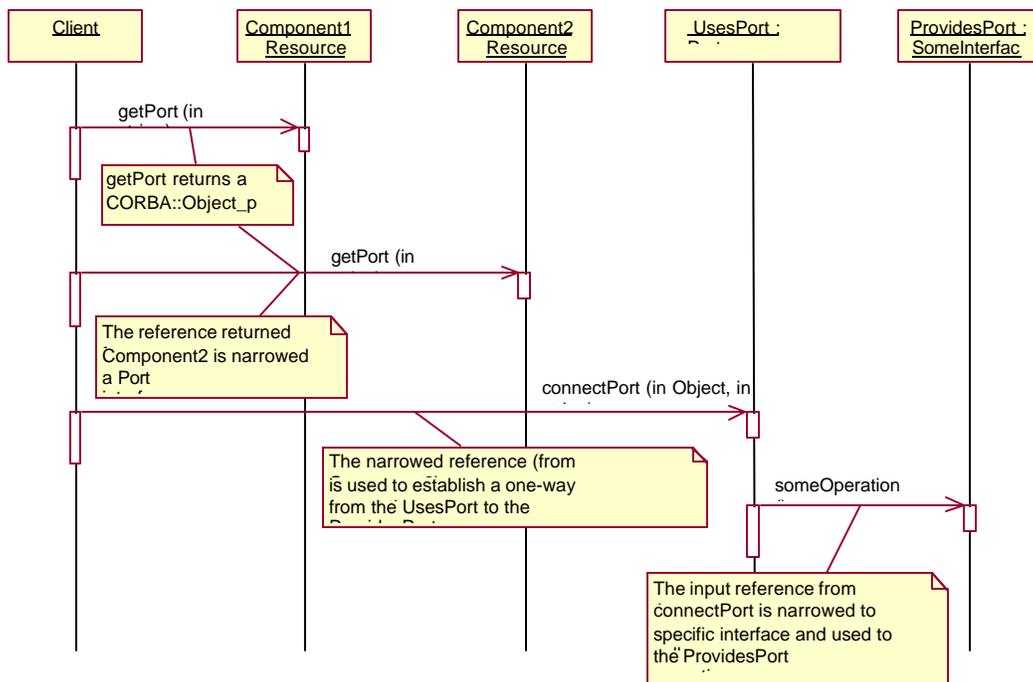


Figure 2.5-2 Sequence Diagram of One Way Connection

2.5.2 LifeCycle

The LifeCycle interface, which is depicted in Figure 2.5-3, provides generic operations for managing initialization and termination of a specific object.

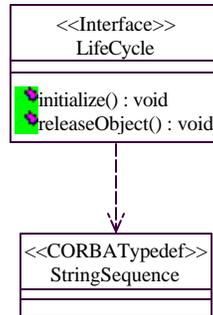


Figure 2.5-3 LifeCycle Interface UML

The LifeCycle interface provides the following operations:

initialize() : void

The initialize operation is invoked on a created component of a Resource type in order to set it to its initial, operative state. The ApplicationFactory implementation that created the Resource is responsible for its initialization. This operation performs tasks that must occur after the component's constructor has terminated but before the component is used. For example, the component's *this* pointer may be needed for some purpose, but the *this* pointer is stable only after the constructor has terminated.

releaseObject() : void

The CF ApplicationFactory implementation that is associated with the target Resource invokes this operation as part of the Application tear down sequence prior to unloading the Resource. This operation should free internal memory, close files, and perform whatever other tasks are needed to return the system to its default state.

2.5.3 TestableObject

The TestableObject interface, which is depicted in Figure 2.5-4, provides a means of performing Built-In Tests (BIT) with respect to a specific object.

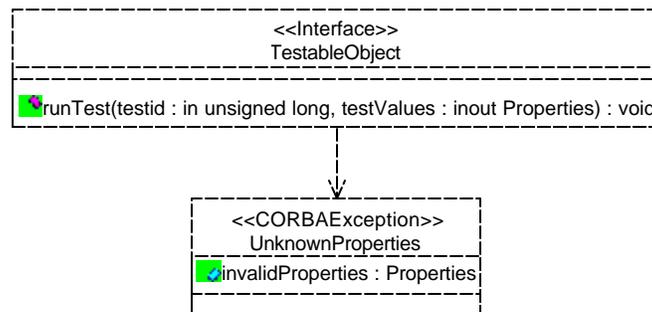


Figure 2.5-4 TestableObject Interface UML

The TestableObject interface provides the following operation:

runTest(testid : in unsigned long, testValues : inout Properties) : void

The runTest operation is typically invoked as part of target Resource testing at the operator's console as part of BIT (Built-In Tests). The parameter testid specifies which test is to be run. Inputs into the testing process are provided as id,value pairs in testValues. Parameters testid and testValues are described in the appropriate properties XML file, but interpretation of both parameters is completely component dependent, as provided in the component's implementation of these functions.

2.5.4 PortSupplier

The PortSupplier interface, which is depicted in Figure 2.5-5, provides means of obtaining a reference to a specific port for a specific object.

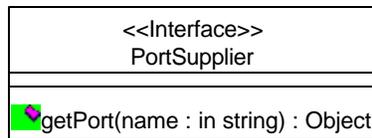


Figure 2.5-5 PortSupplier Interface UML

Figure 2.5-2 illustrates the use of getPort in the context of connecting a uses port to an appropriate produces port. The PortSupplier interface provides the following operation:

getPort(name: in string) : Object

The getPort operation returns a object reference (either a Uses or a Provides port).

The returned Port object reference can be used to establish a connection between Resource components. The getPort operation can be invoked by an external CF client on a CF Application object returning Ports that are defined as external by the Application's SAD XML file. The getPort operation is also invoked by the ApplicationFactory implementation that has established a Resource object as part of an overall software application when connections are documented in the Application's SAD XML file.

The getPort implementation returns a CORBA::Object pointer to the component where the behavior associated with the specified port is implemented. The instructions which receive this pointer may need to narrow it to a pointer to an appropriate interface, so often the pointer is generate through use of an ***Interface::_this()*** call, where ***Interface*** is the appropriate IDL interface.

2.5.5 PropertySet

The PropertySet interface, depicted in Figure 2.5-6, provides a means of accessing attributes of a specific object.

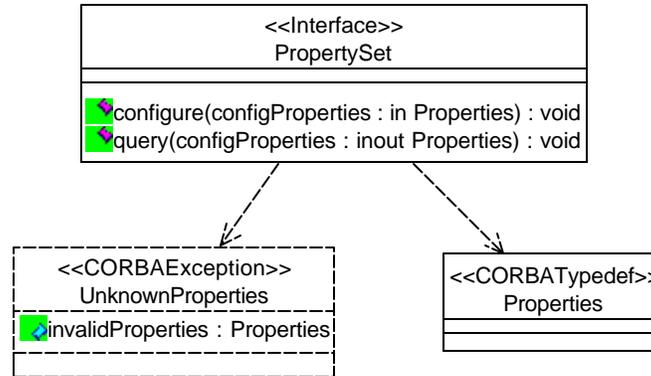


Figure 2.5-6 PropertySet Interface UML

The PropertySet interface provides the following operations:

configure(configProperties : in Properties) : void

The configure operation provides a mechanism to set the current values of configuration parameters identified within the Property XML file of a Resource object. In a typical application, changing configuration parameters is a mean of affecting operating characteristics of the component. The configure operation of a Resource component may be called by any external CF client.

When the Core Framework creates Resources as part of an Application instantiation, the CF Application implementation is required by the SCA to delegate all configure requests to the Assembly Controller Resource of the Application. The implementation of an Assembly Controller may in turn delegate requests to other Resources within the Application. When the Application developer needs to have external capability of configuring or querying other Application Resources (besides the Assembly Controller) those Resources may be provided as “ports” by the Assembly Controller. In this case, the ports for the other Resources will show up in the SAD externalports element.

This operation is also invoked by an ApplicationFactory implementation following the initialization (initialize method) of a newly created Resource object during the Application create process. The ApplicationFactory implementation establishes the preliminary settings for the recently created object based upon those configuration parameters established within the Property XML file of the Resource.

query(configProperties : inout Properties) : void

As a counterpart to configure, the query operation retrieves the current values of configuration parameters identified within the Property XML file of a Resource object. The query operation of a Resource component may be called by any external CF client. A CF Application implementation is required by the SCA to delegate all query requests to the Assembly Controller Resource of the Application. However, any Application Resource component may receive a query if the Assembly Controller Resource delegates the query to it.

2.5.6 Resource

The *Resource* interface (depicted in Figure 2.5-7) provides a common API for the control and configuration of a software component. The *Resource* interface inherits the *LifeCycle*, *PropertySet*, *TestableObject*, and *PortSupplier* interfaces. The *Resource* interface may be inherited by other application interfaces as described in the Software Profile's Software Component Descriptor (SCD) file.

An Application is comprised of one to many components. A component can realize the Resource Interface, in which case, the interface provides a common method of configuring, querying, etc. the component.

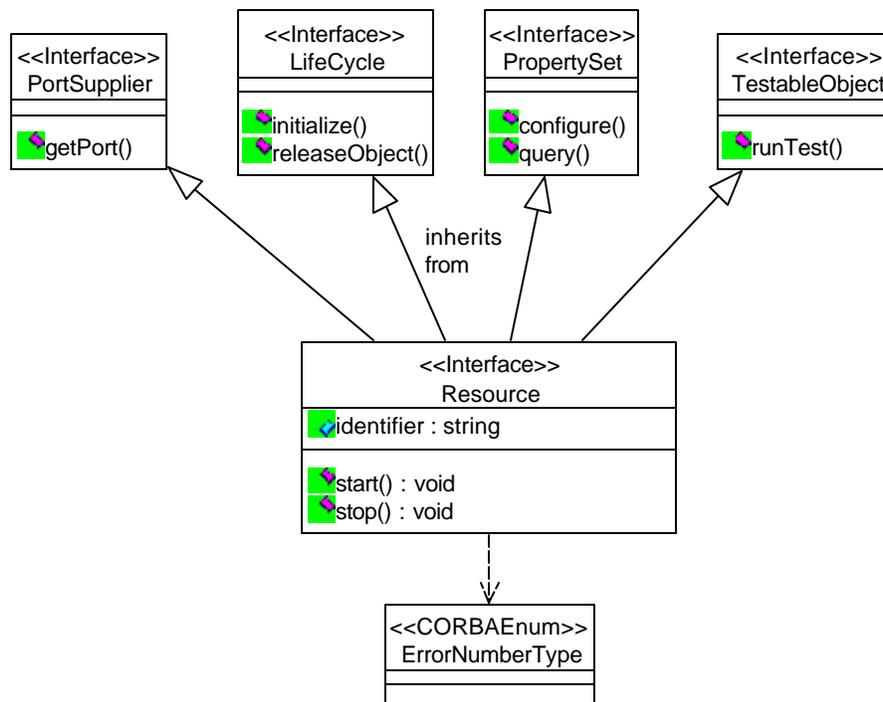


Figure 2.5-7 Resource CORBA Interface UML

The SCA includes requirements specifying when certain operations may be performed. Although the SCA itself provides no terminology to label these situations, application developers commonly label them by distinguishing between IDLE state (when only **configure**, **query**, and **start** may be performed) and OPERATIONAL state (when any operation may be performed). Actual enforcement of the "IDLE state" limitations is left to the developer. Often this is accomplished by spawning a special class that realizes only **configure**, **query**, and **start** behavior (or by having each function base its behavior upon an attribute that denotes the current state).

The Resource interface provides the following operations:

start() : void

The start operation is used to move a created Resource to OPERATIONAL state from IDLE state. The start operation may be called by any external CF client.

stop() : void

The stop operation is used to move a created Resource to IDLE state from OPERATIONAL state. The stop operation may be called by any external CF client.

2.5.7 ResourceFactory

A *ResourceFactory* may be used to create and tear down a *Resource*. The *ResourceFactory* interface is designed after Factory Design Patterns. The *ResourceFactory* interface UML is depicted in Figure 2.5-8. The *ResourceFactory* interface provides a standard API for obtaining, creating, and destroying Resources within an Application. If a Resource within an Application needs to be executed multiple times, the *ResourceFactory* interface could be used in the Application developer's design and implementation.

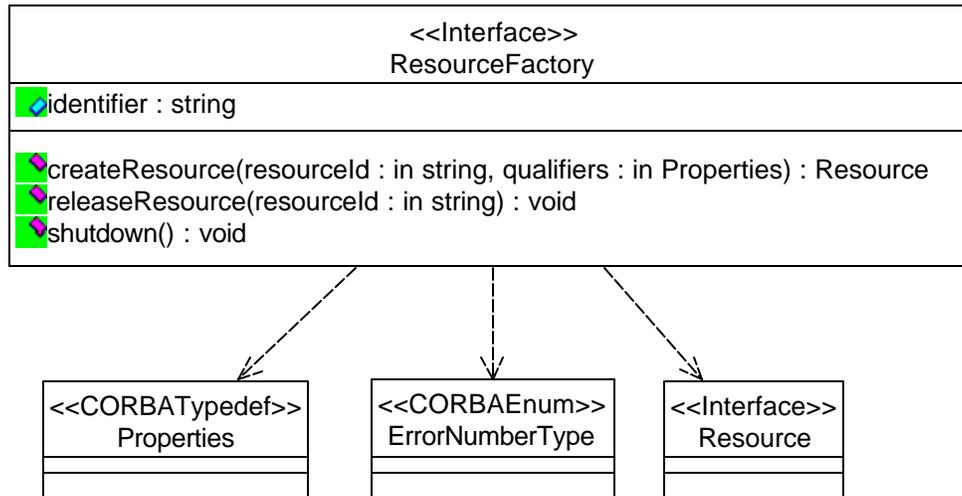


Figure 2.5-8 ResourceFactory CORBA Interface UML

The ResourceFactory interface provides the following operations:

createResource() : Resource

If a Resource with the specified resourceId does not exist, the createResource operation creates the Resource with a reference count⁴ of 1 and returns a reference to that Resource. If a Resource with the specified resourceId does exist already, the createResource operation increments the Resource's reference count by 1 and returns a reference to the Resource.

releaseResource() : void

The releaseResource operation decrements the Resource's reference count; if the reference count now equals zero, the operation then releases the Resource from the CORBA environment. In any case, the client has responsibility to release its reference to the Resource.

shutdown() : void

The shutdown operation terminates the ResourceFactory implementation on the server side. The client has responsibility to release its reference to the ResourceFactory.

The *ResourceFactory* interface is an optional interface that can be used by an application developer. In the most common case, a ResourceFactory is not provided within an Application, so the Core Framework's ApplicationFactory implementation is used to construct the application.

⁴ reference count keeps track of how many clients are using the server-side entity.

2.6 Service Interfaces

2.6.1 Naming Service

The OMG Naming Service is one of CORBA's standardized services. It provides a mapping from names to object references: given a name, the service returns an object reference stored under that name. This is similar to the Internet Domain Name Service (DNS), which translates an Internet domain name (such as acme.com) into an IP address (such as 234.234.234.234)⁵.

The Naming Service provides a number of advantages to clients (a client of the Naming Service):

- ?? Clients can use significant names for objects instead of dealing with stringified object references.
- ?? By changing the value of a reference advertised under a name, you can get clients to use a different implementation of an interface without having to change source code. The clients use the same name but get a different reference.
- ?? The Naming Service can be used to solve the problem of how application components get access to the initial references for an application. Advertising their references in the Naming Service eliminates the need to store them as stringified references in files.

The Naming Service maps names to object references. A name-to-reference association is called a *name binding*. The same object reference can be stored several times under different names, but each name identifies exactly one reference. A *naming context* is an object that stores name bindings.

A hierarchy of contexts and bindings is known as a *naming graph*. In the graph below (Figure 2.6-1), hollow nodes are naming contexts and solid nodes are application objects. Within a particular context, name bindings are unique (each binding can appear only once within its parent context). The sequence of bindings used in the traversal forms a pathname that uniquely identifies the target object. The same name binding can appear multiple times provided that each binding is in a different parent context. A single object or context can have multiple names. It is possible for the graph to have contexts that have no names. Such contexts are known as *orphaned* contexts.

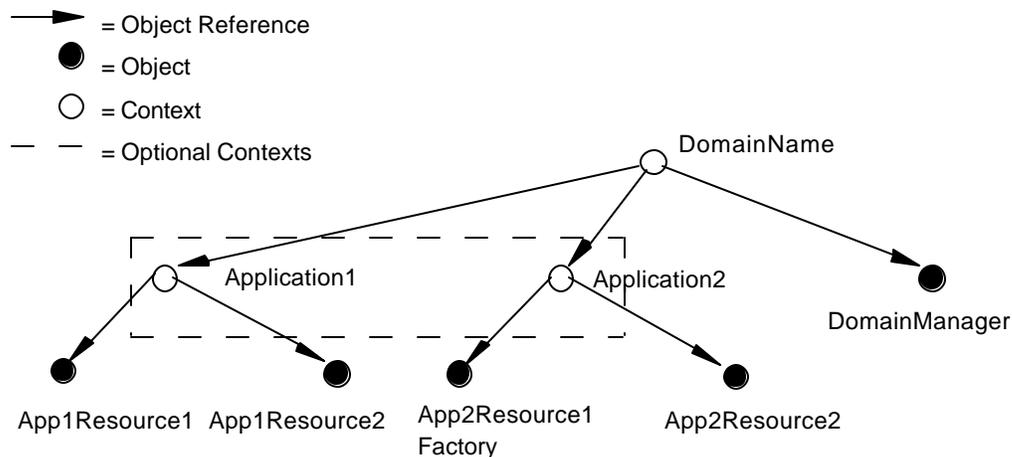


Figure 2.6-1 Naming Graph

⁵ “Advanced CORBA Programming With C++” (Addison-Wesley Professional Computing) Henning & Vinoski p772

Because of critical size and tight processing requirements, it is difficult to incorporate a full-featured OMG Naming Service into a real-time embedded application domain, such as a JTRS radio. The requirements of these types of systems differ vastly from other general-purpose transaction-based CORBA applications. Removing interfaces and methods not required for the embedded real-time environment would allow Naming Service functionality to be more economically sized. The SCA Naming Service is a 'lightweight' version of the OMG Interoperable Naming Service. When there were two methods for accomplishing a required task, one method was supported, and the other dropped. The OMG Naming Service List and BindingIterator interfaces were omitted, since they were deemed as being unnecessary for most embedded domain use. The following list summarizes the required interfaces supported by an SCA Naming Service:

- ?? The bind Interface
- ?? The bind_new_context Interface
- ?? The resolve Interface
- ?? The unbind Interface
- ?? The Destroy Interface

The following OMG Naming Service interfaces are not supported in the SCA's Naming Service definition and should not be used:

- ?? The rebind Interface
- ?? The rebind_context Interface
- ?? The bind_context Interface
- ?? The new_context Interface
- ?? The list Interface
- ?? The BindingIterator Interface

2.6.1.1 Use of Naming Service

All components managed by a particular DomainManager must use the same Naming Service, but that Naming Service can be located on any processor that is accessible to the components. If the Naming Service and other facilities provided by the ORB vendor conform to the OMG standard, then calling an ORB's *resolve_initial_reference* operation will obtain a connection to the Naming Service. Proprietary methods may be available for connecting to the Naming Service, but these methods do not meet SCA requirements.

In general, an object that wishes to 'publicize' its object reference registers with the Naming Service. Within a JTRS system, the SCA requires the CF DomainManager and Application (waveform) Resources to publicize their respective IORs in the Naming Service. No other JTRS component (CF or waveform) is required to store its IOR on the Naming Service.

A Naming Service's NameComponent structure is used to identify a context or an object's IOR binding to a context. A Naming Service's NameComponent structure is made up of an id-and-kind pair. For JTRS, the SCA requires the "id" element of each NameComponent to be a string value that uniquely identifies a NameComponent and the "kind" element of the NameComponent be "" string (null string).

During component construction, the *DomainManager* creates a "naming context" under the root Naming Service context using *"/DomainName"* as its name.ID component and "" (Null string) as its name.kind component. It then creates a "name binding" to the *"/DomainName"* naming context using *"/DomainManager"* as its name.ID component, "" (Null string) as its name.kind component, and the *DomainManager's* object reference.

For a waveform Resource, the context to which the Resource is required to bind its object reference is dictated by the CF ApplicationFactory. During the initial execution of a waveform, the ApplicationFactory create operation passes mandatory execute-parameters to a Resource's entry point. A Naming Context IOR, a Name Binding, and the identifier for the Resource (in the form of *CF::Properties*) are passed to the

entry point of the *Resource* components to be executed via a *Device's* execute operation. For the Naming Context IOR parameter the *CF::Properties* ID element is set to the string of "NAMING_CONTEXT_IOR" and the *CF::Properties* value element is set to the stringified IOR of a naming context to which the component will bind its IOR. The *create* operation creates the naming contexts to which the component will bind its IOR. For the Name Binding parameter, the *CF Properties* ID element is set to the string of "NAME_BINDING" and *CF Properties* value element set to a string in the format of "ComponentName_UniqueIdentifier". The ComponentName value is the SAD *componentinstantiation findcomponent namingservice* element's name attribute. The UniqueIdentifier is determined by the implementation of the ApplicationFactory. The Name Binding parameter is used by the component in the naming structure when binding its object reference to the Naming Context IOR parameter.

The process is reversed, and the component's name and reference are removed from the Naming Service's Naming Graph by the CF ApplicationFactory implementation, when the application is torn down.

The ApplicationFactory implementation performs these steps when it creates an application, so the waveform developer need be concerned with them only if the ApplicationFactory implementation is not used to build the waveform.

Figure 2.6-1 depicts an example listing from a JTRS utilizing the Naming Service. Optional application (waveform) context objects may be listed under the Domain context.

2.6.2 Log Service

A JTRS Log Service stores log records written to it. The stored log records can be retrieved and deleted from a Log. There can be any number of Log Services in a JTRS system. A log producer is a CF component⁶ or an application's CORBA capable component⁷ that enters records into a Log by calling the *writeRecords* operation of the Log interface.

A CF component or application component can write log records (e.g., messages) to a Log Service for storage. The stored log records can be retrieved as fault history, event history, general application messages, etc. There is no requirement that an application component write log records, but the Log Service does provide a standard mechanism if logging is desired.

A standard record type is defined for all log producers to use when writing log records. The log producer may be configured via the *PropertySet* interface to output log records only for specific log levels. Log producers implement a configure property with an ID of "PRODUCER_LOG_LEVEL". The PRODUCER_LOG_LEVEL configure property provides the ability to "filter" the log message output of a log producer. This property is of type *LogLevelSequence*. The configure property *LogLevelSequence* contains all log levels that are enabled. Only messages containing an enabled log level are sent by a log producer to a Log. Log levels that are not in the *LogLevelSequence* are disabled.

2.6.2.1 Use of Log Service

The *LogService* module contains the Log interface and the types necessary for a log producer to generate standard SCA log records. This module also defines the types necessary to control the logging output of a log producer. An SCA Log Service, as specified in this section, may be provided in a JTRS installation. The optional aspect of the Log Service is restricted to its deployment. A JTRS installation (e.g., a handheld platform with limited resources) may choose not to deploy a Log Service as part of its domain. Several CF components contain requirements to write log records using the Log Service. CF components and applications that are required to write log records are also required to account for the absence of a Log service and otherwise operate normally.

⁶ e.g., *DomainManager*, *Application*, *ApplicationFactory*, *DeviceManager*, or *Device*

⁷ e.g., *Resource*, *ResourceFactory*

Once a Log Service has been registered with the DomainManager, the Log Service can then be used by components in the system. Log Services are registered with the DomainManager by type (i.e. Log) and name (e.g. Security, Failure Log). For waveform applications, the DomainManager utilizes the XML *connections* element of the Application's Software Assembly Descriptor (SAD). The connection element documents uses and provides ports to be connected together. The connection of a component to a Log is established using this method. A SAD file *connection / providesport / findby / domainfinder* XML can contain attributes of type = "Log" and name = "Test". The DomainManager would connect the Log named Test to the component using the *getPort / connectPort* mechanism. If the name attribute is not supplied in the XML, a null reference is provided to the Resource component. If the requested Log is not registered in the system the DomainManager keeps the requesting pending and is required to perform the connection when the appropriate Log Service registers.

Log Service connections to Devices are established in the same manner as connections to application (waveform) Resources. When a DeviceManager or Device registers with the DomainManager the DomainManager performs the connections documented in the DeviceManager's Device Configuration Descriptor (DCD) file. The DCD XML *connections* element is used in the identical manner as in an Application's SAD file.

The Log Service connections for the DomainManager are documented in the DomainManager's XML Configuration Descriptor (DMCD) file. The DMCD file contains the XML *service / findby* elements which dictate the type and name of the Log Service to utilize for logging. The DomainManager is required to log to the Log Service(s) dictated by the DMCD file.

This method of connecting ports of a Resource to a service is identical for the connection of both the Event and Log Service. Section 2.6.3.1, Use of Event Service, provides sample XML depicting the XML for describing the connection of a service to a CF component or Resource component.

2.6.3 CORBA Event Service

The OMG Event Service allows applications to use a de-coupled communications model rather than strict client-to-server synchronous request invocations. With synchronous requests, a client actively invokes requests on passive servers - After sending a request, the client blocks waiting for the response. Clients are aware of the destinations of requests because they hold object references to the target objects, and each request has a single destination denoted by the object reference used to invoke it. The OMG Event Service allows suppliers to send messages to one or more consumers with a single call. In fact, suppliers using an implementation of the Event Service need not be aware of any of the consumers of its messages; the Event Service implementation also shields suppliers from exceptions resulting from any of the consumer objects being unreachable or poorly behaved.

In the OMG Event Service Model, suppliers produce events and consumers receive them. Both suppliers and consumers connect to an event channel, which conveys events from suppliers to consumers without requiring suppliers to know about consumers or vice versa. Event channels play a central role in an Event Service. They are responsible for supplier and consumer registration, timely and reliable event delivery to all registered consumers, and the handling of errors associated with unresponsive consumers.

The OMG Event Service provides two models for event delivery: the push model and the pull model. The SCA, however, only uses the push model. With the push model, suppliers push events to the event channel, and the event channel pushes events to consumers. Figure 2.6-2 illustrates the push style of event delivery. Note that the arrows indicate the client and server roles and point from client to server.

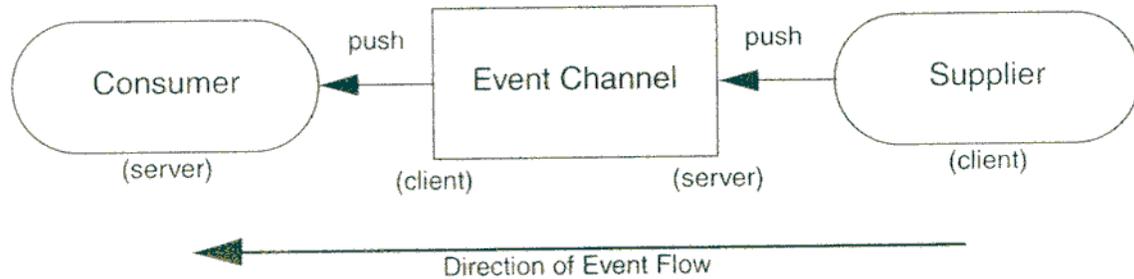


Figure 2.6-2 OMG Event Service – Event Channel Models⁸

2.6.3.1 Use of Event Service

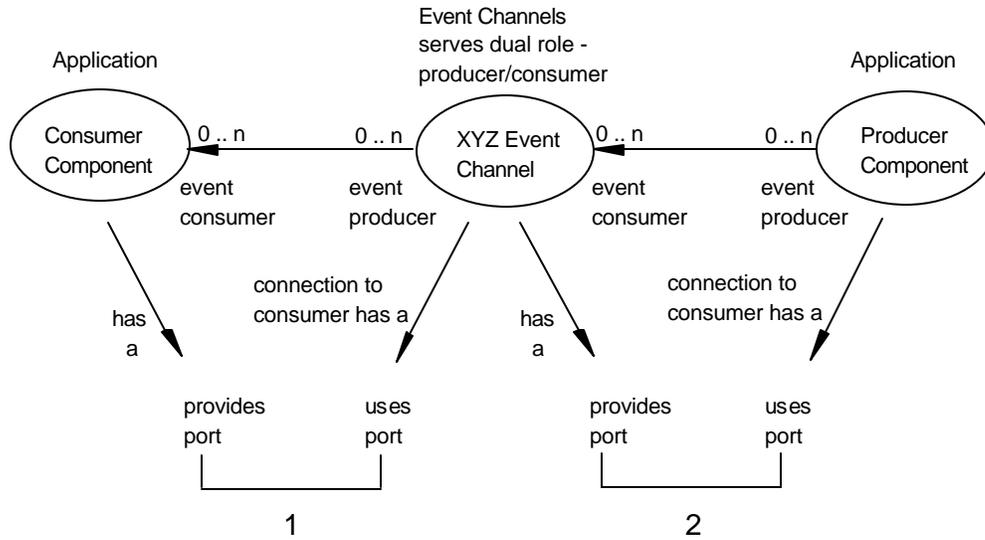
A CF implementation is required to provide two standard event channels: Incoming Domain Management and Outgoing Domain Management. The standardized name of the Incoming Domain Management Channel is "IDM_Channel". The Incoming Domain Management event channel is used by components (e.g., *Device* state change event) within the domain to generate events that are consumed by domain management components (e.g., *ApplicationFactory*, *Application*, *DomainManager*, etc.) The standardized name of the Outgoing Domain Management Channel is "ODM_Channel". Domain clients (e.g., HCI) use the Outgoing Domain Management Channel to receive Domain Management events generated from domain management components (e.g., *ApplicationFactory*, *Application*, *DomainManager*, etc.). The *DomainManager* creates the standard event channels upon startup.

Besides these two standard event channels, the OE allows other event channels to be set up for Application usage. An event channel provides an asynchronous transfer of data between components. It is not recommended that an event channel be used for real-time transfers; instead, it is recommended that a specific waveform API be used for real-time activity. Event channels are intended for non real-time messages and are best used when developing a specific API would be impractical.

The definition of a nonstandard event channel is accomplished through an Application's Software Assembly Descriptor (SAD) file (Section 4.4). An Application's SAD file can contain a domainfinder element within a connection element specifying the interconnection of a component (e.g., Resource) to a service. The *domainfinder* element is a child element of the *findby* element. The *domainfinder* element is used to indicate to the CF ApplicationFactory implementation the necessary information to find an object reference that is of specific type and may also be known by an optional name within the domain. The type attribute value of "eventchannel" is used to specify the event channel to be used in the CF Event Service for producing or consuming events. If the name attribute is supplied, but the name is not one of the standard event channel names (IDM_CHANNEL or ODM_CHANNEL) and the type attribute has a value of "eventchannel", then the CF ApplicationFactory implementation will create the specified channel. If the name attribute is not supplied and the type attribute has a value of "eventchannel" then the Incoming Domain Management event channel is used. This method of connection ports of a Resource to a Service is identical for the connection to both the Event and Log Service.

Figure 2.6-3 illustrates the event channel flow between producer and consumer. One concept this figure shows is that a consumer must supply its provides object reference to a producer so that it may consume what the producer provides.

⁸ "Advanced CORBA Programming With C++" (Addison-Wesley Professional Computing) Henning & Vinoski p931



XML for Consumer Component (1)

```
<connections>

<connectinterface>

<usesport>

  <usesidentifier>
    event_channel_consumer_port_that_can_be_any_name
  </usesidentifier>

  <findby>
    <domainfinder
      type="eventchannel"
      name="XYZ"/>
  </findby>
</usesport>

<providesport>

  <usesidentifier>consumer_event_in_port</usesidentifier>
  <componentinstantiationref
    refid="consumer_component"/>
</providesport>

</connectinterface>

</connections>
```

XML for Producer Component (2)

```
<connections>

<connectinterface>

<usesport>

  <usesidentifier>
    producer_event_out_port
  </usesidentifier>

  <componentinstantiationref
    refid="producer_component"/>
</usesport>

<providesport>
  <findby>
    <domainfinder
      type="eventchannel"
      name="XYZ"/>
  </findby>
</providesport>

</connectinterface>

</connections>
```

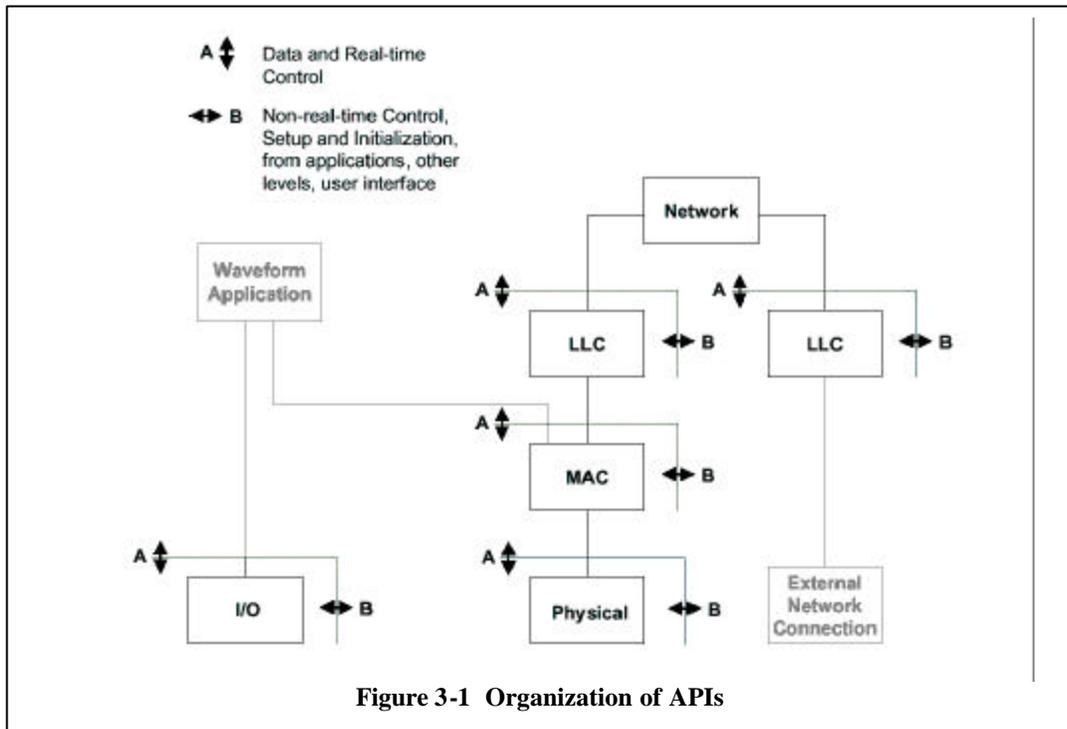
Figure 2.6-3 Event Channel Flow between Producer & Consumer Components

3 Application Program Interface (API) Overview

The API Supplement to the SCA defines structures to simplify the construction of portable SCA applications. Organized similarly to the OSI, each API contains related functionality as is shown in Figure 3-1. Each API contains "A" interfaces (data and real-time control carried by the data stream) and "B" interfaces (non-real-time control from the User Interface (UI), other layers, and other applications).

If an API is designed for a device, it includes interface CF::Device; otherwise it includes interface CF::Resource⁹. The API Supplement provides for all other interfaces needed by an SCA-compliant component.

A data / real-time control connection is labeled as being either "*downstream*" (moving messages towards the modem) or "*upstream*" (moving messages from the modem).



Each API is divided into small closely related partitions, called "Service Groups", allowing a designer to use just those portions that are needed for a particular application. A concrete Service Group is a complete class. A generic Service Group, also known as a "Building Block", is a template class for which one, or more, parameters must be replaced by a particular type in order to create a concrete class specific to the application. The process of associating an actual type with the parameter is known as "*binding*", and the process of creating a concrete class from a template class is known as "*instantiating*"¹⁰. A unified interface structure can be created by bringing together specific Service Groups as needed - the desired unifying class inherits from all relevant classes, so it contains all attributes and operations contained in those classes.

⁹ See Section 3.2.2.1 and Figure 3-3 of the API Supplement.

¹⁰ This is another example of terminology overloading, since this use of "instantiating" to mean "create a concrete class from a template class" is easily confused with the more common use of "create an object of a particular class".

The interfaces between two layers are defined by the lower layer.¹¹ Both directions must be defined - one interface for messages moving downstream¹², and one interface for messages moving upstream¹³. For example, the interfaces between the Physical and MAC layers are defined by the Physical layer. The downstream ("provider") interface is implemented by the Physical layer and is invoked by the MAC layer, and the upstream ("user") interface is implemented by the MAC layer and is invoked by the Physical layer.

3.1 Generic Packets

Generic Packets is a collection of Building Blocks (BB). As is implied by the name, these are generic classes that are used to define packet structure for any real-time interface. These classes/BuildingBlocks are tailored to handle any type of data, since type of data is a parameter used when binding the template class to create a concrete class. This grouping is comprised of the following Building Blocks:

- ?? SimplePacketBB - data transfer only
- ?? PacketBB - data transfer with Quality-of -Service and flow-control
- ?? ErrorSignalBB - asynchronous error notification
- ?? SignalsBB - notify when queue is empty or has reached high watermark or low watermark

3.2 Physical API

The Physical layer provides functionality directly related to operating a modem. In this particular case, the API Supplement separates "A" Interface and "B" Interface services (see Figure 3-1 above).

3.2.1 Physical Real Time

Physical Real Time Service Groups contain types and operations relating to control information carried by the data stream (actual packets are built from generic packets described in section 3.1). The following Service Groups are contained in this group:

- ?? TransmitPackets - "instantiation" of SimplePacketBB or PacketBB with a data class corresponding to the type of data actually received from the attached (MAC) layer.
- ?? ReceivePackets - "instantiation" of SimplePacketBB or PacketBB with a data class corresponding to the type of data actually sent to the attached (MAC) layer.
- ?? ReceiveCommand - specifics of command are contained in actual type bound to the parameter

3.2.2 Physical Non-Real Time

Physical Non-Real Time Service Groups contain types and operations relating to control information delivered apart from the datastream (typically from the UI). The following Service Groups are contained in this group:

- ?? AntennaControlBB - controls which antenna(s) are connected to the transceiver
- ?? ModulationSetupBB - controls modulation/demodulation type {AM, FM,...} and settings
- ?? MediaSetupBB - controls setup needed for voice, data, etc.
- ?? TransceiverSetupBB - controls characteristics which are covered by neither MediaSetupBB nor ModulationSetupBB
- ?? RadioModeBB - controls whether transceiver is off, operational, in setup mode, in test mode, etc.
- ?? ReceiveTerminationBB - used to terminate current reception and determine new status
- ?? Transmit_Inhibit - causes transceiver to be silent
- ?? PhysicalManagement - sets maximum and minimum Transmission Units

¹¹ See Figure 3-3 (page 3-7) of the API Supplement for an illustration of this rule.

¹² The API Supplement calls this the "Provider Interface"

¹³ The API Supplement calls this the "User Interface"

3.3 Medium Access Control (MAC) API

The MAC layer, which is responsible for configuring the elements that move data between applications, consists of the following Service Groups:

- ?? MACCommonUtilityBB - generalized methods needed by more than one MAC Building Block; one common use of this Building Block would be to provide preset channel and/or power levels
- ?? TRANSECBB - non-Type 1 TRANSEC operations
- ?? ChannelErrorControlBB - attempts to maintain integrity of message over the physical channel (error correction is an example of what could be provided by this Building Block)
- ?? ChannelAccessBB - channel access includes sync, end-of-message, and TDMA/CSMA/DAMA/etc
- ?? MACAddressBB - allows user to define address at MAC level
- ?? DropCaptureBB - facilitates returning to search state
- ?? QOSBB - allows user to access channel quality of service information when MAC calculates this
- ?? TransmitPackets - similar to corresponding Service Group in the Physical layer¹⁴
- ?? ReceivePackets - similar to corresponding Service Group in the Physical layer¹⁴

3.4 Logical Link Control API

The LLC layer provides three levels of service between end-points:

1. Connectionless service is a datagram service which
 - a. notifies the sender if an error is detected
 - b. provides flowcontrol between the end-points
2. Acknowledged Connectionless service is a datagram service which, in addition to services provided by connectionless service
 - a. provides acknowledgements so that sender can verify arrival of a message
 - b. delivers packets in the same order in which they were sent
3. Connection-mode service is not yet defined. When fully specified, this service will create a virtual circuit that provides convenient delivery between the end-points, including a "moving window" to enable one reply to acknowledge a group of messages.

The LLC layer currently consists of the following Service Groups:

- ?? LocalManagement - apply to both connectionless and connection-mode services
- ?? Connectionless Mode Data Transfer - applies to connectionless service only
- ?? Acknowledged Connectionless Mode Data Transfer - applies to acknowledged connectionless service only

3.5 I/O API

The I/O layer, which establishes the means of communicating with a specific type of I/O device, consists of the following Service Groups:

- ?? IOConfigurationBB - provides means of configuring specific I/O device
- ?? IOControlBB - provides means of controlling specific I/O device during operation
- ?? IOSignals - provides means for device to signal "downstream" component
- ?? Audible Alerts And Alarms - provides means to define audible signal as a sequence of single tones (resulting sound could be as complex as a siren, for example)

¹⁴ These Service Groups are not explicitly included in the "Cross-Reference of Services and Primitives" table for the MAC layer, but the accompanying text clearly indicates that they are included in MAC layer.

4 Domain Profile Components

The hardware devices and software components that make up an SCA system domain are described by a set of XML descriptor files that are collectively referred to as a Domain Profile. This section provides a high-level overview of each profile, emphasizing the purpose of each. This section will not go into detail on the various DTD elements for each profile – That type of detail can be found in Appendix D of the SCA. Figure 4-1 depicts the relationships between the various descriptor files used to describe a system's hardware and software assets. This section, however, only focuses on the descriptor files essential to an Application developer (Outlined in red in Figure 4-1) - These include the Software Package, Properties, Software Component, Software Assembly, and Profile descriptor files. The profiles applicable to a device developer (outlined in green in Figure 4-1) are discussed in section 7, Device Creation. A Software Profile is either a Software Assembly Descriptor (for applications) or a Software Package Descriptor (for all other software components and hardware devices). These descriptor files describe the identity, capabilities, properties, and inter-dependencies of the hardware devices and software components that make up the system. All of the descriptive data about a system is expressed in the XML vocabulary. This section includes a UML diagram of each root-element defined in the specified profile, along with some guidance on how to use the specified profile.

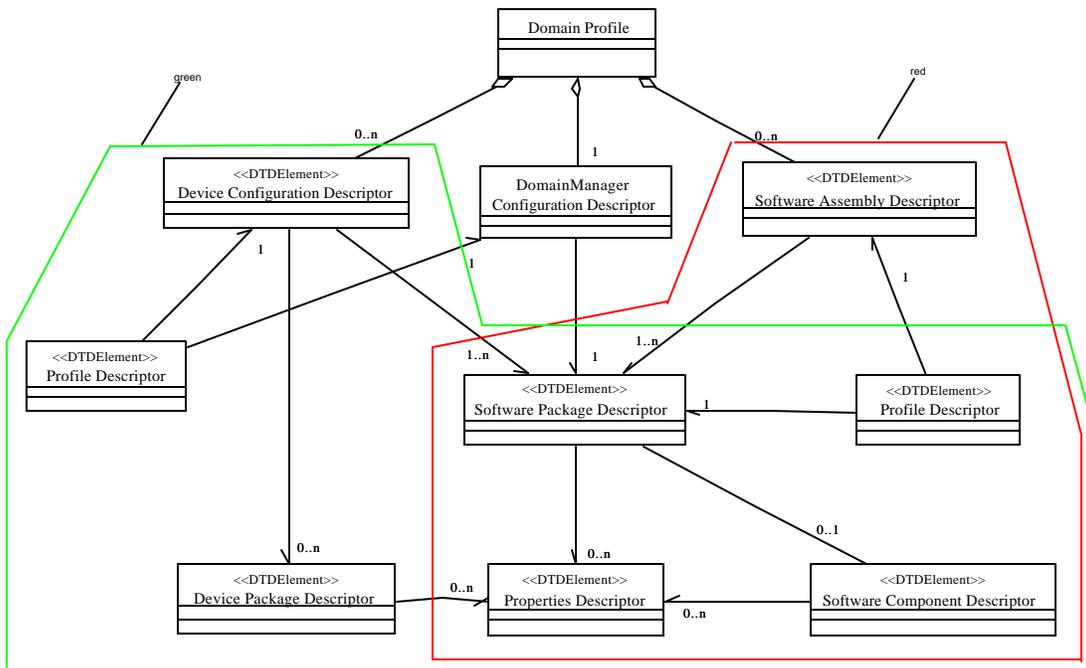


Figure 4-1 Domain Profile Descriptor File Relationships

4.1 Software Package Descriptor

The SCA Software Package Descriptor (SPD) is used at deployment time to load and/or execute an SCA compliant component. The information contained in the SPD provides the basis for the Domain Management function to manage the component within the SCA architecture.

The SPD may contain various deployment implementations of any given component. Within the specification of an SPD, several other files are referenced including a component-level *propertyfile* and a software component *descriptor* file.

The *softpkg* element defines an SPD. The *softpkg* id attribute uniquely identifies the package and is a DCE UUID, as defined by the DCE UUID standard (adopted by CORBA). The DCE UUID format starts with the characters "DCE:" and is followed by the printable form of the UUID, a colon, and a decimal minor version number, for example: "DCE:700dc518-0110-11ce-ac8f-0800090b5d3e:1". The decimal minor version number is optional. The version attribute specifies the version of the component. The name attribute is a user-friendly label for the *softpkg* element. The type attribute indicates whether or not the component implementation is SCA compliant. All files referenced by a Software Package are located in the same directory as the SPD file or a directory that is relative to the directory where the SPD file is located.

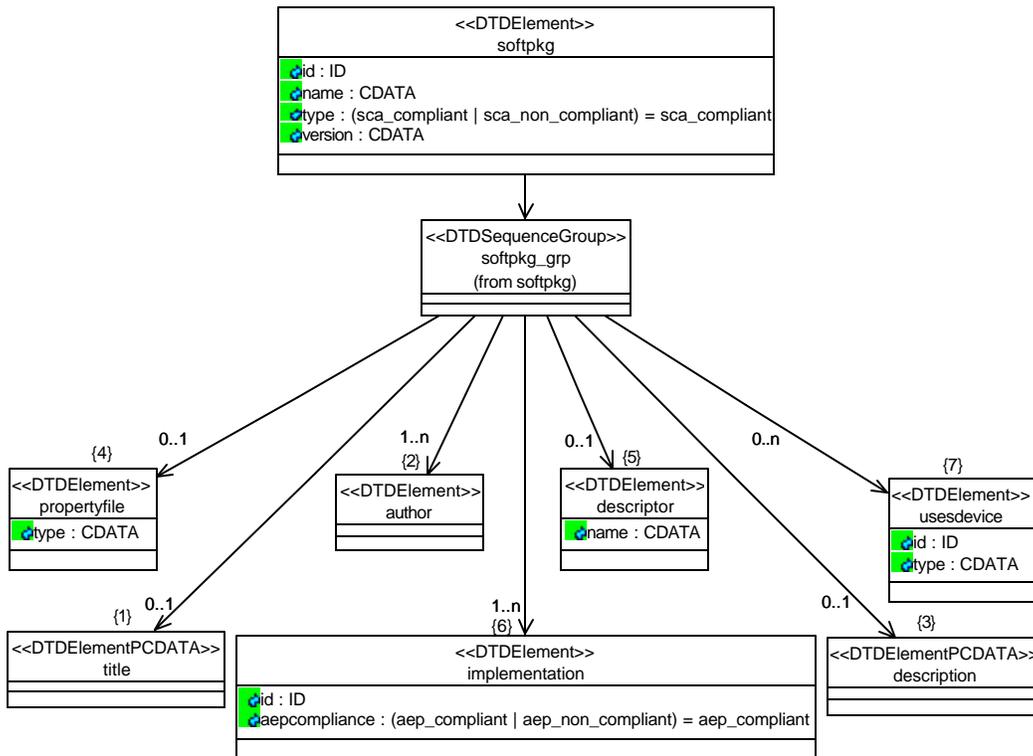


Figure 4.1-1 softpkg Element Relationships

An SPD should be provided as part of the software documentation for an application or device implementation. The SPD should contain all of the mandatory XML elements as well as many of the optional elements. There should be one implementation element for each variant (processor, operating system, etc) of the component. The propertyfile element of the SPD is optional but should be provided,

because it provides the definition of properties elements common to all component implementations and implementation-specific properties being deployed in accordance with the Software Package. The `usesdevice` element is not required, because an application may not use a device; however, there should be a `usesdevice` element for each device that is needed by the application.

4.2 Properties Descriptor

The Properties Descriptor file details component and device attribute settings. For purposes of the SCA, properties files contain *simple*, *simplesequence*, *test*, *struct* or *structsequence* elements. These elements are used to describe attributes of a component that are used for dependency checking.

The *simple* element is the property descriptor's central element. It provides for the definition of a property, which includes a unique id, type, name, and mode attributes. Also included in the simple element are the value, units, and range sub-elements, which are provided as initial configuration or execute parameters of a component.

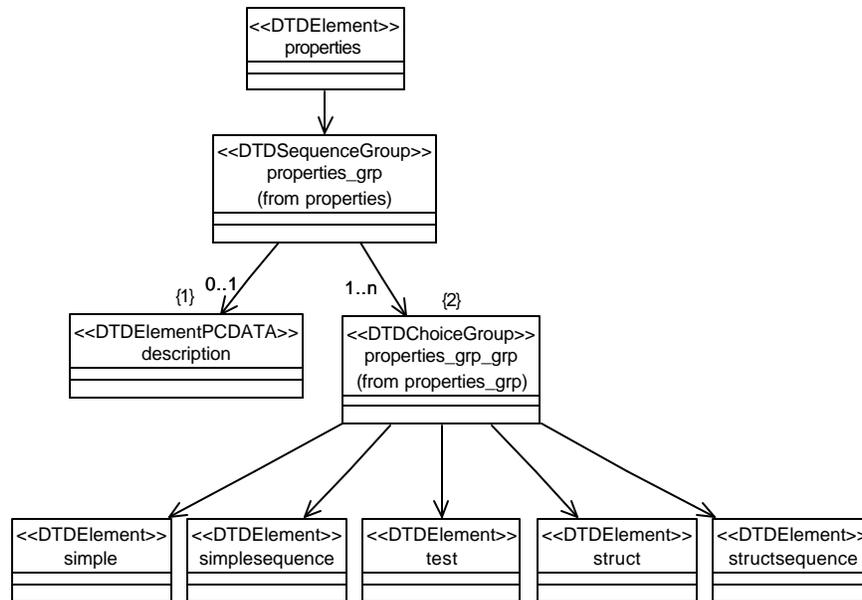


Figure 4.2-1 properties Element Relationships

A properties file should be provided as part of the software documentation for an application or device implementation. The properties file should not exclude any element.

The description element of the properties file is optional but should be provided. The description element can be used to provide text information about how the properties file is used and what is meant by each of the properties.

The properties file should include a *properties* element for each attribute used in the `configure()` and `query()` operations for SCA CF *Resource* components, for each attribute used for dependency checking, and for each attribute used in the CF *TestableObject* `runTest()` operation to configure tests and provide test results.

4.3 Software Component Descriptor

The Software Component Descriptor (SCD) describes a component with respect to the interfaces that it inherits from, the interfaces the component supports, and lists its provides and uses ports. The supported interfaces are those distinct interfaces that were inherited by the component's specific interface. The specified port names (uses and provides) can be used in the Software Assembly Descriptor to connect the component ports to other components.

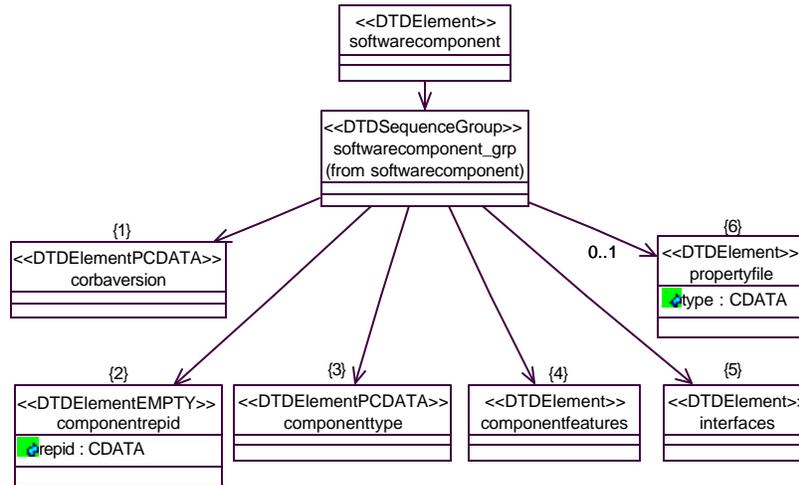


Figure 4.3-1 softwarecomponent Element Relationships

An SCD should be provided as part of the software documentation for an application or device implementation. The SCD should contain each of the following:

- ?? *corbaversion* – indicates which version of CORBA the component is developed for
- ?? *componentrepid* – the repository id of the component
- ?? *componenttype* – identifies the type of software component object
{ *resource / resourcefactory / device* }
- ?? *componentfeatures* – describes each supported message port for the component
- ?? *interface* – describes the component unique id and name for each supported interface

4.4 Software Assembly Descriptor

This section describes the XML elements of the Software Assembly Descriptor (SAD) XML file. The *softwareassembly* element is the root element of the software assembly descriptor file. The SAD is based on the CORBA Components Specification Component Assembly Descriptor. The intent of the software assembly is to provide the means of describing the assembled functional application and the interconnection characteristics of the SCA components within that application. The component assembly provides four basic types of application information for Domain Management. The first is partitioning information that indicates special requirements for collocation of components, the second is the assembly controller for the software assembly, the third is connection information for the various components that make up the application assembly, and the fourth is the visible ports for the application assembly.

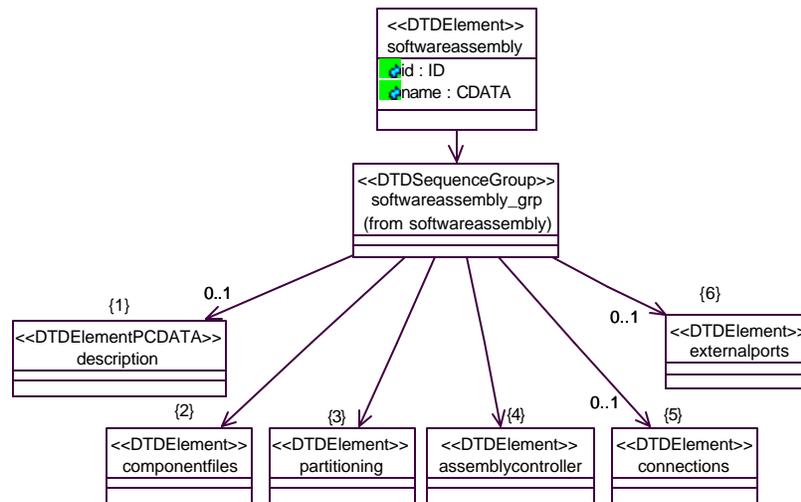


Figure 4.4-1 softwareassembly Element Relationships

An SAD should be provided as part of the software documentation for an application implementation. The SAD should contain all of the mandatory XML elements as well as most of the optional elements. The description element of the SAD is optional but should be provided, because it can be used to provide text information about the application implementation. The connections element is not required, because some application may not have any connections; however, there must be a connections element if the CF ApplicationFactory implementation needs to connect components together. Likewise, the externalports element identifies any port which might be the subject of a getPorts query from outside the application (from a GUI, for example); thus there should be an externalports element for any port which might be requested by external entity.

Installing an application into the system consists of installing a SAD file. The SAD file references component's SPD files to obtain deployment information for these components. The *softwareassembly* element's id attribute is a DCE UUID, as specified in section 4.1, which uniquely identifies the assembly. The *softwareassembly* element's name attribute is the user-friendly name for the CF ApplicationFactory name attribute.

4.5 Profile Descriptor

The *profile* element can be used to specify the absolute profile file pathname relative to a mounted CF *FileSystem*. The filename attribute is the absolute pathname relative to a mounted *FileSystem*. This filename can also be used to access any other local file elements in the profile. The type attribute indicates the type of profile being referenced. The valid type attribute values are "SAD", "SPD", "DCD", and "DMD". This element can be given out for any CF interface (e.g., CF *Application*, CF *Device*, CF *ApplicationFactory*, CF *DeviceManager*, CF *DomainManager*) that has profile attributes.

The format described by the profile descriptor XML can be used in the implementation of certain SCA interface attributes. The attributes that utilize the format dictated by the profile descriptor XML are the Application's softwareProfile, the DomainManager's domainManagerProfile, the DeviceManager's deviceConfigurationProfile, and the Device's softwareProfile attribute. The profile descriptor XML is not used in any delivered XML file.

5 Design Progression

An interface defines the connection between a waveform component and another waveform component or between a waveform component and an entity¹⁵ outside the waveform. CORBA prescribes an Interface Definition Language (IDL) to define interfaces. Servant code provides the actual implementation for an interface. Numerous design methodologies are available to develop an implementation based upon IDL. This section discusses object-oriented procedures (using a software modeling tool). Following these procedures will enable the developer to create implementation servant code when IDL provides the initial definition.

5.1 IDL Modeling

CF interfaces are expressed in CORBA IDL. The SCA IDL has been generated directly by the Rational Rose UML software-modeling tool. This “forward engineering” approach ensures that the IDL accurately reflects the architecture definition as contained in the UML models. Any IDL compiler for the target language of choice may compile the generated IDL.

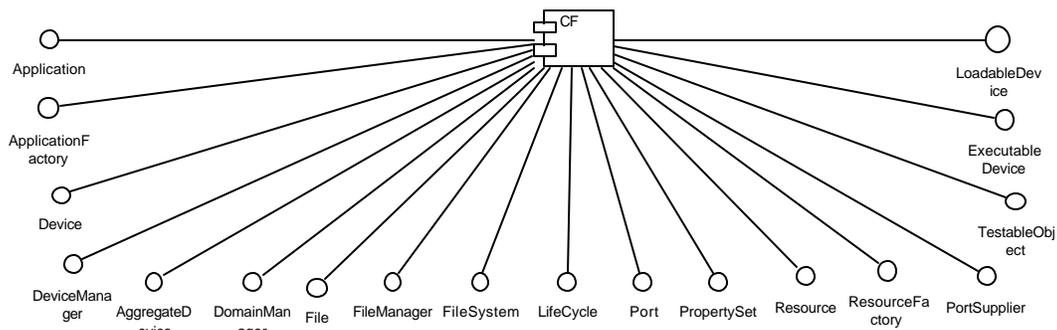


Figure 5.1-1 CF CORBA Module

Forward Engineering is the process of transforming a model into code through a mapping to an implementation language. To forward engineer a class diagram,

- ?? Identify the mapping rules from UML to your implementation language(s) of choice
- ?? Depending on the semantics of the languages chosen, you may have to constrain the use of certain UML features. For example, the UML permits modeling of multiple inheritance, but Smalltalk (for example) permits only single inheritance.
- ?? Use tagged values to specify your target language. A tagged value extends the properties of a UML building block, allowing the creation of new information in that element's specification. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages. Collaborations represent the implementation of patterns that make up a system.

¹⁵ for example, the UserInterface

5.2 IDL Generation

At any point in the IDL design, the tool chosen to develop the UML model can be told to generate IDL. The following is a code snippet of the CF IDL generated from such a tool.

```
//Source file: CF.idl

#ifndef __CF_DEFINED
#define __CF_DEFINED

/* CmIdentification
   %X% %Q% %Z% %W% */

/* This package provides the main framework for all objects within the
radio. */

module CF
{
  interface File;
  interface Resource;
  interface Application;
  interface Device;
  interface ApplicationFactory;
  interface DeviceManager;

  /* This type is a CORBA IDL struct type which can be used to hold
any CORBA basic type or static IDL type. */

  struct DataType
  {
    /* The id attribute indicates the kind of value and type
(e.g.,frequency, preset, etc.). The id can be an UUID string,
an integer string, or a name identifier. */
    string id;

    /* The value attribute can be any static IDL type or CORBA basic
type. */
    any value;
  };

  /* This exception indicates an invalid component profile error. */
  exception InvalidProfile
  {
  };

  /* The Properties is a CORBA IDL unbounded sequence of CF
DataType(s), which can be used in defining a sequence of name and
value pairs. */
  typedef sequence <DataType> Properties;

  /* This exception indicates an invalid CORBA reference error*/
  exception InvalidReference
  {
    string msg;
  };
};
```

```

/* This type is a CORBA unbounded sequence of octets. */
typedef sequence <octet> OctetSequence;

/* This type defines a sequence of strings */
typedef sequence <string> StringSequence;

/* This exception indicates a set of properties unknown by the
component. */
exception UnknownProperties
{
    Properties invalidProperties;
};

```

5.3 IDL Compilation

IDL must be translated into a high level language to be used in a development process. This translation is accomplished with an IDL compiler. When invoked, these tools produce C++ (or Java or Ada, etc.) body (.cpp) & specification (.h) files to be used for both the client-side (stub) and server-side (skeleton) operation.

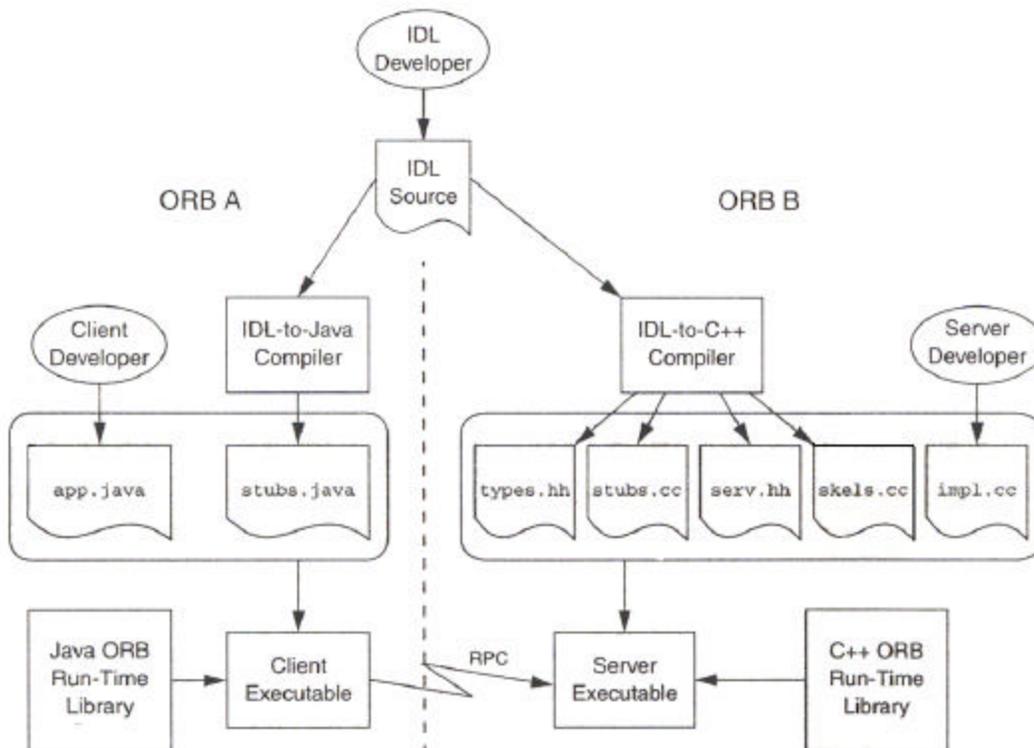


Figure 5.3-1 Development Process for Different Development Environments

The client code is specialized to perform client-oriented operations using a particular language and ORB. The server code is specialized to perform server-oriented operations using a particular language and ORB. Thus, the client code and the server code will tend to differ from each other and from the IDL. Each may have unique pointer types, specialty functions, and class name variations.

5.4 Client/Server Compilation

The IDL compilation outputs, at a minimum, a client-side (stub)code and server-side (skeleton) code. Some IDL compilers also generate implementation templates for each of the interfaces found in the IDL. However, for our discussions, we'll assume the IDL compiler generates only a client and server (.cpp & .h). The client and server files must then be compiled to ensure the code is syntactically correct.

The server code does not perform the tasks promised by the interface. Instead, it calls servant objects, written by the implementer, to perform the tasks promised by the interface. The servant objects could be based on implementation templates generated by the IDL compiler, or they could be designed using standard object-oriented design methods. If the latter procedure is followed (using a software modeling tool), the work of the implementer is made more efficient and more accurate if the tool contains a server-side view of the interface based upon the actual server code.

The client code does not originate requests on the interface. Instead, tasks promised by the interface are requested by a using object, and the client code relays these requests to the server code. If the person who programs the using object designs it using a software modeling tool, this work is made more efficient and more accurate if the tool contains a client-side view of the interface based upon the actual client code.

Reverse engineering of the client/server files is one means of automatically generating models of the server-side view and the client-side view of the interface, thereby providing the reduction in development effort described in the previous two paragraphs.

5.5 Reverse Engineering

Reverse Engineering is the process of transforming code into a model through a mapping from a specific implementation language. Reverse engineering results in a flood of information, some of which is at a lower level of detail than needed to build useful models. From source code, one can reverse engineer back to classes – this is what is most commonly done. Reverse engineering the client and server files at this point has several advantages: 1) the implementation classes can be kept updated as the design changes, and 2) The modeling tool typically provides features to generate source code from the model. The auto-generation of source code can often provide significant reductions in the development phase. To reverse engineer the client/server files,

- ?? Identify the rules for mapping from the implementation language(s) of choice
- ?? Point to the desired code to be reverse engineered, and generate a new model or modify an existing one that was previously forward engineered.
- ?? Create a class diagram by querying the model. For example, one might start with one or more classes then expand the diagram by following specific relationships or other neighboring classes.

5.6 Creation of Servant Implementation Classes

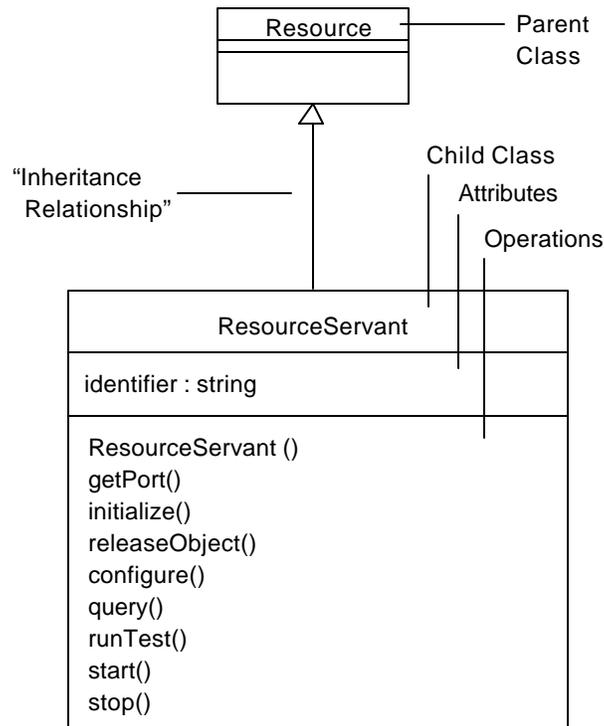


Figure 5.6-1 ResourceServant Inheritance

Assuming the IDL-generated client and servant files were reverse-engineered into the model tool, class diagrams can then be created that depict the various implementation classes. Figure 5.6-1 shows the **ResourceServant** class implementing the CF **Resource** interface. The **Resource** interface is the server-side skeleton class generated by the IDL compiler. The **ResourceServant** class shown describes how the developer plans to implement the **Resource** interface. In this case, the **ResourceServant** class has one attribute and eight operations (not including the constructor). Once the servant design is entered into the model, the tool can be instructed to generate source code. The term “inherit” simply means that attributes at a higher class-level are common with all the subclasses. The “inherit” feature is shown by a hollow arrow; the UML symbol for “generalization”.

5.7 Servant Code Generation

Once the servant design (e.g. ResourceServant in Figure 5.6-1) is entered into the model, the tool can be instructed to generate source code. This provides a template with which to begin the coding stage. Typically, the generated code is placed in preservation sections (provided by the tool) that ensure that any subsequent model updates, and therefore code generations, do not overwrite existing software (these preservation sections are fairly common across code generation tools). This allows the model and implementation to be kept in sync. The following is a code snippet of the ResourceServant.h file. Notice the inheritance of the reverse-engineered class (in red - `class ResourceServant : public POA_CF::Resource`). Also contained in the .h file is the prototype for all the operations shown in Figure 5.6-1.

```

### begin module%37397F0C6910.cm preserve=no
//      %X% %Q% %Z% %W%
### end module%37397F0C6910.cm

#ifndef ResourceServant_h
#define ResourceServant_h 1

### begin module%37397F0C6910.additionalIncludes preserve=no
### end module%37397F0C6910.additionalIncludes

### begin module%37397F0C6910.includes preserve=yes
### end module%37397F0C6910.includes

// ResourceHelper
#include "ResourceHelper.h"
// LoggerPortServant
#include "LoggerPortServant.h"
// CFServer
#include "CFServer.h"

### begin module%37397F0C6910.additionalDeclarations preserve=yes
#include "ConnectionServant.h"
### end module%37397F0C6910.additionalDeclarations

### begin ResourceServant%37397F0C6910.preface preserve=yes
### end ResourceServant%37397F0C6910.preface

### Class: ResourceServant%37397F0C6910
//      This class implements the CF::Resource interface and provides common
//      attributes and operations for a resource implementations
//
### Category: Core CSCI Design Components::Abstract Channel Design
Components%3520FCED8D78; Global
### Subsystem: Core CSCI Implementation Components::Abstract Channel Implementation
Components%3563352AFCB0
### Persistence: Transient
### Cardinality/Multiplicity: n

### Uses: <unnamed>%37397F0CCD10;NameUtilities { -> }
### Uses: <unnamed>%387C7D167000; { -> }

```

```

class ResourceServant : public POA_CF::Resource /// Inherits: <unnamed>%3A9E60D9E1A8
{
    /// begin ResourceServant%37397F0C6910.initialDeclarations preserve=yes
    /// end ResourceServant%37397F0C6910.initialDeclarations

public:
    /// Constructors (specified)
    /// Operation: ResourceServant%947682840
    ResourceServant (PortableServer::POA_ptr oe_poa, const char *name);

    /// Destructor (generated)
    virtual ~ResourceServant();

    /// Other Operations (specified)
    /// Operation: initialize%955649192; C++
    virtual void initialize (CORBA::Environment& _env) = 0;

    /// Operation: releaseObject%955649193; C++
    void releaseObject (CORBA::Environment& _env);

    /// Operation: query%955649194; C++
    virtual void query (CF::Properties& configProperties, CORBA::Environment& _env) = 0;

    /// Operation: configure%955649195; C++
    virtual void configure (const CF::Properties& configProperties, CORBA::Environment& _env);

    /// Operation: runTest%955649196; C++
    virtual CORBA::Long runTest (CORBA::ULong testNum, CORBA::Environment& _env) = 0;

    /// Operation: start%955649199; C++
    virtual void start (CORBA::Environment& _env) = 0;

    /// Operation: stop%955649200; C++
    virtual void stop (CORBA::Environment& _env) = 0;

    /// Operation: getPort%959702974; C++
    CORBA::Object* getPort (const CORBA::Char* name, CORBA::Environment& _env);

    // Additional Public Declarations
    /// begin ResourceServant%37397F0C6910.public preserve=yes
    /// end ResourceServant%37397F0C6910.public
};

/// begin ResourceServant%37397F0C6910.postscript preserve=yes
/// end ResourceServant%37397F0C6910.postscript

// Class ResourceServant

/// begin module%37397F0C6910.epilog preserve=yes
/// end module%37397F0C6910.epilog

#endif

```

5.8 XML Generation

As was discussed in section 1.2, the SCA-defined DTDs define exactly what is allowed to appear inside an XML document. Since those DTDs have already been created/validated (see Appendix D of the SCA), the developer only needs to be concerned with the XML needed to define the application. Appendix A provides guidance to those new to XML. Appendix D of this document contains the XML describing the fictitious XYZ Application discussed in Section 6. Appendix E contains the XML describing the fictitious XYZ device discussed in Section 7. Domain Profile files need to follow the format of the Document Type Definitions (DTDs) provided in Appendix D of the SCA. DTD files are installed in the domain and need to have “.dtd” as their filename extension. There are a number of commercial XML authoring tools that help the developer concentrate on the content of the XML, and not on the syntax and formatting issues.

6 Waveform Development

Implementing any SCA-compliant waveform software follows certain steps. As is indicated by notations in the following list, some steps are discussed elsewhere in this guide, but the entire procedure is outlined here as a checklist to assist the developer to ensure that no step is omitted.

1. Identify functionality to be provided by the waveform software { *section 6.1.2* }
2. Determine which API Service Groups are needed { *section 6.1.3* }
3. Determine what services are needed beyond the API Service Groups { *section 6.1.4* }
4. Build UML model of interface { *section 6.2* }
5. Generate IDL from UML model of interface { *section 5.2* }
6. Translate IDL into language-appropriate implementation files { *section 5.3* }
7. Compile code generated in step 6 { *section 5.4* }
8. Reverse engineer UML model from language-specific implementation files { *section 5.5* } (*optional*)
9. Build UML model of waveform software { *section 6.3* }
10. Generate language-appropriate template files for servant and user software { *section 5.7* }
11. Write servant and user software
12. Write XML for each component { *section 5.8* }
13. Build User Interface { *section 8* } (*optional*)
14. Integrate software and hardware
15. Test resultant application

6.1 Functional Allocation to API Design

The sections 6.1 - 6.3 discuss steps 1-4 and 9 in detail. For each step, a short discussion of the process is followed by an example. This continuing example uses an imaginary waveform, called XYZ, to illustrate the steps involved in implementing any waveform. The example steps assume that no existing API meets the needs of waveform XYZ, so all new APIs are developed.

6.1.1 Introduction

Section 3 of this guide describes the layered structure inherent in a design based upon the API Supplement. Various organizations of components are possible within this general structure. A typical waveform implementation might consist of the components shown in Figure 6.1-1:

- ?? **Modem** receives/transmits the actual signal.
- ?? **Waveform** provides main implementation of functionality based upon the SCA and API. It has ports (usually one in each direction) to handle data flow with the application or device that is connected to it. It must provide CF::Resource functionality (see Section 3), so a CF::Resource port is also shown. The Core Framework's ApplicationFactory implementation is the only software that uses CF::Port, and it already has access to the waveform application (obtained when it created the waveform), so CF::Port is not shown on diagrams in this guide.
- ?? **UI** represents interface to user. This topic is discussed in general terms in sections 8.1 - 8.3, but often this is a specialty of its own.

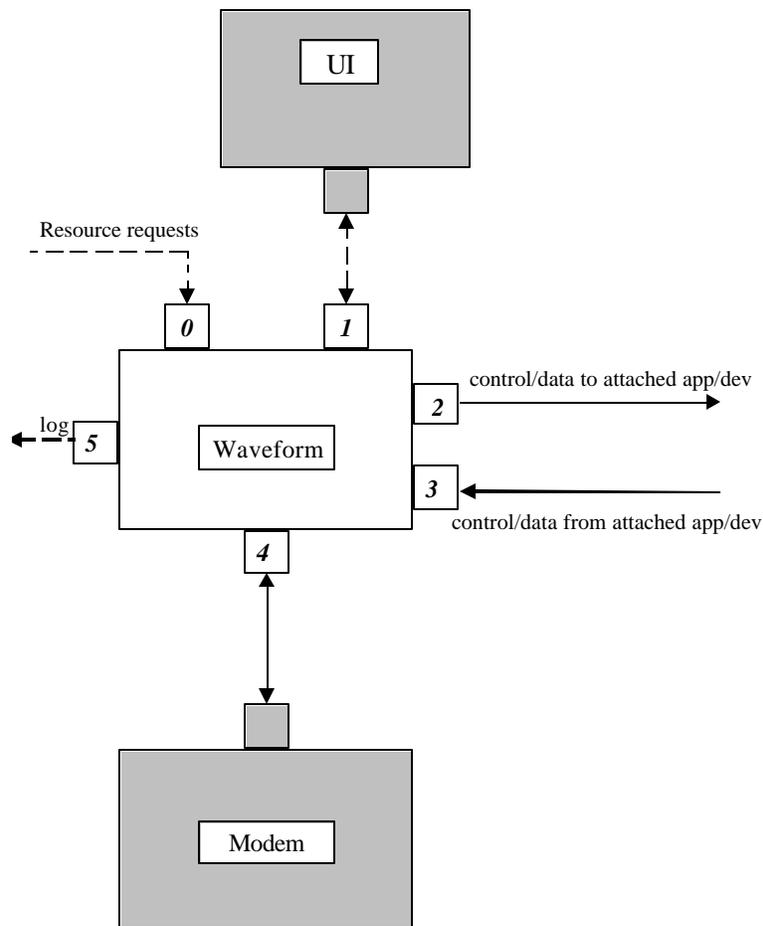


Figure 6.1-1 Typical Waveform Implementation

Since this document relates to the work of implementing a waveform application based upon the SCA and API, it focuses upon the **Waveform** software; the other components are shaded in Figure 6.1-1.

The ports are numbered 0...5 in Figure 6.1-1 to provide a convenient means of identifying a particular port during discussions later in this guide.

The actual type of connection between the waveform software and the modem depends upon the actual hardware used. Since this guide does not focus upon that connection, for reasons of simplicity it is shown as a port (in particular, as port 4) in Figure 6.1-1.

A waveform application can be connected either to another application or to a device - the design of the software is not affected by this choice. However, in order to simplify terminology, this guide assumes that the waveform application is attached to a device.

6.1.2 Identifying Application Functionality

The waveform software must implement the **CF::Resource** and **CF::Port** interfaces. Additional functionality required of the waveform software depends upon the messages that it exchanges with other applications.

6.1.2.1 Data and Control from Attached Device

For the purposes of this guide, we assume that the waveform application receives the following messages from the device that is attached to it:

- a) Prepare to receive data (e.g., audio unit would send this for PTT {**PushToTalk**}).
- b) information carried by the signal¹⁶

6.1.2.2 Data and Control to Attached Device

Similarly, the waveform application sends the following messages to the device that is attached to it:

- a) prepare to receive data (normally this means "squelch has been broken")
- b) information carried by the signal¹⁶

6.1.2.3 Control from User Interface (UI)

We assume that the waveform might receive the following commands from the UI:

- a) set levels of AGC and noise squelch used to recognize the presence of a signal
- b) set frequency to be used for receiving and sending
- c) disable transmission (regardless of PTT activity)
- d) set mode of operation (VOICE or DATA)
- e) set speed of transmission when in DATA mode of operation
- f) set power level of transmitter

¹⁶ Transmission between digital radio components may be either digital data or analog ("voice") data that has been converted into a digital form. In either case, the bits are packaged into a packet structure. This information stream consists of a series of these packets.

6.1.3 Functional API Mapping

Using Sections 3.1 through 3.4, an appropriate API primitive service should be identified, if possible, for each API requirement listed in section 6.1.2. Identifying needed primitive services (operations) automatically identifies the enclosing Service Groups as the classes needed for the waveform interface.

Table 6.1-1 shows a mapping for the XYZ waveform described in sections 6.1.2.1 - 6.1.2.3. This table shows, for each requirement identified in sections 6.1.2.1-6.1.2.3, which service (contained within which Service Group) satisfies that requirement. Each waveform variable is implemented as a class attribute. Thus, Table 6.1-1 shows these values being set from the UI by using the CF::Resource::configure operation even though they could be set using various methods provided by the Physical API.

needed functionality {description}	waveform port number in Figure 6.1-1	API	Service Group / Class	primitive service / operation	note
6.1.2.1-a {prepare to receive data}	3				
6.1.2.1-b {data from app}	3	Physical RT	SimplePacket via TransmitPacket	pushPacket	instantiated with PayloadType as octet or other appropriate type
6.1.2.2-a {prepare to receive data}	2				
6.1.2.2-b {data to app}	2	Physical RT	SimplePacket via TransmitPacket ¹⁷	pushPacket	instantiated with PayloadType as octet or other appropriate type
6.1.2.3-a {squellch params}	1			<i>configure</i>	
6.1.2.3-b {frequency}	1			<i>configure</i>	
6.1.2.3-c {disable xmit}	1	Physical NRT	Transmit_Inhibit	inhibitTransmit	boolean parameter enables this operation to turn transmission on or off as needed
6.1.2.3-d {voice/data mode}	1				
6.1.2.3-e {set data speed}	1			<i>configure</i>	
6.1.2.3-f {set xmit power}	1			<i>configure</i>	

Table 6.1-1 Primitive Service for Each Requirement

¹⁷ The definitions (in the API Supplement) of TransmitPacket and ReceivePacket are identical except for direction, and the API Supplement provides no further direct guidance. The API Supplement seems to assume that data normally differs between the two directions (for example, received data might include QOS information, but transmitted data never would). For the simple XYZ example used in this guide, there is no difference in data between the two directions, so we derive just one concrete UML class to handle data in both directions and label it "TransmitPacket" ("ReceivePacket" would have been equally valid as a label for this class).

The Service Group "TransmitPacket" could be instantiated in both the Physical layer and the MAC layer. In order to emphasize that this particular instantiation is in the Physical layer, it is referred to as "TransmitPacketPhys" for the remainder of this guide.

6.1.4 Mapping Remaining Needs

The APIs were selected to provide commonly needed services. However, a waveform may have needs which were not anticipated in the API design. In that case, the waveform designer may

- ?? extend an existing Service Group
- ?? design an entirely new Service Group

The requirements for the simple XYZ waveform include requirements (e.g., the requirements represented by grayed-out rows in Table 6.1-1) which do not have a good match in the APIs defined in the SCA API Supplement or by a waveform procurement document. In Table 6.1-2, those requirements are met by adding services to the model. In this case, the requirements are met by extending Service Group "TransmitPacketPhys" (thereby creating "TransmitPacketPhys_XYZ") and creating new Service Group "SetMode".

needed functionality {description}	waveform port number in Figure 6.1-1	API	Service Group / Class	primitive service / operation	note
6.1.2.1-a {prepare to receive data}	3	Physical RT	extend TransmitPacketPhys to create TransmitPacketPhys_XYZ	new signalDetected	occurs each time status changes - boolean parameter indicates whether signal is currently present
6.1.2.1-b {data from }	3	Physical RT	SimplePacket via TransmitPacketPhys	pushPacket	instantiated with PayloadType as octet or other appropriate type
6.1.2.2-a {prepare to receive data}	2	Physical RT	extend TransmitPacketPhys to create TransmitPacketPhys_XYZ	new signalDetected	occurs each time status changes - boolean parameter indicates whether signal is currently present
6.1.2.2-b {data to }	2	Physical RT	SimplePacket via TransmitPacketPhys	pushPacket	instantiated with PayloadType as octet or other appropriate type
6.1.2.3-a {squelch params}	1			<i>configure</i>	
6.1.2.3-b {frequency}	1			<i>configure</i>	
6.1.2.3-c {disable xmit}	1	Physical NRT	Transmit_Inhibit	inhibitTransmit	boolean parameter enables this operation to turn transmission on or off as needed
6.1.2.3-d {voice/data mode}	1	Physical NRT	new SetMode	new setMode	parameter from enum { VOICE_MODE, DATA_MODE }
6.1.2.3-e {set data speed}	1			<i>configure</i>	
6.1.2.3-f {set xmit power}	1			<i>configure</i>	

Table 6.1-2 Extending Service Group to Meet All Requirements

6.2 Building API Layer Definitions

6.2.1 Introduction

A purpose in using Service Groups is to reduce analysis/design effort, since a basic Service Group is developed just once. Once the appropriate Service Groups have been identified, the next step is to build these Service Groups into a model that can be used to generate appropriate IDL. The API Supplement provides precise procedures for constructing and using APIs. Assuming that the needed API does not exist (according to the section 6.1 introduction to waveform XYZ, this assumption is true of XYZ), new interfaces should be built using one of the methods summarized here:

1. A concrete Service Group is not changed, since it is a usable class already.
2. A generic Service Group must be instantiated to create a usable concrete class. In this situation, instantiation consists of replacing the generic type(s) with type(s) needed for a particular waveform. A type must be defined if it is not already defined.
3. If a Service Group is to be "extended", a new concrete class must inherit from an existing concrete class. The extra attributes and/or operations needed by the waveform are placed into the new class.
4. A completely new class is constructed using usual methodology.

Options 2-4 create new concrete classes, which are used in implementing the waveform. They do *not* reflect back on the API, but they must conform to API Supplement requirements by specifying interfaces, behavior, state information and exceptions.¹⁸

The API Supplement specifies that scoped names should be used to ensure that each name is unique within the system; that is, a separate module is created for each waveform, and that module contains a separate module for each API. Within this structure, additional packaging may be used to group together interfaces that are realized by a given component and tend to be used together.

Once all building blocks are in the model, they may be combined¹⁹ so that the final design involves inheritance from just a small number of classes.

6.2.2 Waveforms with Physical Layer

Table 6.1-2 shows that the simple XYZ waveform requires Service Group *TransmitPacket* from the Physical Real-Time API and Service Groups *SetMode* and *Transmit_Inhibit* from the Physical Non-Real-Time API.

In order to simplify individual diagrams, each instantiated building block is put into a separate diagram, and a special diagram (called "XYZ_Phys") is created to hold integrating classes. The other CORBA modules (CF, PacketAPI, PhysicalNonRealTimeAPI, and PhysicalRealTimeAPI) contain pre-existing components which are being reused. The resulting structure is shown in Figure 6.2-1²⁰.

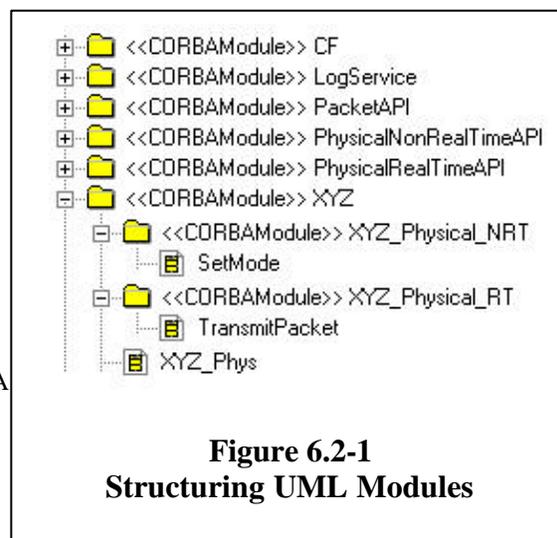


Figure 6.2-1
Structuring UML Modules

¹⁸ See section 1.2.2 of the API Supplement.

¹⁹ For the purposes of this guide, we shall refer to classes created for this purpose as "integrating classes"

²⁰ This example is taken from a Rational Rose implementation - other tool might use different representation, but the essential structure would be the same.

6.2.2.1 Using Concrete Service Group

Of the Service Groups used in the simple XYZ example, only Transmit_Inhibit uses an existing concrete class. Figure 6.2-2 shows the UML for the pre-existing class Transmit_Inhibit.

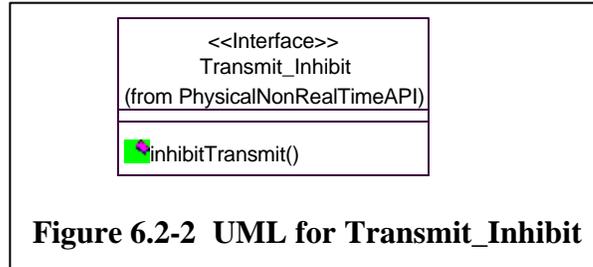


Figure 6.2-2 UML for Transmit_Inhibit

6.2.2.2 Instantiating Generic Building Block

`TransmitPacketPhys` is based upon a template class. Class `SimplePacket` has parameters `Payload_Type` and `Control_Type`, which must be replaced by actual types in order to create a concrete class.

Assuming that data is transmitted octet-by-octet, `Payload_Type` is replaced by `OctetSequence`, which is defined within the module CF. Assuming that no control information accompanies individual messages, an empty structure, `NullControlPhys`, is created to replace `Control_Type`.

`SimplePacket` is instantiated by binding these replacements, resulting in new concrete class `TransmitPacketPhys`.

Figure 6.2-3 uses UML notation to depict the relationships involved in creating class `TransmitPacketPhys`.

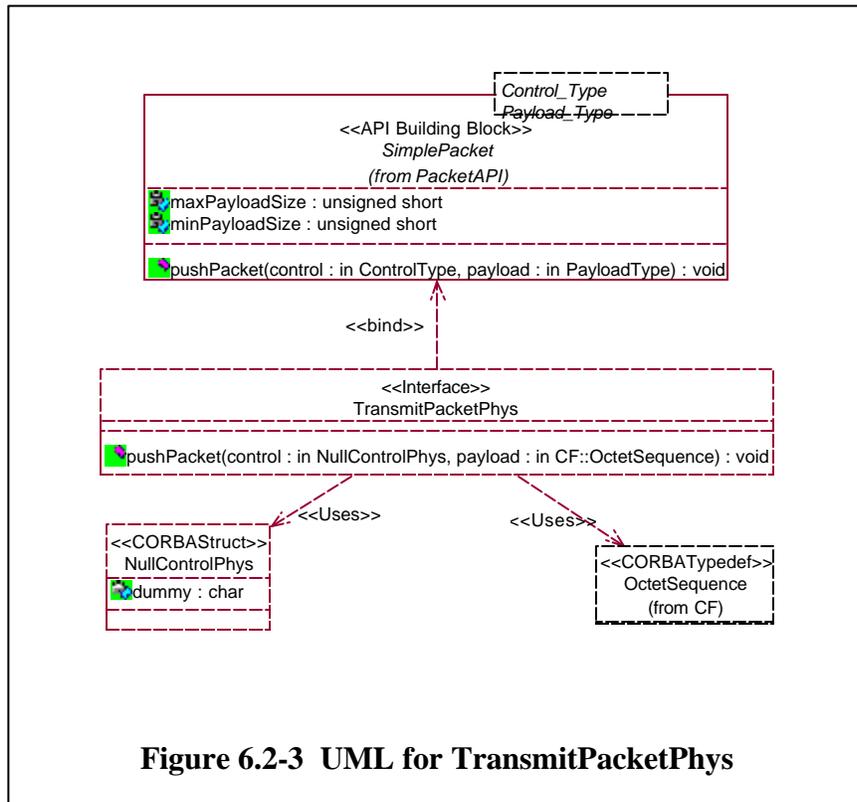


Figure 6.2-3 UML for TransmitPacketPhys

6.2.2.3 Extending A Service Group

Table 6.1-1 shows that instantiating class `TransmitPacket` is insufficient, because requirements 6.1.2.1-a and 6.1.2.2-a remain unsatisfied. Table 6.1-2 provides one solution to this problem - we can create an extended class that includes the operation `signalDetected`.

Figure 6.2-4 uses UML notation to depict the inherited relationships of the extended class `TransmitPacketPhys_XYZ`.

In order to include additional functionality, `TransmitPacketPhys_XYZ` is created inheriting from the concrete class `TransmitPacketPhys`. The newly-created operation `signalDetected` is defined in `TransmitPacketPhys_XYZ`. Since `TransmitPacketPhys_XYZ` inherits from `TransmitPacketPhys`, it includes the `pushPacket` operation.

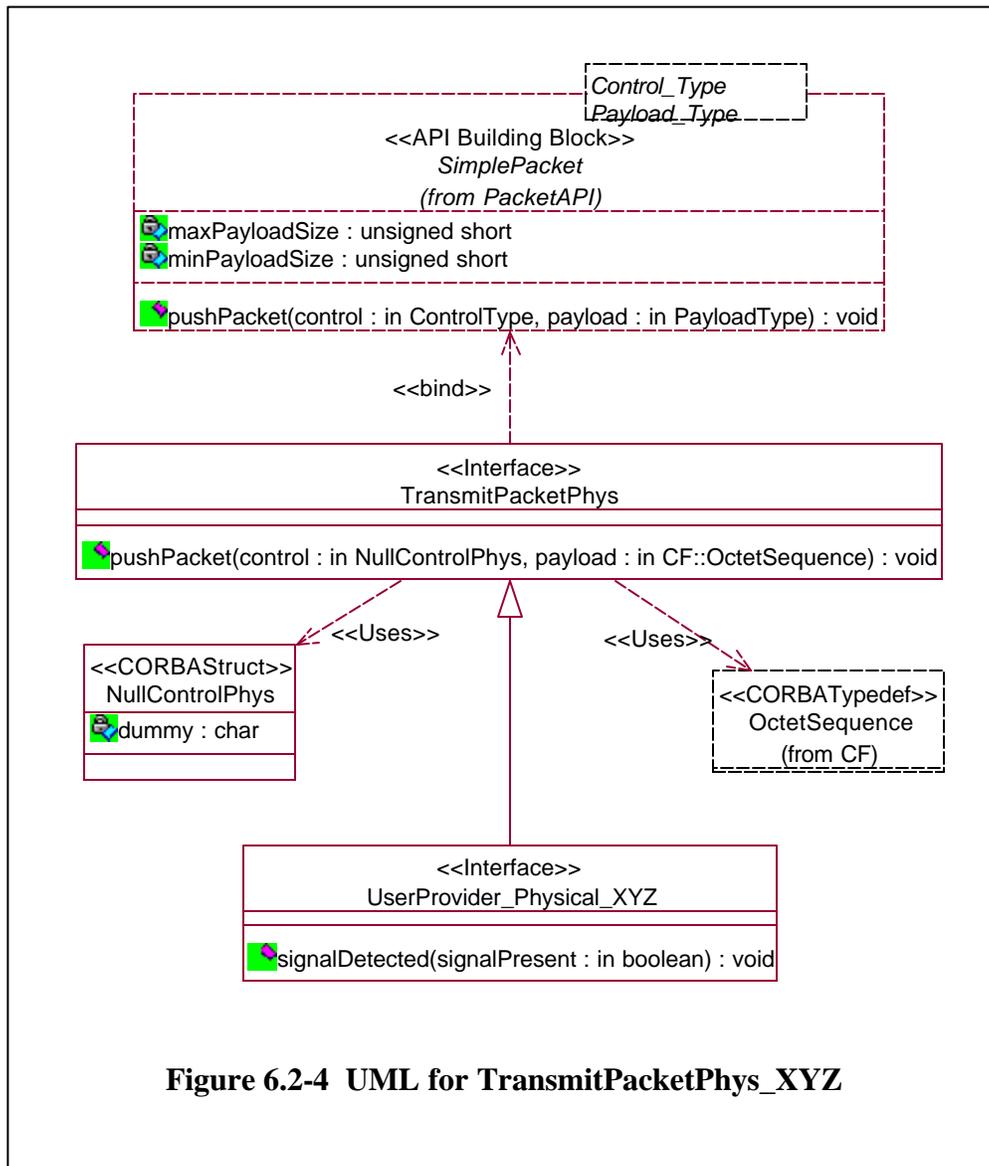
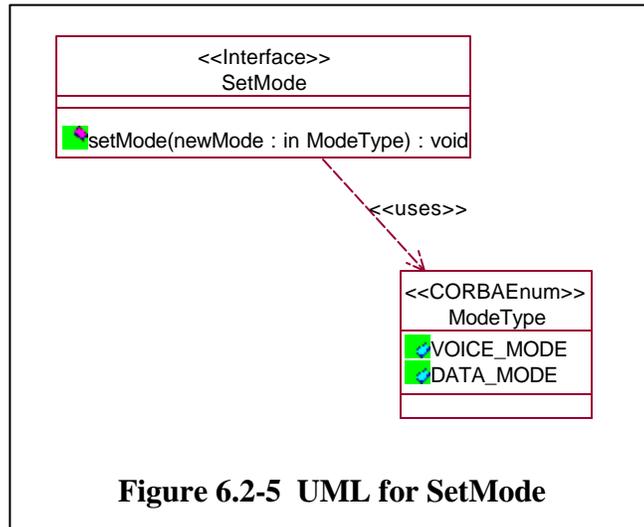


Figure 6.2-4 UML for `TransmitPacketPhys_XYZ`

6.2.2.4 Constructing A New Service Group

Table 6.1-2 shows that a new class, SetMode, is needed for the XYZ waveform. Defining this class is the same as developing IDL for any simple new class.

Figure 6.2-5 shows interface class SetMode using UML notation.



6.2.2.5 Combining Service Groups to Form A New Interface

As is discussed in section 6.3.1.2, normally a uses port is implemented as a pointer of an appropriate type. If a provides port realizes more than one interface, then a uses port can access only the one type associated with the uses port pointer. Thus, the implementation phase of software development is simplified if all interfaces used or provided at a single port are combined into a single ("integrating") interface.

Figure 6.2-6, which is a modification of the relevant portions of Figure 6.1-1, shows which operations connect through each port for the simple XYZ waveform.²¹

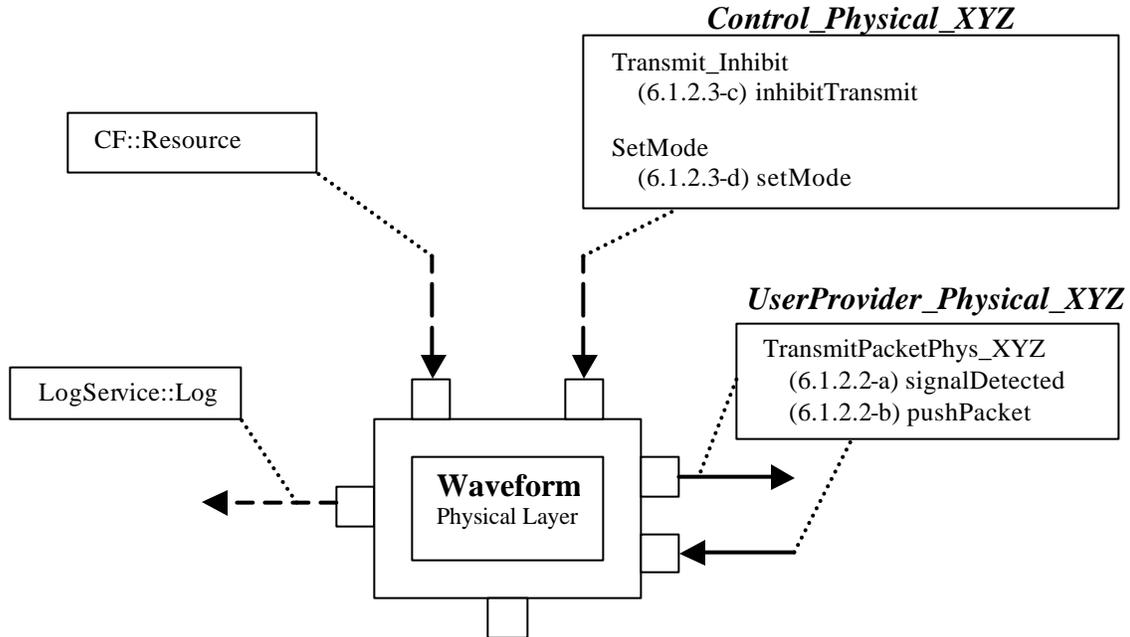


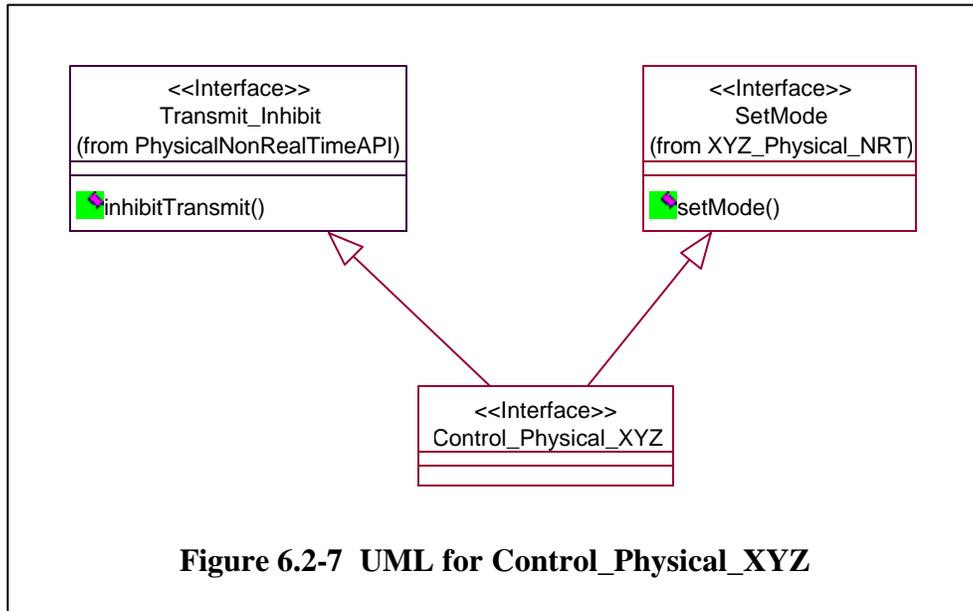
Figure 6.2-6 Assignment of Ports for XYZ Physical Layer

TransmitPacketPhys_XYZ is used by messages moving upstream from this Physical layer to the "attached layer"²². TransmitPacketPhys_XYZ is used also by messages moving downstream from the "attached layer" to this Physical layer. Consequently, TransmitPacketPhys_XYZ is renamed "UserProvider_Physical_XYZ" for the remainder of this guide.

²¹ In this, and future diagrams, each connection to the UI is shown as a single arrow, because a typical system uses return from "downward" call to communicate results "upward" to UI.

²² Depending upon system design, this Physical layer could be attached either to the Physical layer of another application (as is done in this example) or to the MAC layer of the same application (as is done in Section 6.2.3). For the remainder of this guide, we use the term "attached layer" to mean the corresponding layer in another application or higher layer in current application that is connected to the upstream ports of the layer under discussion.

A new class, labeled here as "Control_Physical_XYZ", serves as the interface with the UI. Figure 6.2-7 uses UML to define the Control_Physical_XYZ interface class.



6.2.2.6 Completing the Waveform Interface

The procedures described and demonstrated in sections 6.1.1 through 6.2.2.5 determine which interfaces, including CF::Resource (or, in the case of a device, CF::Device), are needed by the component. Sections 6.3.1 through 6.3.1.2 show how these design decisions are translated into the implementation of ports.

After design of interfaces is complete, IDL can be generated for the new interfaces. Appendix B contains the IDL generated from the design for waveform XYZ (presented in sections 6.2.2 through 6.2.2.5).

6.2.3 Waveforms with MAC Layer

6.2.3.1 Explanation of the Task

XYZ is an example of a waveform that is implemented by using Physical layer functionality only. Now assume that waveform XYZa is created by adding optional error-correcting functionality to XYZ. Error correcting is provided by ChannelErrorControl BB (see Section 3.3), so XYZa needs to include a MAC layer filter to provide the required error-correcting capabilities. Figure 6.2-8 shows how figure 6.1-1 is modified to provide error-correcting functionality.²³

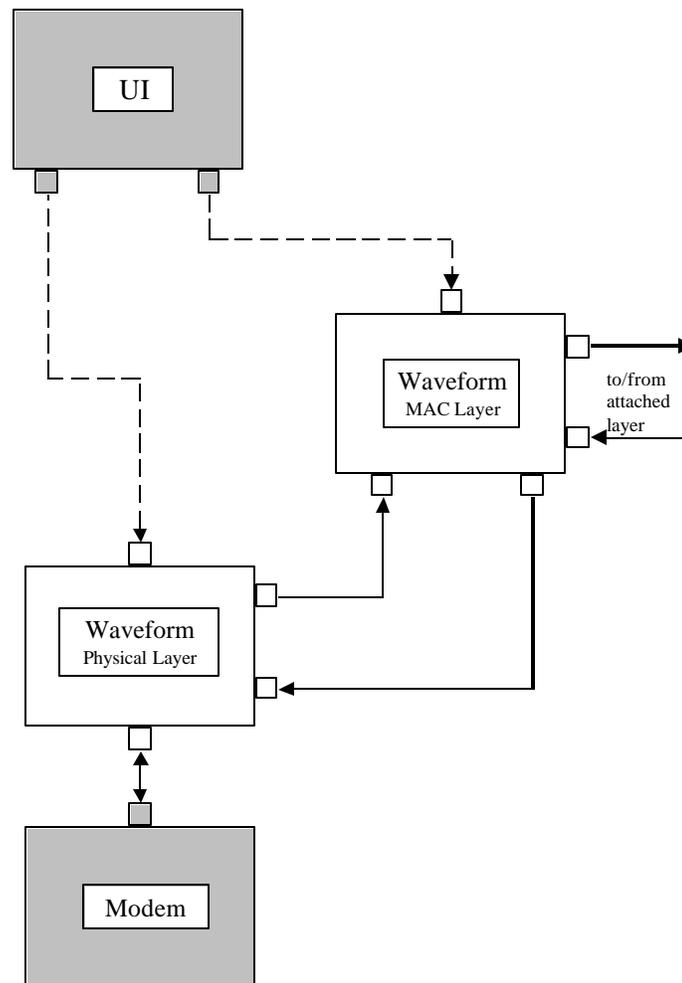


Figure 6.2-8 Two-Layer Waveform Application

6.2.3.2 Solution Using MAC Building Block(s)

Two MAC Layer ports, those that connect to the Physical Layer, require interfaces which were previously defined for the Physical Layer, so only three ports remain to be analyzed.

²³ in order to simplify drawings, resource and log ports are omitted from this, and later, waveform drawings

Packet structure defined by the MAC layer is the same as that defined by the Physical layer. However, the XYZ MAC layer must enable an object to turn error checking ON or OFF, so ChannelErrorControl from the MAC API must be available at both interfaces for ports connecting the XYZ MAC layer to the upstream-attached layer.

If the MAC Layer detects an error that it cannot correct, it must notify the upstream-attached layer. Thus, the upstream uses interface includes an additional operation called errorDetected.

Figure 6.2-9 shows the complete interface classes - named "Provider_MAC_XYZ" and "User_MAC_XYZ") - for ports connecting the XYZ MAC Layer to the upstream attached layer.

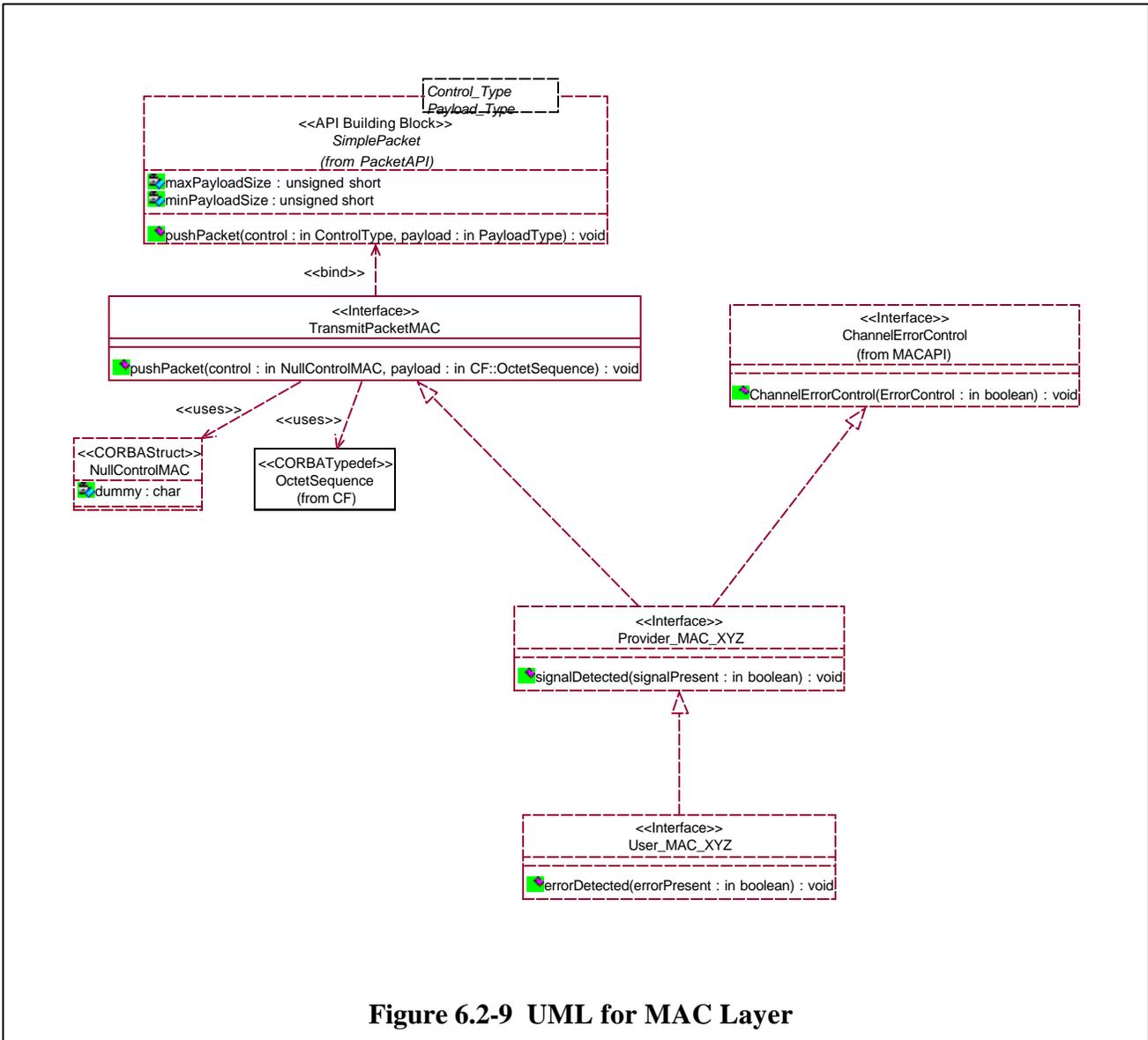


Figure 6.2-9 UML for MAC Layer

Although no attributes are shown in this abbreviated example, we assume that the designer does include attributes corresponding to characteristics of the system. All control of the XYZ MAC layer is accomplished by using the `CF::Resource::configure` operation to update these object attributes, so no further IDL is needed by the XYZ MAC layer.

6.2.4 Waveforms with Link Layer

6.2.4.1 Explanation of the Task

XYZa is an example of a waveform which was implemented by using Physical and MAC layer functionality only. Now assume that waveform XYZb communicates over an unreliable network link, and notifies the sender whenever erroneous data is rejected. Examining Section 3.2 - 3.5 reveals that the most straightforward way to provide error notification capability is to include an LLC layer implementing connectionless service. Figure 6.2-10 shows how figure 6.2-8 is modified to provide this added functionality.

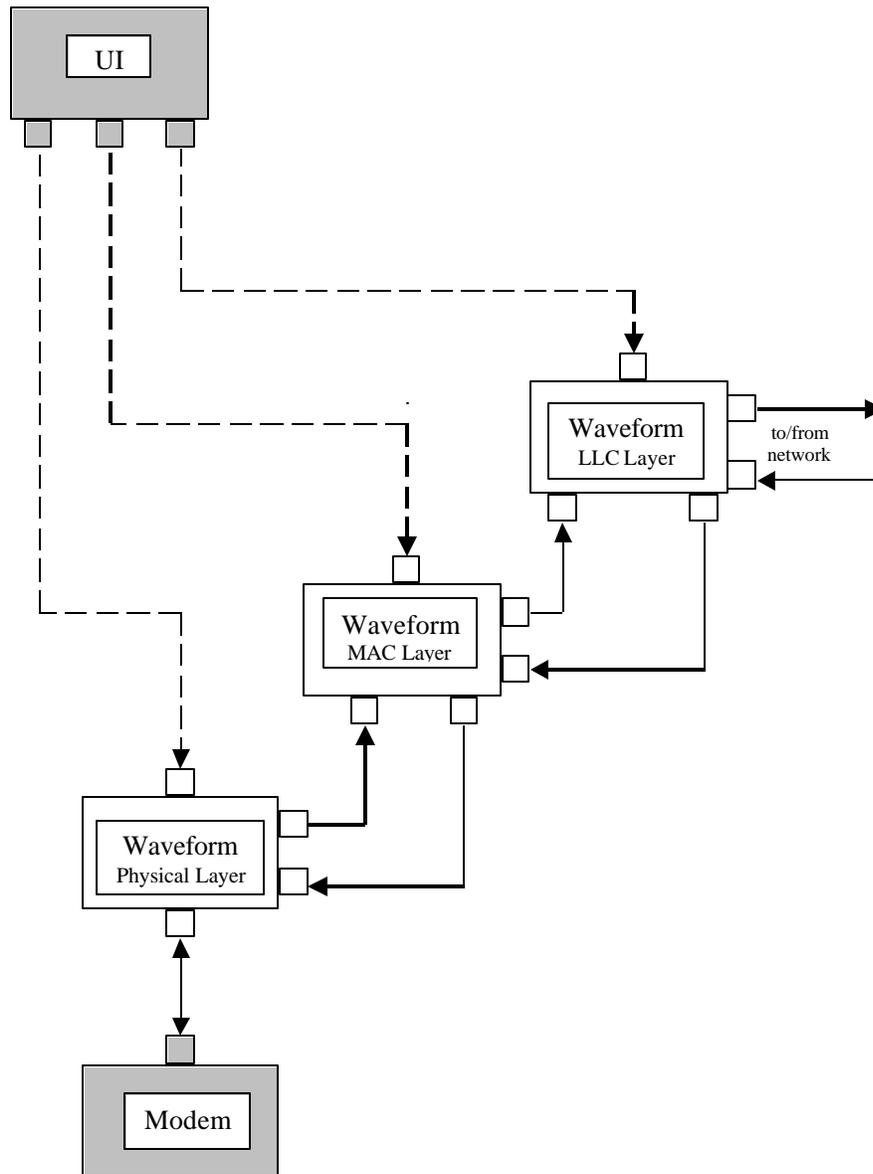


Figure 6.2-10 Three-Layer Waveform Implementation

The LLC API specifies interfaces for the Connectionless mode. Figure 6.2-11 shows structures needed to support these interfaces. The actual interfaces are shown in Figures 6.2-12 (downstream interface - named simply "Provider") and 6.2-13 (upstream interface - named simply "User"), respectively.

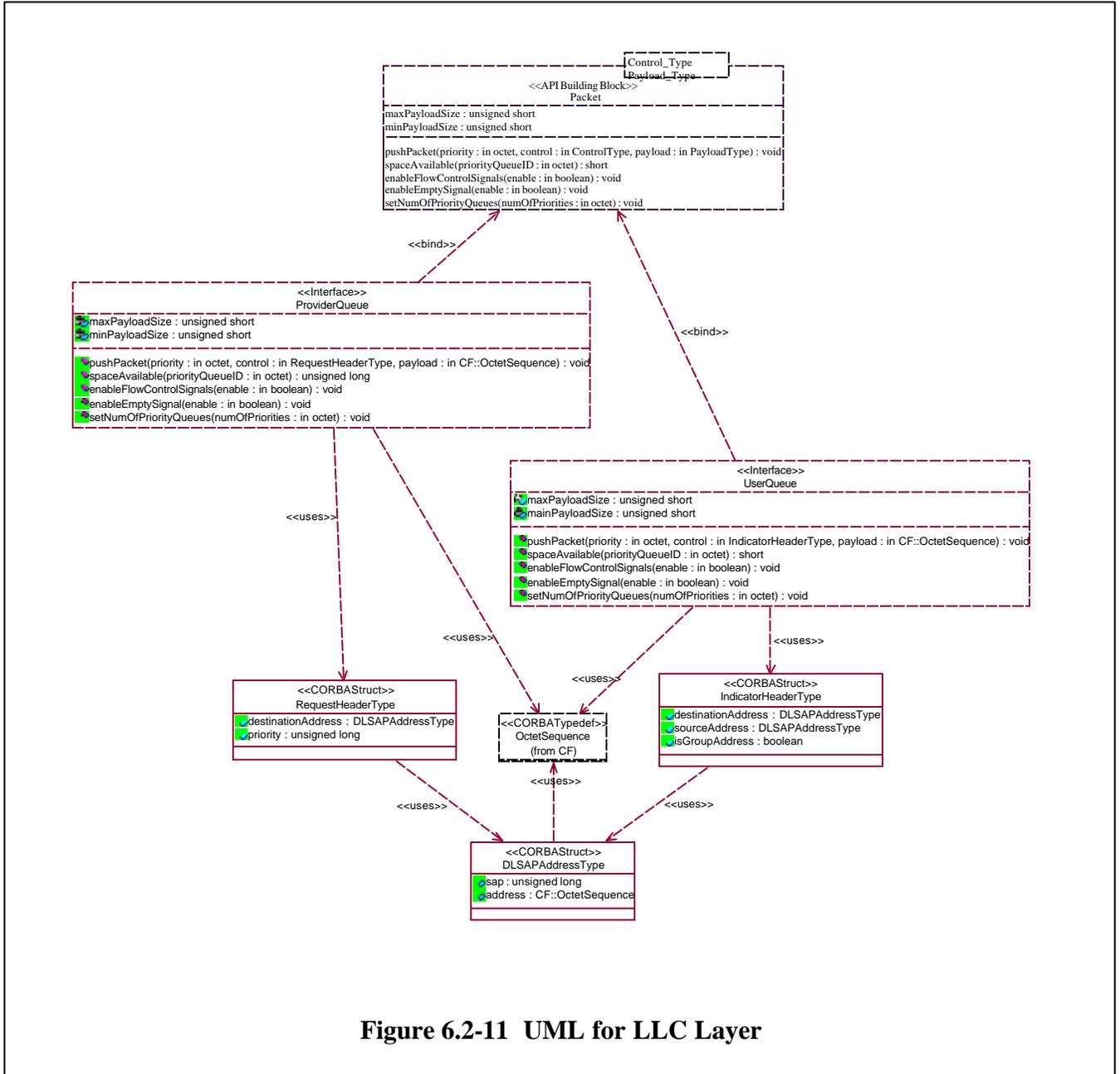


Figure 6.2-11 UML for LLC Layer

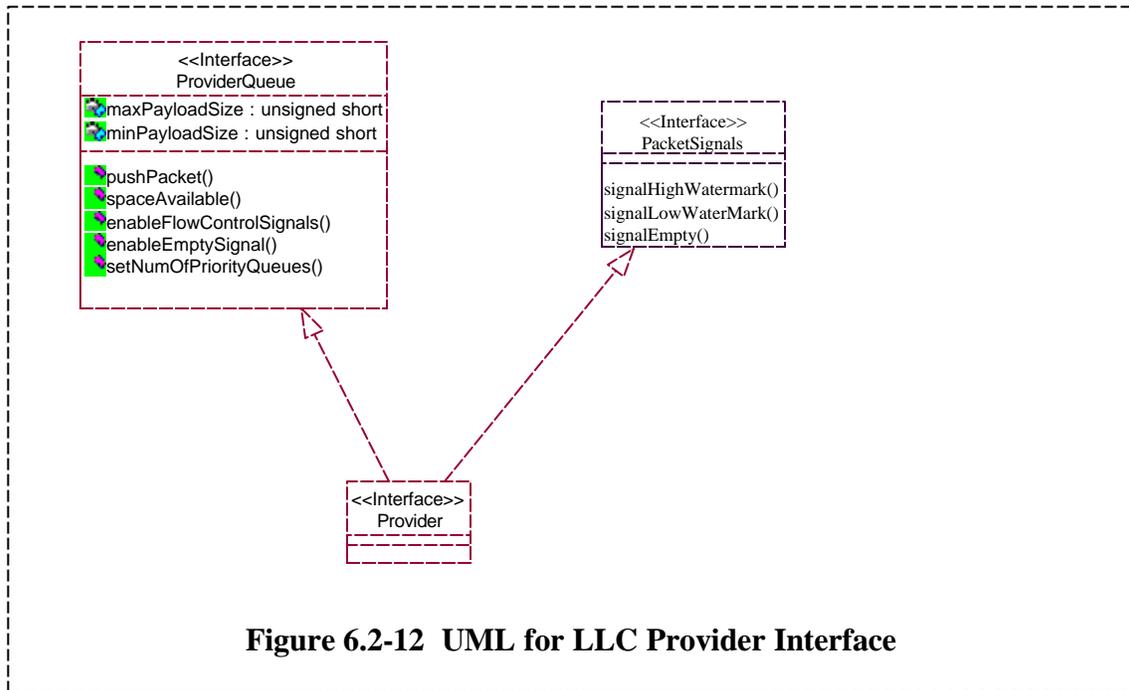


Figure 6.2-12 UML for LLC Provider Interface

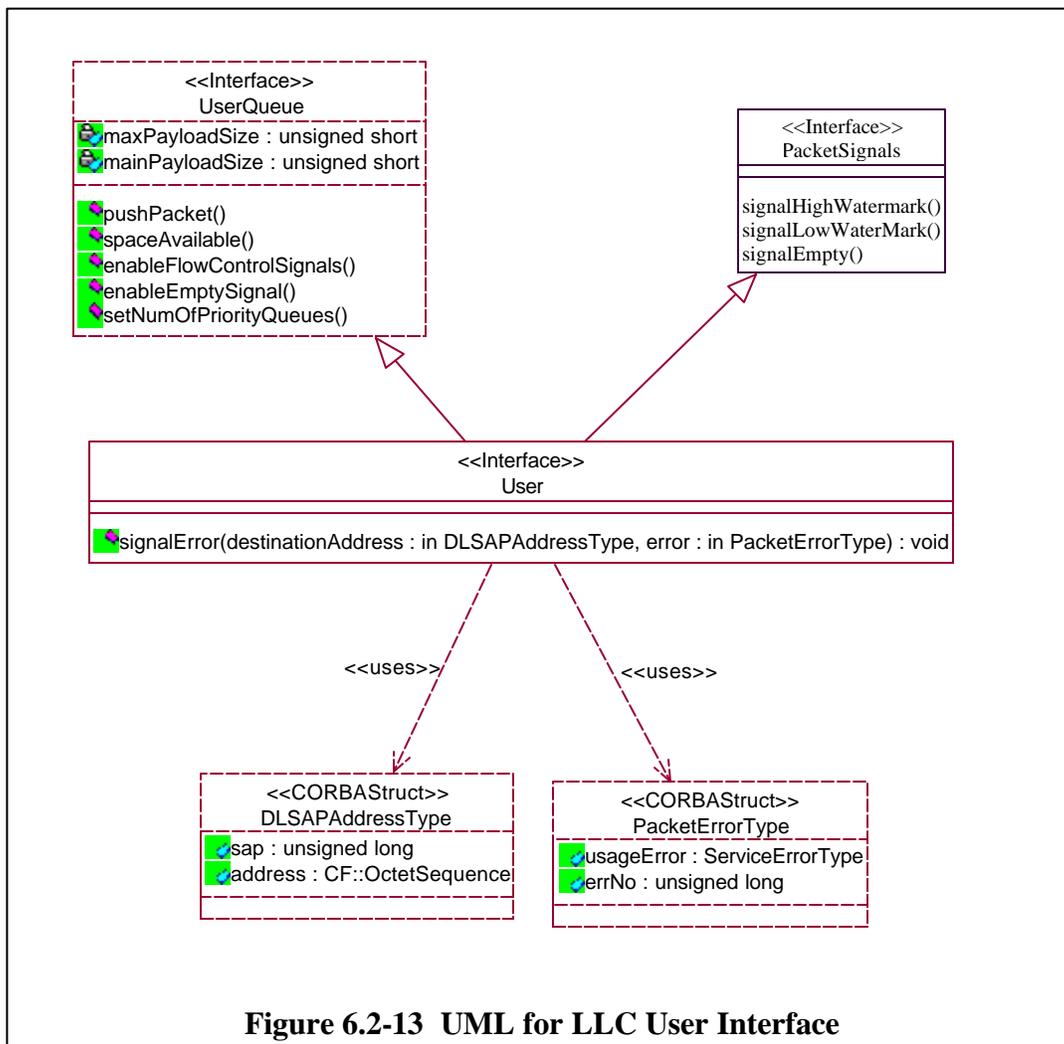


Figure 6.2-13 UML for LLC User Interface

6.2.5 AssemblyController

Waveform XYZb, as developed in sections 6.2.4 - 6.2.4.1, consists of three independent layers. This three-layer design is consistent with API Supplement requirements and with ISO network design methodology. However, the SCA requires that the Application implementation delegate all Resource operations other than getPort to a software component called the "Assembly Controller", which is identified by the *assemblycontroller* element of the Software Assembly Descriptor (SAD) XML file. Thus, an AssemblyController is added to Waveform XYZb, resulting in Waveform XYZc, as shown in Figure 6.2-14.

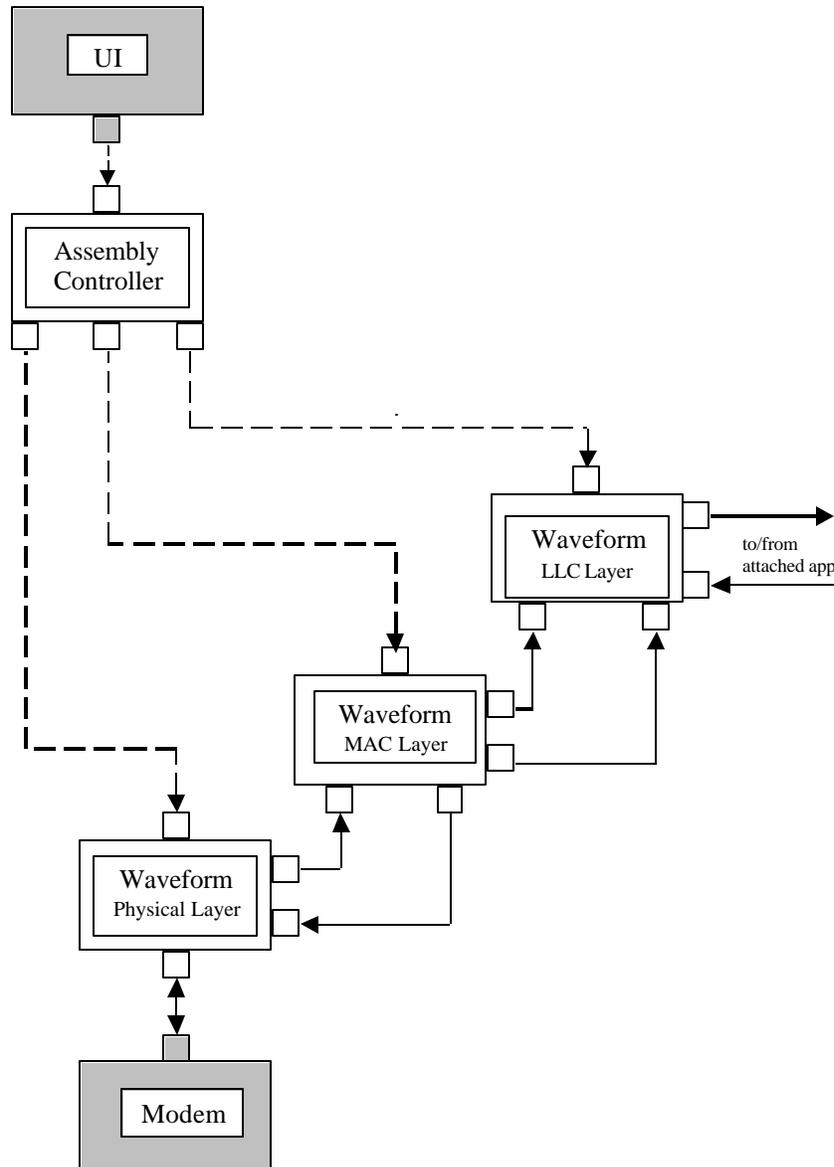


Figure 6.2-14 Waveform with Assembly Controller

In this design, ports are used to connect the AssemblyController to other waveform components. These ports enable connections between the components to be defined in the SAD XML file. The ports would not be needed if connection logic for the AssemblyController were hard-coded into the AssemblyController or if none of the connections were needed.

Often, the AssemblyController implements only functionality that was already located somewhere in the waveform software design. That is, no new interfaces will be realized in the AssemblyController. In that case, which is true for our XYZ Waveform example, no new interfaces are needed, so nothing needs to be added to the IDL.

6.3 Refining API Definitions with Implementation Design

6.3.1 Use of Interfaces

Figure 6.3-1 is excerpted and modified from Figures 6.2-6, 6.2-8, and 6.2-9 to focus on the use of ports by waveform software. Each port is labeled with either a "U" (for uses port) or a "P" (for provides port)²⁴. A uses port requests data or service from another component, while a provides port returns requested data or performs a requested service. An SCA-compliant application is built using CORBA to implement client-server connections. Under this model, software assumes the role of client when it is calling through a uses port, and the role of a servant (within a server) when it is answering at a provides port.

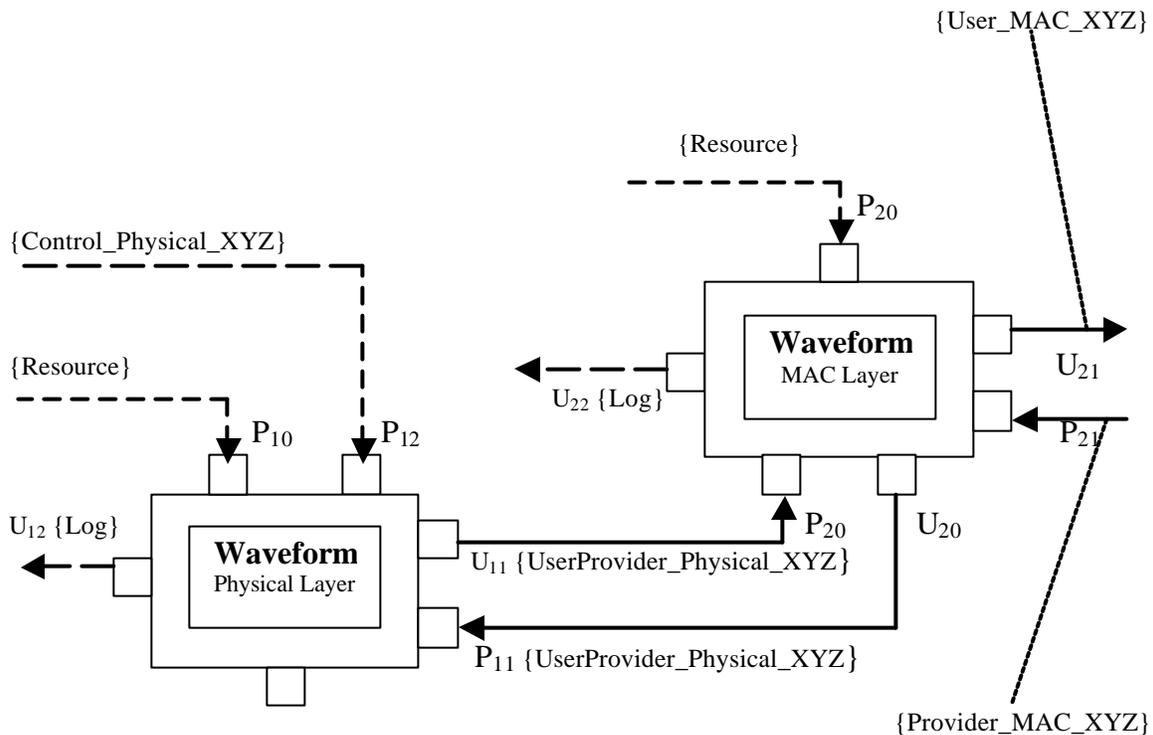


Figure 6.3-1 Port Connections for Physical and MAC Layers

Further examining Figure 6.3-1, we can see that a message that is sent by the MAC layer from port U₂₀ is the result of a message that was received at port P₂₁. Correspondingly, a message that is sent by the MAC layer from port U₂₁ is the result of a message that was received at port P₂₀.

²⁴ The name of the interface used or provided at the port is included in braces {}.

6.3.1.1 Implementing A Provides Port

Since a Provides Port is implemented as servant software, the waveform software must inherit from the IDL-generated skeleton class for the interface at that port, as described in section 5.6. The waveform software "realizes" this class by implementing each operation included in the class. If functionality for some operation in the class is not appropriate for the waveform, then the software should implement the operation anyway, even if it is a "stub" operation (i.e., an operation with no instructions)²⁵.

6.3.1.2 Implementing A Uses Port

Since a Uses Port is implemented as client software, the waveform software calls the corresponding servant software. To accomplish this, the client software maintains a pointer (of the correct type) to the servant. Good practice suggests using a CORBA _var type, because this "intelligent pointer" relieves the programmer of most responsibilities relating to memory management. Before the pointer can be used, however, the operation connectPort() must be called to establish a value for it.

6.3.1.3 Implementing Multiple Ports

Our examples tend to show exactly one Uses Port connected to exactly one Provides Port, but the actual requirements allow one-to-many, many-to-one, and many-to-many connections. Connecting many users to one provider ("many-to-one") has no effect on the servant software, because the result is simply that more than one pointer contains the address of the provider object.²⁶ Connecting many providers to one user ("to-many") adds complexity - it requires that the simple pointer to the provider be replaced by an array²⁷ of pointers, and the software must loop through the array each time the port is used.

6.3.1.4 Implementing getPort()

An application can consist of one or more CORBA objects. Each application is created by the Core Framework's ApplicationFactory implementation according to instructions provided in the appropriate XML document. The XML specifies particular objects to be created by the ApplicationFactory; each of these objects must implement the CF::Resource interface. An object created by the ApplicationFactory may create and activate other objects²⁸, but the ApplicationFactory is not aware of them. If an object (which we shall call A_0) creates another object (which we shall call A_1), the ApplicationFactory invokes the getPort operation implemented by A_0 to locate ports located in A_1 . The designer of the application has two choices for handling a getPort request for ports that are located in A_1 .

- ?? the getPort operation implemented by A_0 provides A_1 port addresses to the Core Framework
- ?? the getPort operation implemented by A_0 delegates the request to some operation (perhaps even another getPort) implemented by A_1 .

²⁵ Of course, under some conditions, invoking an error return would be more appropriate.

²⁶ Figure 6.3-4 is an example showing three uses ports connected to the provides Log port

²⁷ or CORBA sequence

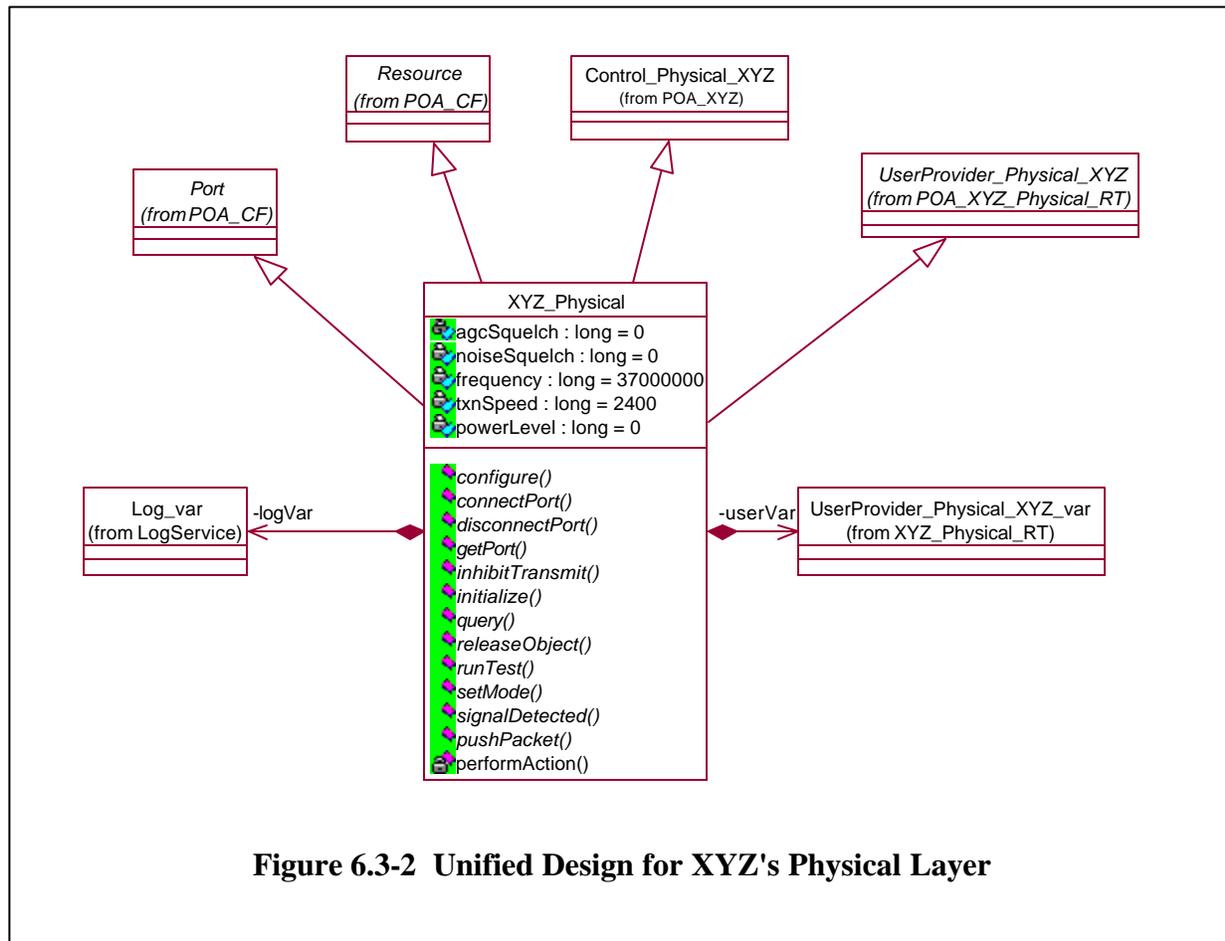
²⁸ These additional objects need not implement CF::Resource.

6.3.2 Model for Physical Layer

6.3.2.1 Unified Design

Figure 6.3-2 shows a single-class implementation design for waveform XYZ's physical layer.

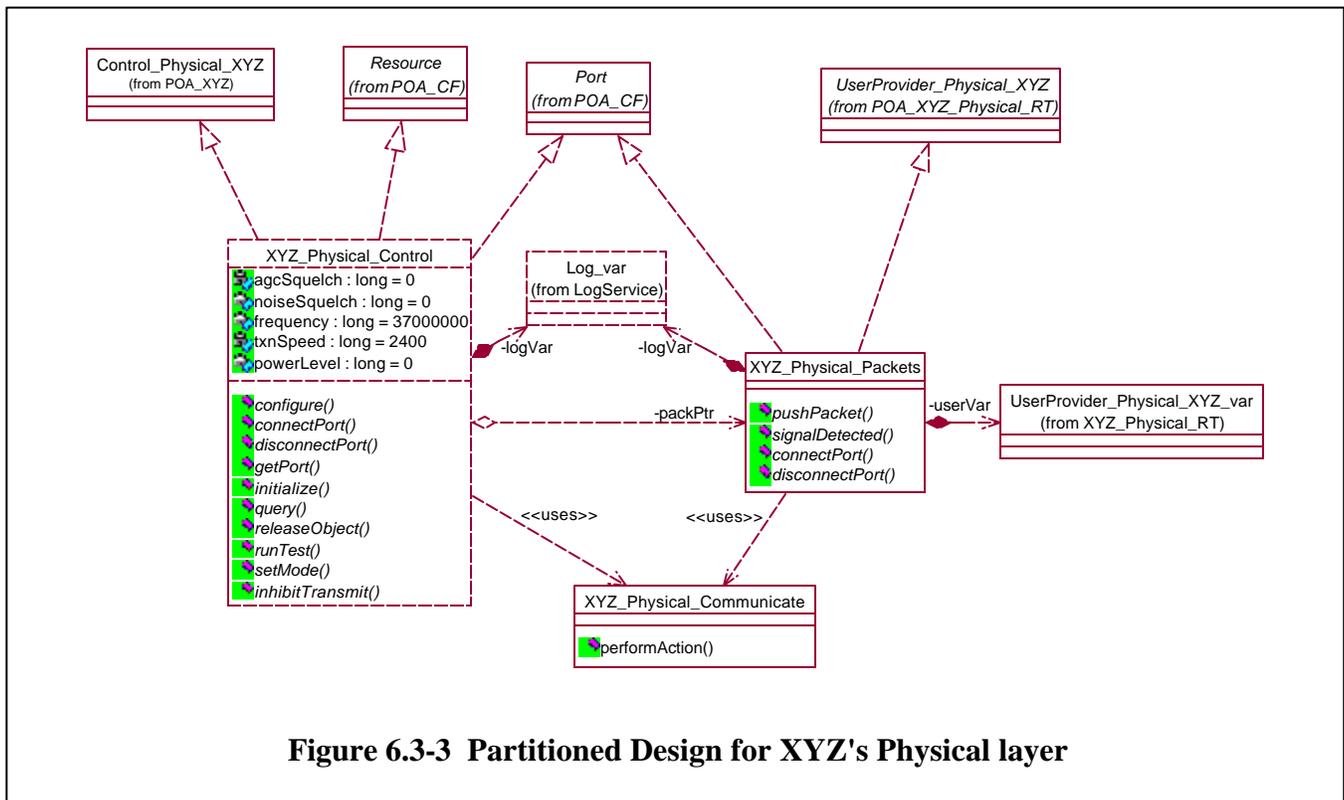
This object inherits from CF::Port so that it can connect each uses port to a corresponding provides port. It also inherits from each interface implemented as a provides port, and has an association for each uses port.²⁹ In this example design, a single operation ("performAction") is included to provide an interface with the modem hardware.



²⁹ As already described in section 6.3.1.2, the code generated from this part of the model would be pointers of the right type. For example, logVar would be a pointer which is able to point to an implementation of LogService::Log, and userVar would be a pointer able to point to an implementation of UserProvider_Physical_XYZ. The CoreFramework's ApplicationFactory implementation would be instructed via XML to invoke connectPort to link the first pointer to the actual log provides-port and the second pointer to an actual physical-layer provides-port.

6.3.2.2 Partitioned Design

Figure 6.3-3 shows a multiple-class implementation design for waveform XYZ's physical layer. XYZ_Physical_Control, which is created directly by the Core Framework as described in section 6.3.1.4, provides resource and control ports. XYZ_Physical_Packets, which is created by XYZ_Physical_Control, provides the data handling port. Uses ports are assigned to objects in such a way as to simplify program design. XYZ_Physical_Communicate is an additional object containing (static) common operations needed by operations in the other objects; the "performAction()" operation, which provides an interface with the modem hardware, was included in XYZ_Physical_Communicate. The application has two uses ports for logging, because each object may have a need to make entries in the Log. The XML shows two different ports, each having a different name, to be attached to the Log object. The getPort operation implemented by XYZ_Physical_Control reports the address of XYZ_Physical_Control for one log object name, and the address of XYZ_Physical_Packets for the other name. Both objects inherit from CF::Port, because both provide connectPort services.³⁰



After the design is complete, the model is used to generate source code templates as described in section 5.7, and the programmer adds any needed details. Appendix C contains a C++ header file generated from the design presented in this section.

³⁰ The designs presented in this guide are not intended to be complete. For example, local communications (in this case between `XYZ_Physical_Control` and `XYZ_Physical_Packets`) are not included. The links between these objects are standard links (not involving CORBA), so this communicating could be done by any means (function calls, queues, etc) normally used to communicate between local objects in the implementation language of choice, assuming of course, that SCA requirements are satisfied.

6.3.3 Model for MAC Layer

Figure 6.3-4 shows a partitioned design for the XYZ MAC layer. As is true of all multi-object designs, one object is created directly by the Core Framework's ApplicationFactory implementation. This object (XYZ_MAC_Control), which provides resource and control ports, is responsible for creating the other objects. There is a natural pairing of ports based upon the flow of messages. This natural pairing is reflected in the assignment of ports to objects, as one object moves messages "downstream" and the other moves messages "upstream".

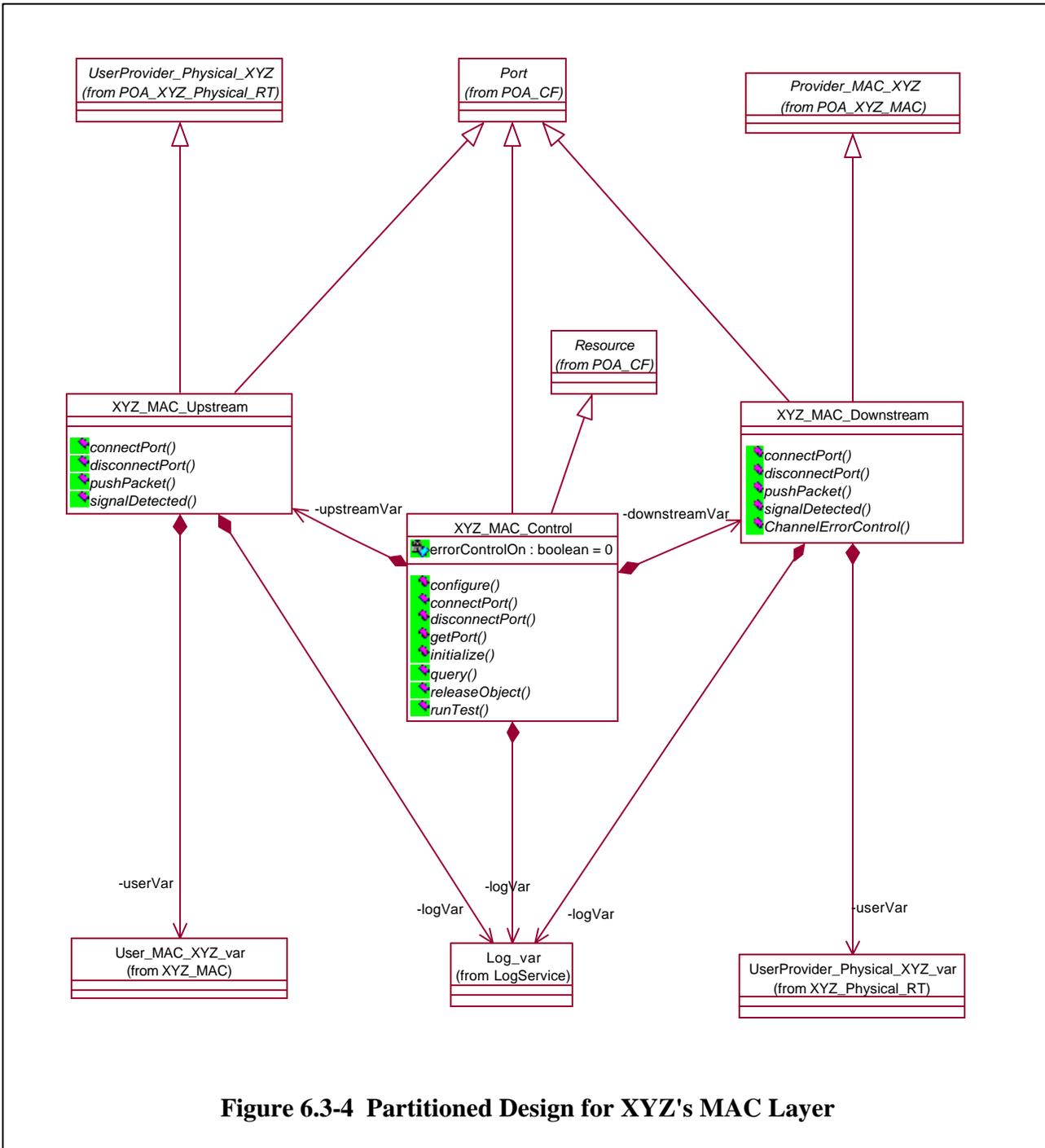


Figure 6.3-4 Partitioned Design for XYZ's MAC Layer

6.3.4 Model for Link Layer

Functionally, the XYZ Link layer performs tasks that are completely different from those performed by the MAC layer. However, the MAC layer handles two flows of messages, one downstream from an attached layer to the Physical layer, and one upstream from the Physical layer to the attached layer. Likewise, the Link layer handles two flows of messages, one downstream from an attached layer to the MAC layer, and one upstream from the MAC layer to the attached layer. Not surprisingly, then, the structure of the Link layer shown in Figure 6.3-5 is very similar to the structure of the MAC layer shown in Figure 6.3-4. In this particular case, the API includes several operations that are not needed for our application (in this example, that would include all of the operations related to flow-control). C++ errors tend to occur if there is no implementation for an operation included in the interface, so the implementation code should include each of these operations (even if the code does nothing).

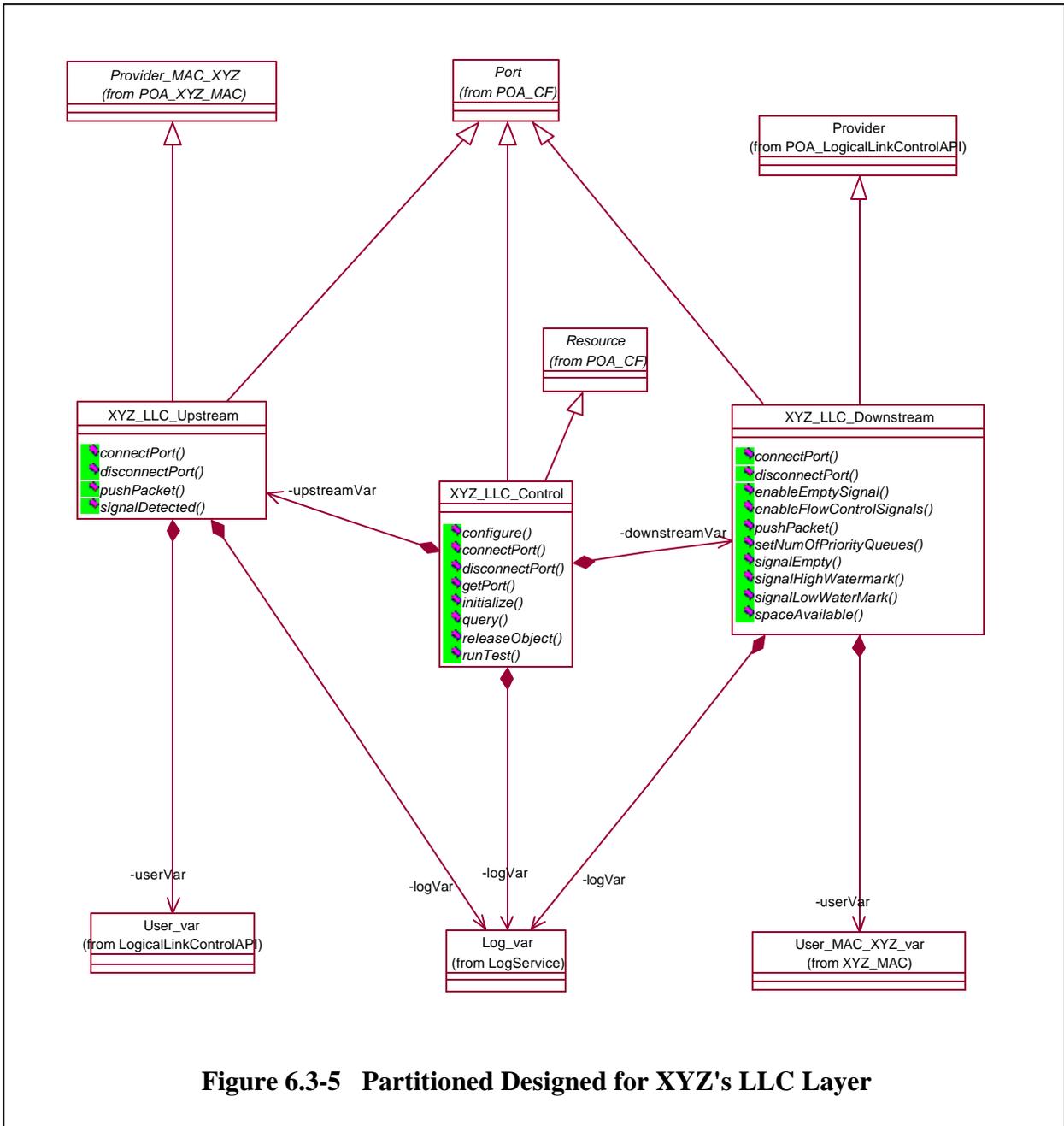
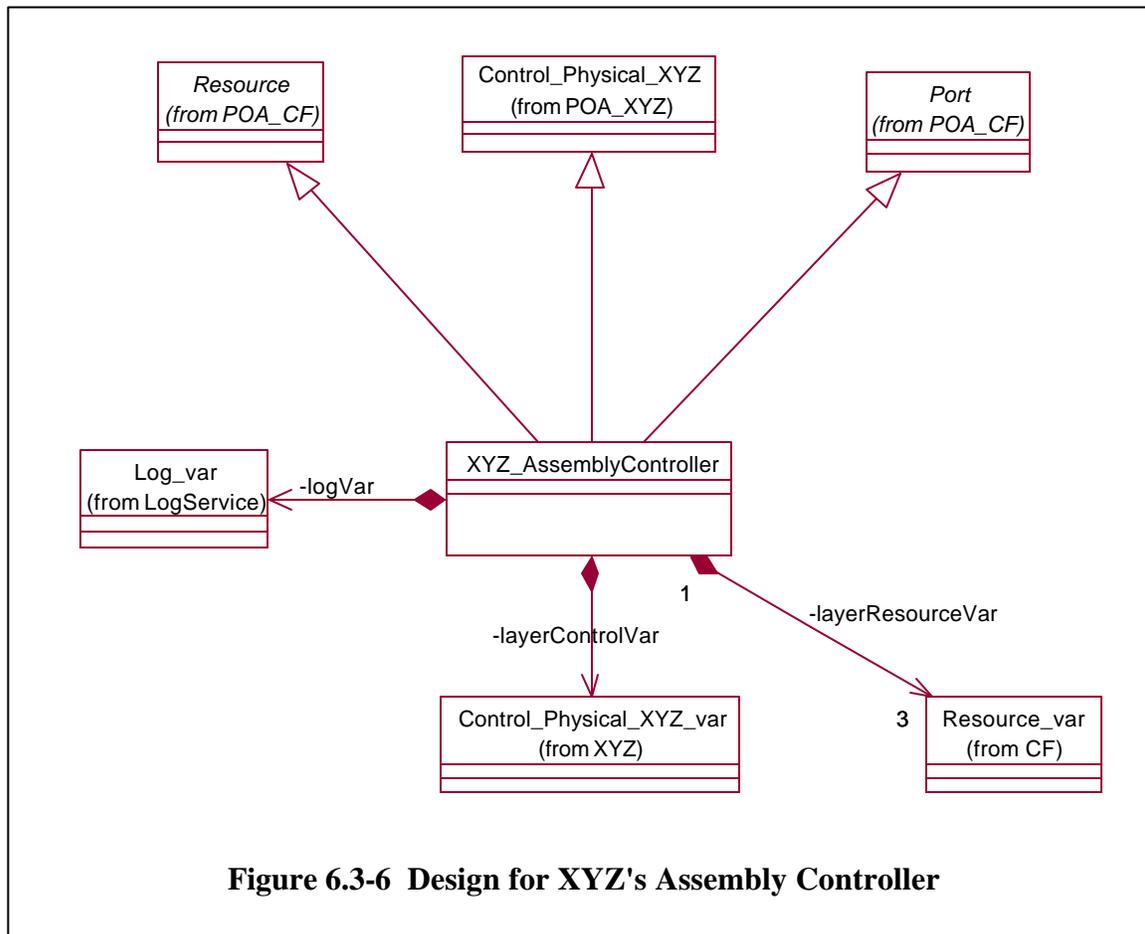


Figure 6.3-5 Partitioned Designed for XYZ's LLC Layer

6.3.5 Model for AssemblyController

The AssemblyController connects to three different layers, but only two ports are involved, because most of the activity occurs on the CF::Resource interface. The CF::Resource uses port is presented as a 1:3 association named *layerResourceVar*. The XML presents association *layerResourceVar* as three different ports, each port having a unique name, and the XML instructs the Core Framework's ApplicationFactory implementation to connect each of these ports to the CF::Resource port of one of the layers. Association *layerResourceVar* could be implemented in various ways, but a common approach consists of using an array of three pointers, each pointer corresponding to one of the names used by the XML when instructing the Core Framework. The UML for the AssemblyController is shown in Figure 6.3-6.



7 Device Creation

7.1 Device Interfaces

The device interfaces are for the implementation and management of logical Devices within the domain. The device interfaces include Device, LoadableDevice, ExecutableDevice, and AggregateDevice. The devices within the domain can be simple devices with no loadable, executable, or aggregate device behavior, or devices with a combination of these behaviors. Device Management is accomplished by the DeviceManager interface, which is responsible for creation of logical Devices and launching service applications on these logical Devices.

7.1.1 Device

A *Device* is a type of *Resource* within the domain having the requirements as stated in the *Resource* interface. This interface defines additional capabilities and attributes for any logical *Device* in the domain. A logical *Device* is a functional abstraction for a set (e.g., zero or more) of hardware devices and provides the following attributes and operations:

Software Profile Attribute – This SPD XML profile defines the logical *Device* capabilities (data/command uses and provides ports, configure and query properties, capacity properties, status properties, etc.), which could be a subset of the hardware device's capabilities.

State Management & Status Attributes – This information describes the administrative, usage, and operational states of the device.

Capacity Operations - In order to use a device, certain capacities (e.g., memory, performance, etc.) must be obtained from the *Device*. The capacity properties vary among devices and are described in the Software Profile. A device may have multiple allocatable capacities, each having its own unique capacity model.

I

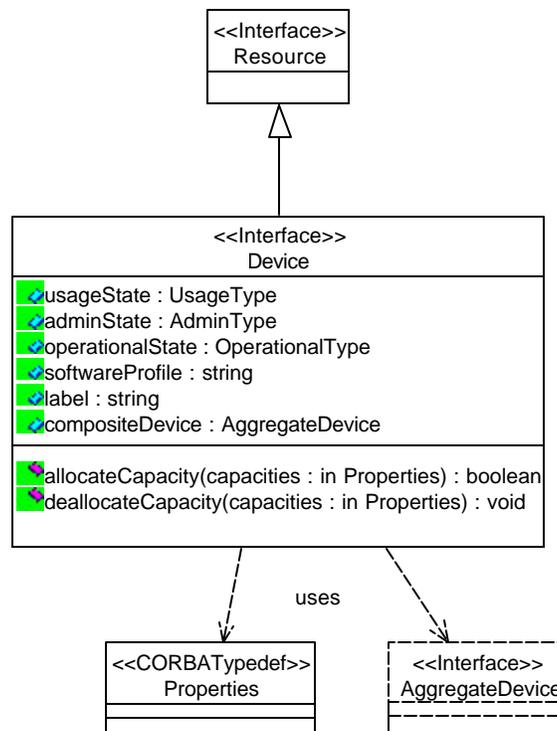


Figure 7.1-1 Device CORBA Interface UML

The Device interface provides the following operations:

allocateCapacity() : boolean

The allocateCapacity operation is used to allocate properties available to the device; the value returned reports whether the allocation was successful. In the simplest case, the property is a simple binary value (IDLE or BUSY), and the allocateCapacity operation is used to reserve the device (the operation would return FALSE if the device was in use already).

deallocateCapacity() : void

The deallocateCapacity operation is used to deallocate properties that had previously been allocated by use of the allocateCapacity operation. In the simplest case, the deallocateCapacity operation is used to return a device to its IDLE condition so that it will be available to other users.

7.1.2 LoadableDevice

This interface extends the *Device* interface by adding software loading and unloading behavior, thereby enabling software to control what is in memory available to the device. Thus, FPGA contents can be changed, and/or a digital signal processor can run a choice of code (for example, either AM or FM).

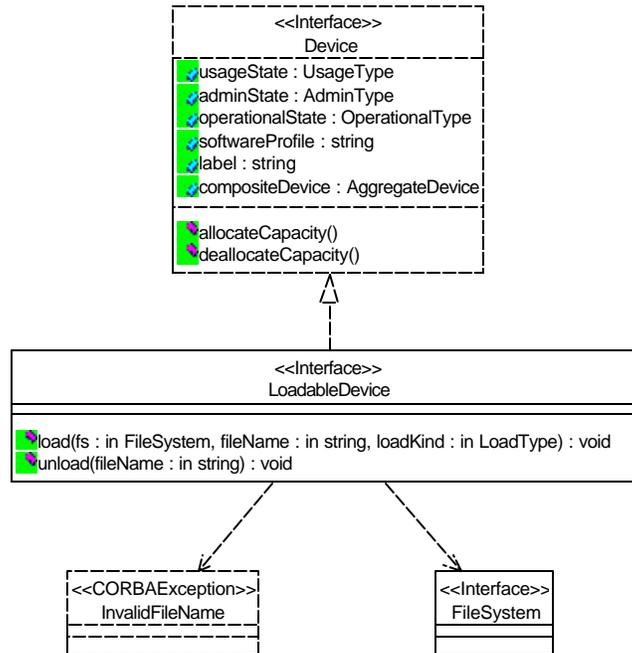


Figure 7.1-2 LoadableDevice CORBA Interface UML

The LoadableDevice interface provides the following operations:

load() : void

The load operation is used to load a file into memory. If the specified file is already loaded, a load count is incremented, but the file is not reloaded.

unload() : void

The unload operation causes the load count to be decremented. If the count has reached zero (i.e., no more users remain for the specified file) then the memory is returned to its default condition.

7.1.3 ExecutableDevice

This interface extends the *LoadableDevice* interface by adding execute and terminate behavior, thereby enabling software to determine exactly when programs will execute on the device. Thus, execution time is not necessarily connected to load time (e.g, a program need not be "load-and-go").

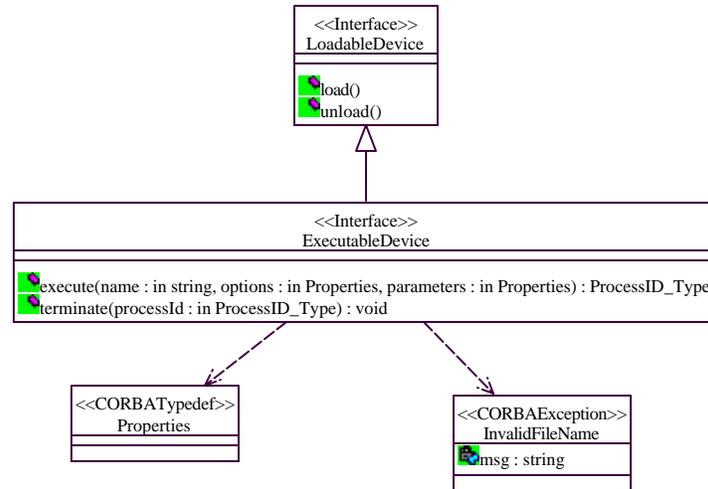


Figure 7.1-3 ExecutableDevice CORBA Interface UML

The ExecutableDevice interface extends the LoadableDevice interface by adding these operations:

execute() : Process_ID_Type

The execute operation is used to start the program, allowing the initiating software to pass parameters that will be received by the program in an argv vector as used by a standard POSIX exec call.

terminate() : void

The terminate operation is used to end a program that was started by use of the execute function

7.1.4 AggregateDevice

The *AggregateDevice* interface provides behavior that can be used to add and remove *Devices* from a composite device. This interface can be provided via inheritance or as a "provides port" for any device that is capable of an aggregate structure³¹. Aggregated *Devices* use this interface to add or remove themselves from a composite device when being created or torn-down.

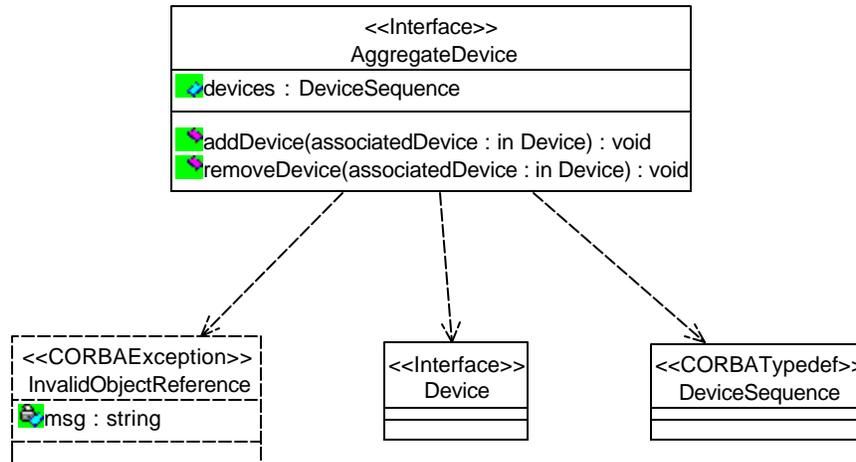


Figure 7.1-4 AggregateDevice CORBA Interface UML

7.1.5 DeviceManager

The *DeviceManager* interface is used to manage a set of logical *Devices* and services. The interface for a *DeviceManager* is based upon its attributes, which are:

- ?? Device Configuration Profile - a mapping of physical device locations to meaningful labels (e.g., audio1, serial1, etc.), along with the *Devices* and services to be deployed
- ?? File System - the *FileSystem* associated with this *DeviceManager*
- ?? Device Manager Identifier - the instance-unique identifier for this *DeviceManager*
- ?? Device Manager Label - a meaningful name given to this *DeviceManager*
- ?? Registered Devices - a list of *Devices* that have registered with this *DeviceManager*
- ?? Registered Services - a list of *Services* that have registered with this *DeviceManager*

³¹ This interface is most useful for situations in which a single piece of hardware contains units that are handled as separate devices by an SCA application.

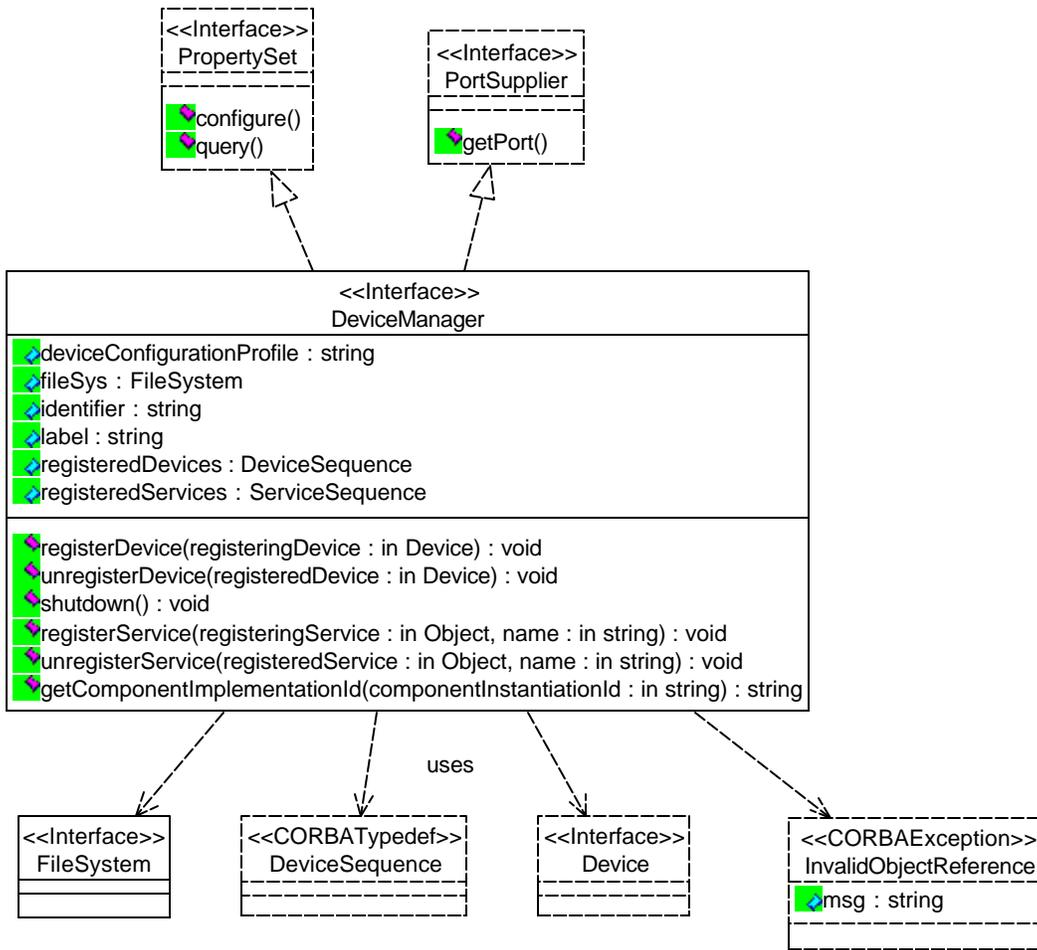


Figure 7.1-5 DeviceManager CORBA Interface UML

7.1.6 Defining the Device

This section describes the process of selecting the appropriate device interfaces for a device implementation, and the relationship to the device XML.

7.1.6.1 Selecting the Appropriate Device Interface

As is indicated in section 7.1, the device can be of type `Device`, `LoadableDevice`, or `ExecutableDevice`.³² In addition to the device type, the designer must also select a means of communicating with the waveform(s) that make use of the device.

As an example, we consider an audio device that is attached to the XYZ waveform described in sections 6.2.2 and 6.3.2. The audio device will have generic audio ports. A simple software application would be inserted between the device and the XYZ waveform to translate between the two data port forms.

Figure 7.1-6 shows which interface is associated with each of the device's ports.

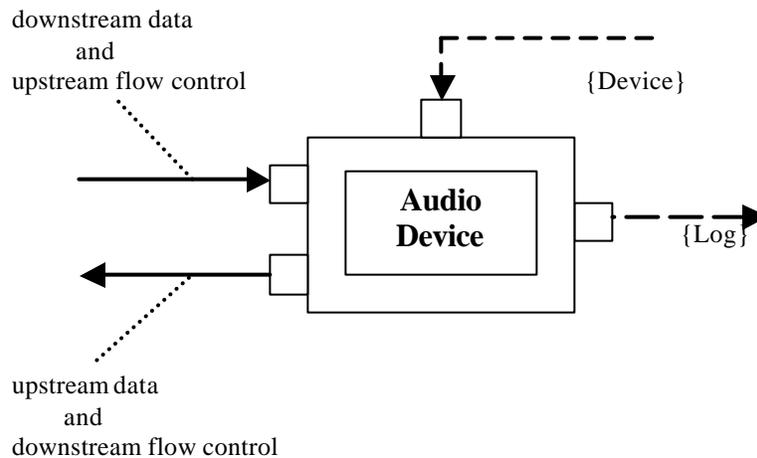


Figure 7.1-6 Audio Device Ports

³² Since an `AggregateDevice` is a composite of devices, the logic presented in Section 7.1.6.1 would be applied multiple times for an `AggregateDevice`.

7.1.6.2 Designing a Device Servant

Although the specifics of inheritance are different, designing a device servant is very similar to designing a waveform application. If the "downstream" and "upstream" data flows and flow controls are the same, the audio device realizes only two interfaces; Device (from section 7.1.1) and an appropriate interface built from I/O API Building Blocks.

Using UML notation, Figure 7.1-7 shows a simple I/O interface.

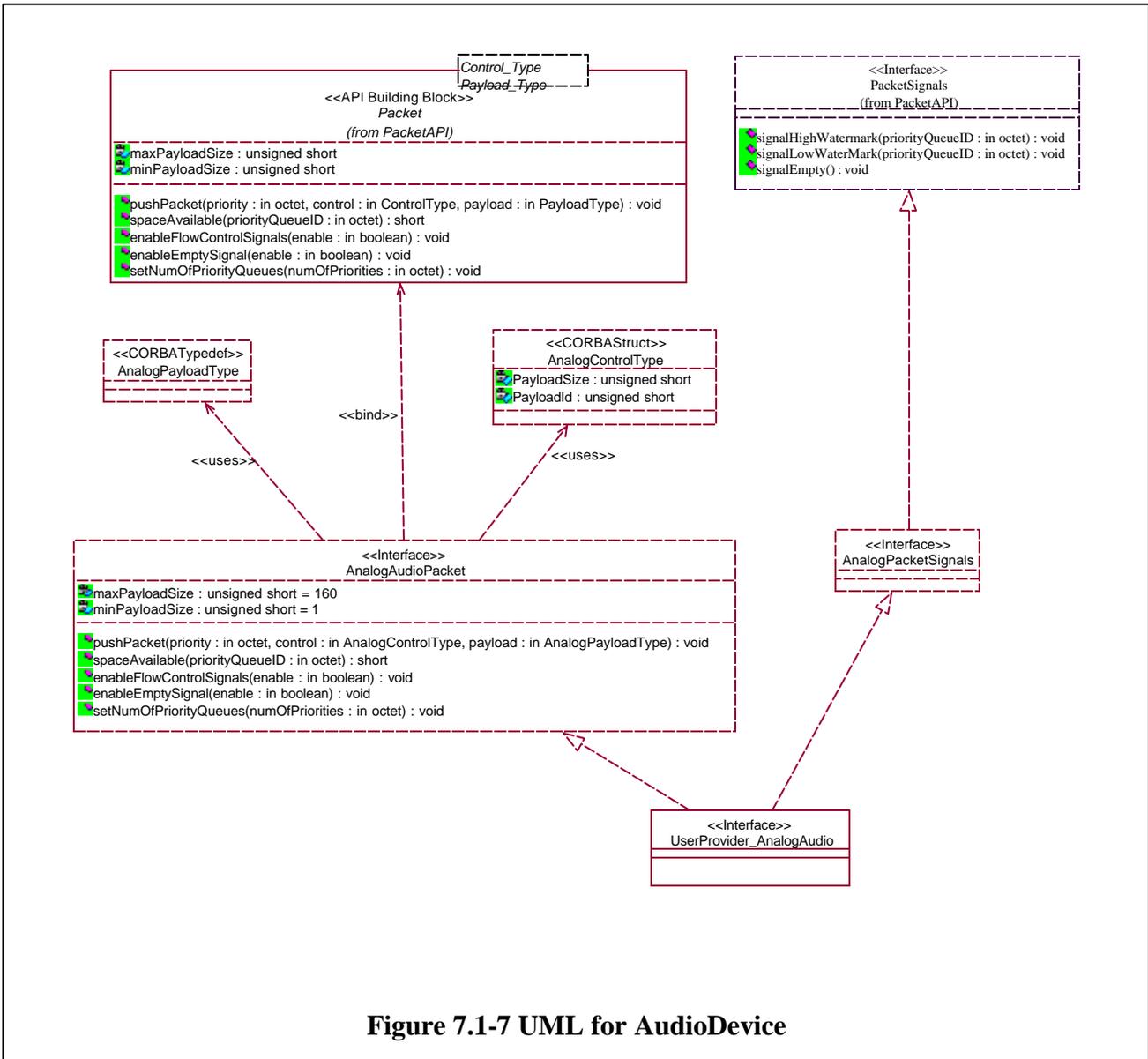


Figure 7.1-7 UML for AudioDevice

Using UML notation, Figure 7.1-8 shows a partitioned design for the resulting audio device.

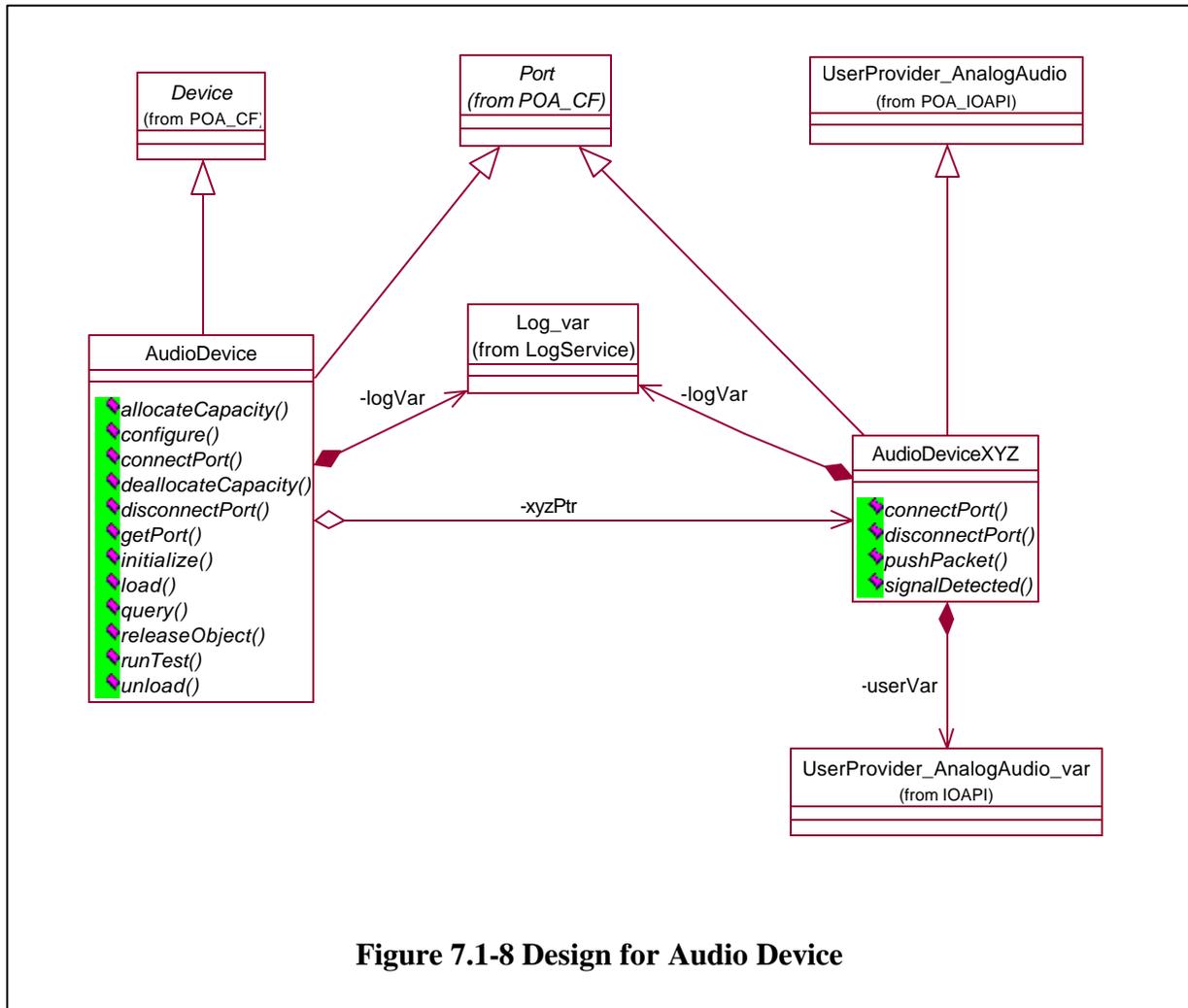


Figure 7.1-8 Design for Audio Device

7.1.6.3 Device Configuration Descriptor

This section describes the XML elements of the Device Configuration Descriptor (DCD) XML file. The *deviceconfiguration* element is the root element of the DCD. The DCD is based on the SAD (e.g., componentfiles, partitioning, etc.) DTD. The intent of the DCD is to provide the means of describing the components that are initially started on the CF *DeviceManager* node, how to obtain the CF *DomainManager* object reference, connections of services to components (CF *Devices*, CF *DeviceManager*), and the characteristics (file system names, etc.) for a CF *DeviceManager*. The *componentfiles* and *partitioning* elements are optional; if not provided, this implies no components are started up on the node, except for a CF *DeviceManager*. If the *partitioning* element is specified then a *componentfiles* element must also be specified.

The *deviceconfiguration* element's *id* attribute is a unique identifier within the domain for the device configuration. This *id* attribute is a UUID value as specified in section 4.1. The *name* attribute is the user-friendly name for the CF *DeviceManager*'s label attribute.

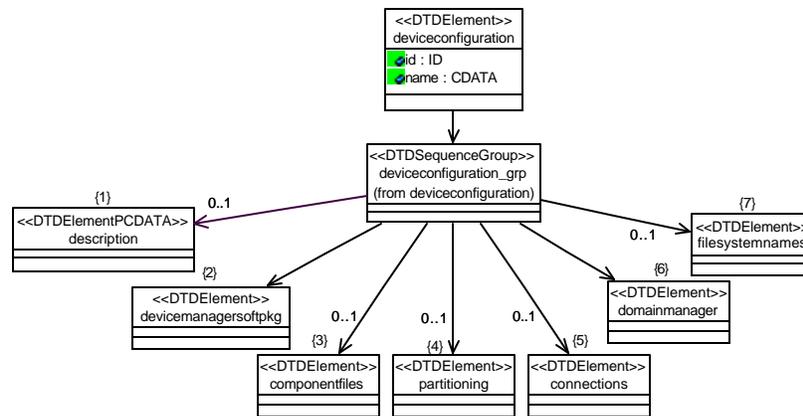


Figure 7.1-9 deviceconfiguration Element Relationships

A DCD should be provided as part of the software documentation for a *DeviceManager* implementation. The DCD should contain all of the mandatory XML elements as well as many of the optional elements. For example, the *description* element of the DCD is optional but should be provided. The *description* element can be used to provide text information about the *DeviceManager* implementation and how the DCD is utilized by the implementation. As another example, the *fileSystems* element is optional but could be provided. The *fileSystems* element documents the names of the host file systems for which the *DeviceManager* implementation should create *FileSystem* servers components. If the *fileSystems* DCD element is not provided the *description* element or XML comment should indicate whether a system integrator could add the *fileSystems* XML element at deployment time. This would indicate that the *DeviceManager* implementation has the capability to generically create *CF::FileSystems* for the host file systems specified by the *fileSystems* DCD element.

7.2 Device Package Descriptor

The SCA Device Package Descriptor (DPD) is the part of a Device Profile that contains hardware device Registration attributes, which are typically used by a Human Computer Interface application to display information about the device(s) resident in a SCA-compliant radio system. DPD information is intended to provide hardware configuration and revision information to a radio operator or to radio maintenance personnel. A DPD may be used to describe a single hardware element residing in a radio or it may be used to describe the complete hardware structure of a radio. In either case, the description of the hardware structure should be consistent with hardware partitioning as described in the Hardware Architecture Definition in section 4.0 of the SCA.

The *devicepkg* element is the root element of the DPD. The *devicepkg* id attribute uniquely identifies the package and is a DCE UUID, as defined in section 4.1. The version attribute specifies the version of the *devicepkg*. The format of the version string is numerical major and minor version numbers separated by commas (e.g., "1,0,0,0"). The name attribute is a user-friendly label for the *devicepkg*

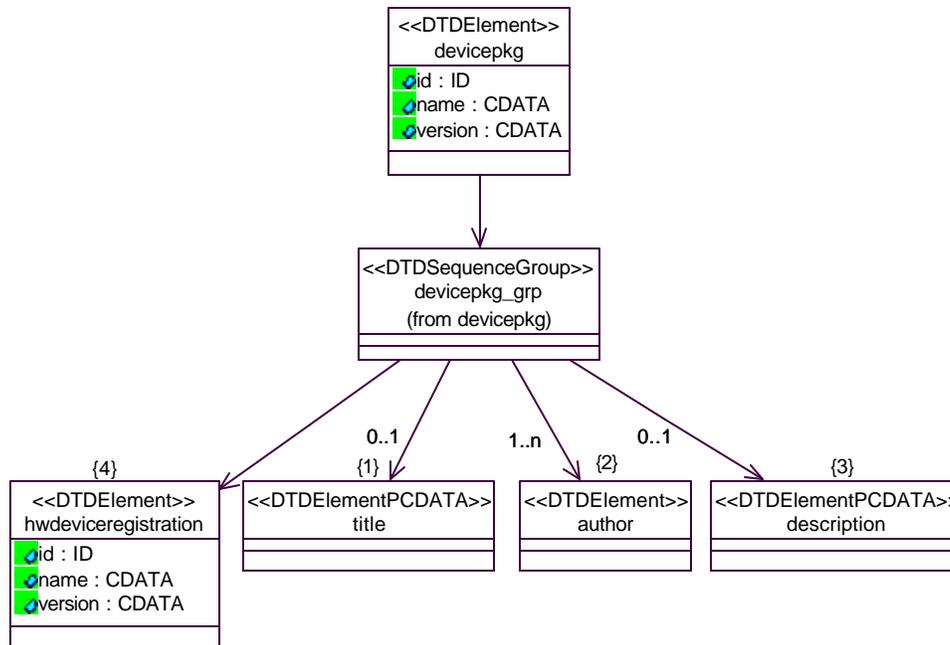


Figure 7.2-1 devicepkg Element Relationships

A DPD should be provided as part of the documentation for a hardware device. The DPD should contain all of the mandatory XML elements as well as some of the optional elements. In particular, the *description* element of the DPD is optional but should be provided. The description element can be used to provide text information about the device. The *hwdeviceregistration* element contains specific information about the hardware.

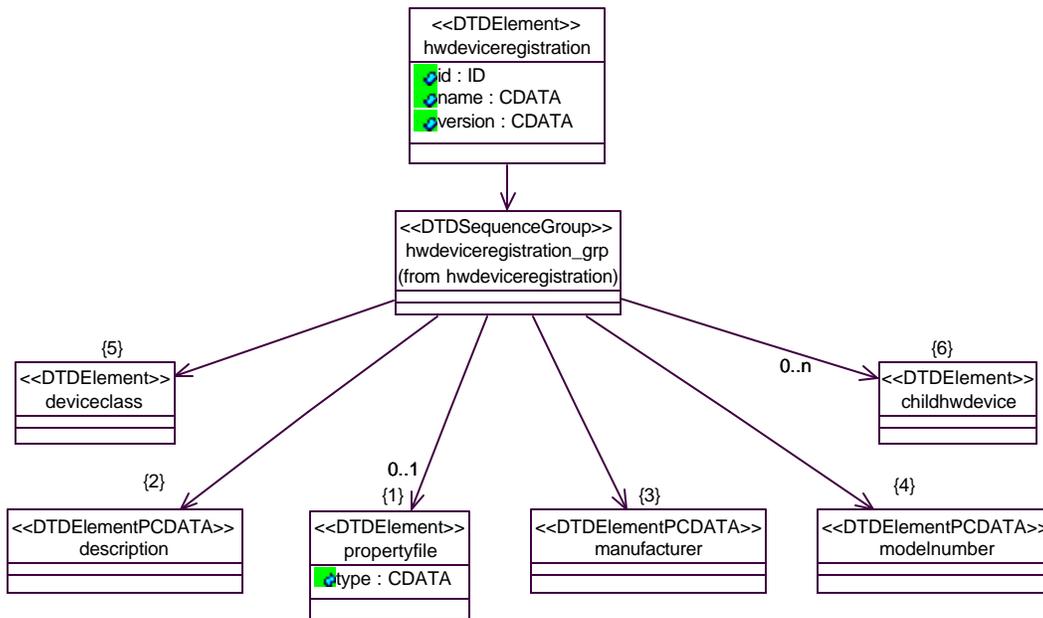


Figure 7.2-2 hwdeviceregistration Element Relationships

The *hwdeviceregistration* element may have any number of *childhwdevice* elements. Each *childhwdevice* element represents a component/subsystem of the device; thus, this structure provides a way of documenting the complete structure of the hardware device.

7.3 DomainManager Configuration Descriptor

This section describes the XML elements of the *DomainManager* Configuration Descriptor (DMD) XML file. The *domainmanagerconfiguration* element is the root element of the DMD. The *domainmanagerconfiguration* element id attribute is a DCE UUID that uniquely identifies the *DomainManager*. The id is a DCE UUID value as specified in section 4.1.

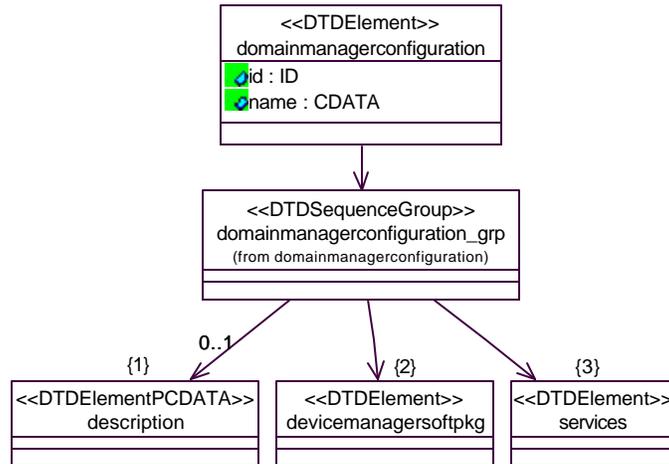


Figure 7.3-1 domainmanagerconfiguration Element Relationships

A DMD should be provided as part of the software documentation for a *DomainManager* implementation. The DMD should contain all of the XML elements. The description element of the DMD is optional but should be provided. The description element can be used to provide text information about the *DomainManager* implementation and how the DMD is utilized by the implementation. The *services* element specifies which service (e.g., Log) instances will be used.

8 UI Discussion

8.1 Introduction

JTRS User Interface applications are used to setup, control, and monitor JTRS compliant Core Framework and radio applications (waveforms).

The design of an operator console and the methods used to control a JTRS radio platform is radio platform and program specific. For example, a handheld radio may only provide a small operator keypad and display, while networked radio system could provide a desktop workstation as the operator console. With so many variations of operator control of a JTRS radio, no one approach for a UI design can be mandated. Operator control messages do have to arrive at the waveform application in the form of calls to operations realizing appropriate CORBA interface(s) - but this requirement can be met in various ways. The following sections provide examples of UI approaches for a JTRS compliant radio.

8.2 Direct CORBA Links

One approach to delivering CORBA messages to the waveform application is to build the UI itself on a CORBA platform. This is a natural structure when the operator's console is a workstation running a graphical user interface (GUI).

An example of a remotely located GUI utilizing Java and CORBA technologies is shown in Figure 8.2-1. The CF interfaces, defined in the SCA CF IDL, are used in the GUI for control and monitoring of the CF. Waveform specific interface-APIs are used to control the Application waveform. CORBA calls in this environment are remote calls across the network connection.

In the XYZ example (see section 6.2.2.5, for example) the GUI performs CORBA calls using the appropriate APIs for the CF components and WF Resource components.

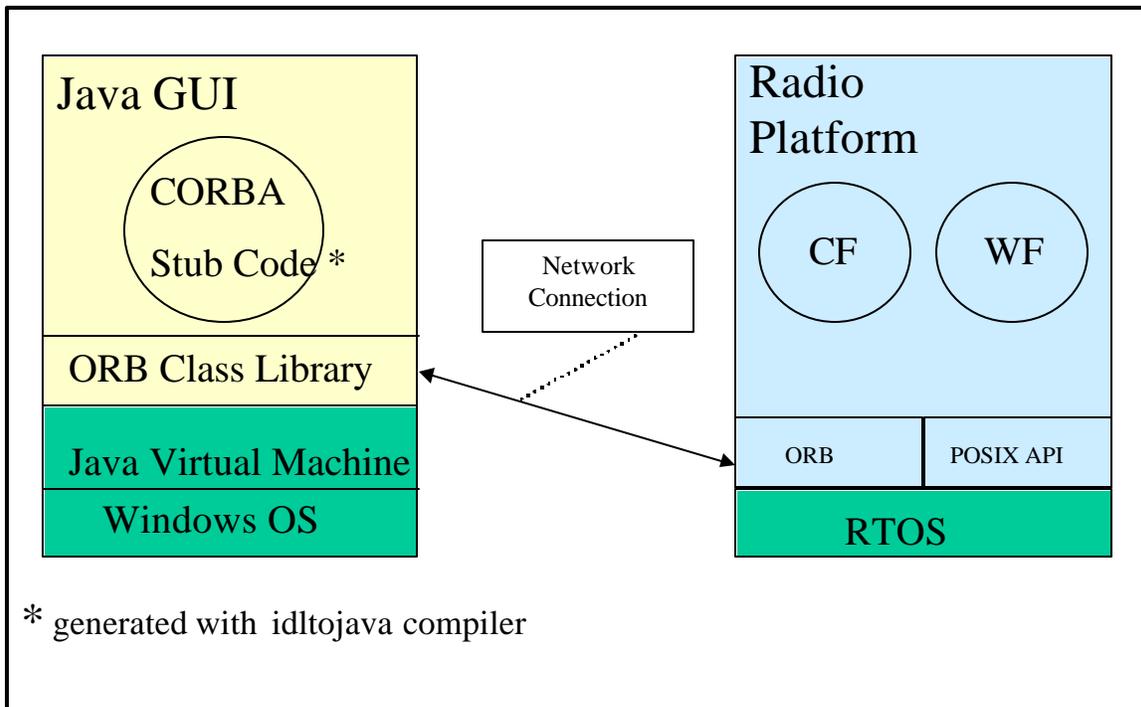


Figure 8.2-1 Direct CORBA Link Block Diagram

8.3 Non-Direct CORBA Links

Another approach is to have "adapter" software in the radio that translates between a proprietary messaging format and the appropriate SCA-defined CORBA APIs. This method would be especially appropriate if the operator interface has limited capabilities, or in order to connect a legacy controller to a JTRS-compliant radio application.

Figure 8.3-1 shows an example of a proprietary controller connected to adapter software in the radio. A serial link provides the actual physical connection between the controller and the radio. The adapter converts between the proprietary format and appropriate operations defined in the CF interface and the waveform-specific interface. CORBA calls in this environment are local calls within the processor.

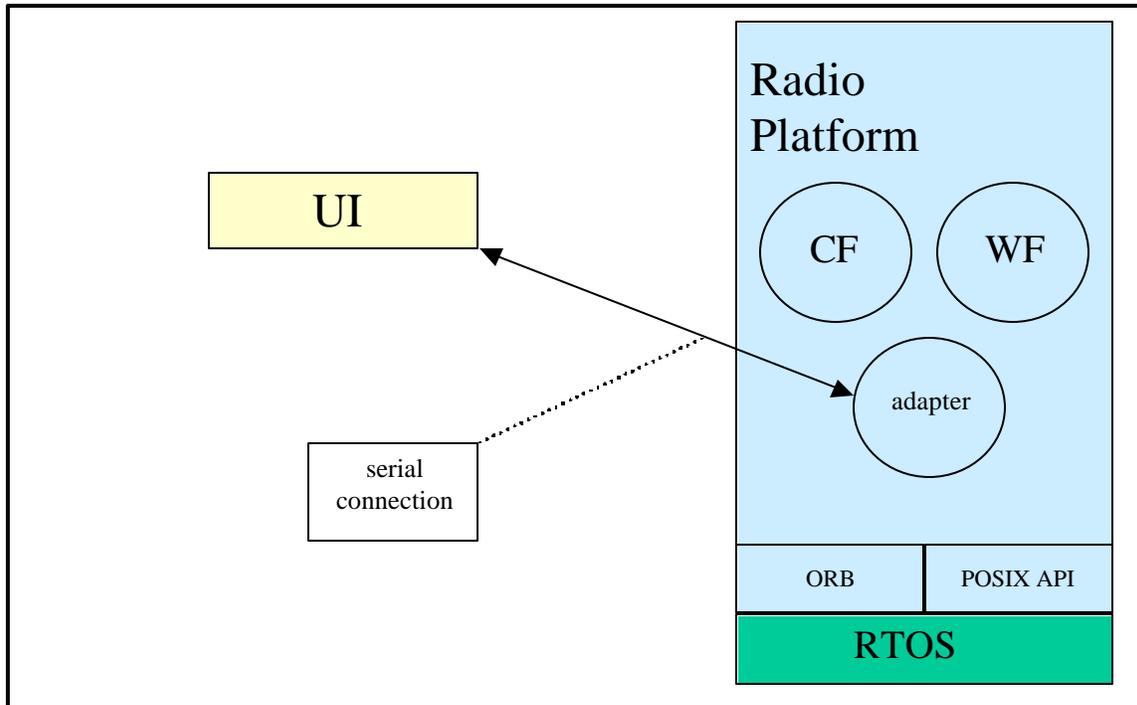


Figure 8.3-1 Non-direct CORBA Link Block Diagram

In the XYZ example, a proprietary formatted serial message is sent to the radio from the UI. The adapter processes the message and forwards the call to appropriate component (CF component or WF Resource) using the proper CF or WF API.

9 Appendices

9.1 *Appendix A – XML Introduction*

9.2 *Appendix B – IDL for XYZ Waveform Physical layer*

9.3 *Appendix C – Header Files for XYZ Waveform Physical Layer*

9.4 *Appendix D – XML for a Sample Waveform*

9.5 *Appendix E – XML for a Sample Device*