

**Application Program Interface Supplement
to the
Software Communications Architecture Specification**

JTRS-5000API
V3.0
August 27, 2004

Prepared by
Joint Tactical Radio System (JTRS) Joint Program Office

Revision Summary

1.0	Initial release.
1.1	Incorporate approved Change Proposals, numbers 470
2.2.1	Document numbering change for consistency with SCA main document numbering.
3.0	No changes.

Table of Contents

1	INTRODUCTION.....	1-1
1.1	SCOPE.....	1-1
1.1.1	Service.....	1-1
1.1.2	Service Definition.....	1-1
1.1.3	Building Block.....	1-1
1.1.4	Interface.....	1-2
1.1.5	Application Program Interface.....	1-2
2	REFERENCE DOCUMENTS.....	2-1
3	APPLICATION PROGRAM INTERFACES.....	3-3
3.1	GENERAL.....	3-3
3.2	JTRS APIs.....	3-3
3.2.1	API Definition.....	3-4
3.2.2	Relationship of APIs to the SCA.....	3-5
3.3	API SERVICES.....	3-8
3.4	BUILDING BLOCKS.....	3-8
4	REQUIREMENTS.....	4-1
4.1	GENERAL REQUIREMENTS.....	4-1
4.1.1	Name Scoping.....	4-1
4.2	API REQUIREMENTS.....	4-1
4.2.1	API Usage and Creation.....	4-1
4.2.2	API Transfer Mechanisms.....	4-1
4.3	SERVICE DEFINITION REQUIREMENTS.....	4-3
4.3.1	Non-JTRS Service Definitions.....	4-3
4.3.2	JTRS API Service Definition Format.....	4-4
4.3.3	Service Definition Identification and Registration.....	4-4
5	ACRONYMS, ABBREVIATIONS, AND OTHER DEFINITIONS.....	5-1
5.1	ACRONYMS AND ABBREVIATIONS.....	5-1
5.2	DEFINITIONS.....	5-2

List of Appendices

- A SERVICE DEFINITION DESCRIPTION**
- B TABLE OF SERVICES**
- C GENERIC PACKET BUILDING BLOCK SERVICE DEFINITION**
- D PHYSICAL REAL TIME BUILDING BLOCK SERVICE DEFINITION**
- E PHYSICAL NON-REAL TIME BUILDING BLOCK SERVICE DEFINITION**
- F MEDIA ACCESS CONTROL BUILDING BLOCK SERVICE DEFINITION**
- G LOGICAL LINK CONTROL BUILDING BLOCK SERVICE DEFINITION**
- H I/O BUILDING BLOCK SERVICE DEFINITION**
- I NETWORK BUILDING BLOCK SERVICE DEFINITION** (to be provided at a later date)

List of Figures

- Figure 3-1. JTRS APIs Mapped to the SCA Software Structure.3-4
- Figure 3-2. Relationship Example of APIs3-5
- Figure 3-3. Connection of Physical and MAC Components.....3-7
- Figure 4-1. Standard and Alternate Transfer Mechanism.....4-2
- Figure 4-2. Reusing an Existing API Without an IDL Interface.....4-3

1 INTRODUCTION.

This supplement to the JTRS Software Communication Architecture (SCA) Specification provides details and requirements for standard interfaces used by SCA applications. These standard interfaces, called application program interfaces (APIs), permit and encourage portability of software components.

1.1 SCOPE.

This document supplements the SCA Specification with standardized APIs and the building blocks to build new APIs.

This supplement excludes APIs for security interfaces that are addressed in a separate Security Supplement. Security APIs are, however, required to follow the approaches used in this supplement.

{This supplement does not address at this time any APIs associated with an operator interface. The operator interface is often referred to as a GUI (graphical user interface), HMI (human-machine interface) or HCI (human-computer interface). The development of APIs and corresponding building blocks associated with the operator interface has been delayed until later. This supplement will be updated to include operator interface API information when it is developed.}

The requirements are written with the assumption that the reader is familiar with the concepts of object oriented programming and the use of interface definition language (IDL) from the Object Management Group. Key definitions.

Certain definitions are key to understanding the material in this supplement and they are presented here. A larger list of definitions is found in section 5.2.

1.1.1 Service.

A service is the behavior provided by a given part of a system.

1.1.2 Service Definition.

A service definition documents an API, specifying the interfaces (requests, responses, indications, confirmations, and acknowledgements), behavior, state information, and exceptions that define the particular services at that interface.

1.1.3 Building Block.

A building block is an abstracted element used to provide the approach and interface definitions for services that are common to many APIs. Building Blocks are conceptual elements that are used to form the basic elements of an API. Building Blocks are converted into API elements by taking the abstract aspects of the Building block and forming concrete IDL interfaces from them. An API interface can then formed by inheriting these concrete IDL interfaces. Building blocks are used to foster reuse and commonality of accepted interface designs between different implemented service definitions.

1.1.4 Interface.

An Interface is described using IDL and is the concrete definition developed from an abstract Building Block.

1.1.5 Application Program Interface.

An Application Program Interface (API) is a definition and standardization of common interfaces between functional partitions of an SCA application. An API is defined in terms of IDL and is formed by inheriting Interfaces that were derived from previously defined Building Blocks.

2 REFERENCE DOCUMENTS.

JTRS-5000SCA , “Software Communications Architecture Specification (SCA)”, V2.2.1, April 30, 2004

Support and Rationale Document for the Software Communications Architecture Specification, MSRC-5000SRD

Data Link Provider Interface Specification, Revision 2.0.0, August 20, 1991

Other documents referenced by the appendices of this supplement.

3 APPLICATION PROGRAM INTERFACES.

3.1 GENERAL.

An Application Program Interface (API) is an agreement between two components (e.g., a network resource and a link resource) on the language / semantics used to communicate across their interface and what behavior will occur as a result of invoking operations defined at their interface. The API uses a Transfer Mechanism that provides the communication between the two components (e.g., waveform service provider and a service user). A Service Definition defines the operations (primitives), the attributes (variables), their representation (structures, types, formats) and the behavior at the interface where the services are provided.

Standardized APIs are essential for portability of applications and interchangeability of devices. Regardless of the operating environment or the software language used in an implementation, a standardized API guarantees the service provider and user can communicate. The degree of granularity of the desired, or achievable, portability and interchangeability determines the number and location of APIs to be defined and standardized.

3.2 JTRS APIs.

The JTRS goal for application portability is to maximize reuse of software components and hardware devices. The maximum is limited by technology, willingness of developers and availability of products, and needs of users. JTRS APIs will be provided to support all degrees of portability, starting first with applications and growing with time to encompass reusable components.

Initially, JTRS APIs are defined at interfaces where, in today's environment, waveforms can be partitioned for the most porting utility. These interfaces are

- A. I/O. This interface is at a domain component containing voice and/or data processing. Common audio and data interfaces are thus provided to any application needing such an interface.
- B. Security. The interfaces to security applications, components, and devices are covered in the Security Supplement to the SCA.
- C. Network. This interface is at the component(s) used for waveform network behavior.
- D. Logical Link Control. This interface is at the component(s) used by waveform applications requiring link layer behavior (Data Link service conforming to the Open Systems Interconnection (OSI) model for networking systems).
- E. MAC. This interface, while analogous to and fully supporting the services of the Medium Access Control sublayer of the OSI model Link Layer, is provided for waveform applications that have media access control behavior (e.g. transmit/receive time slot control in TDMA, error correction coding control, etc.).
- F. Physical. This interface provides for translation from bits/symbols to RF and RF to bits/symbols for waveform transmission and reception.

Figure 3-1 shows these component types mapped to the software structure of the SCA. Depending on the waveform, link behavior can be on either (or both) the Red or Black side.

Depending on the implementation, the MAC API may be located in a (CORBA-capable) "modem" or in a Black general purpose processor. Again depending on implementation, the Physical API may not be visible, with all necessary waveform interface definition contained in the MAC API.

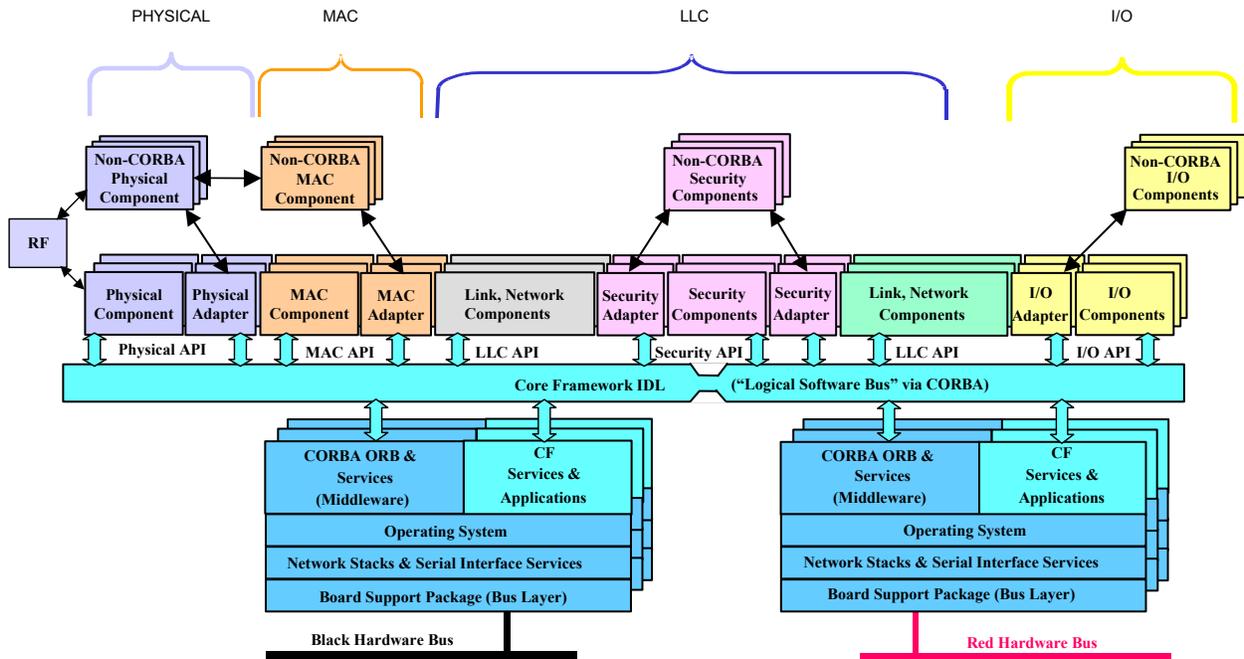


Figure 3-1. JTRS APIs Mapped to the SCA Software Structure.

3.2.1 API Definition.

Each JTRS component requires standardized interfaces to enhance portability of waveforms. Figure 3-2 shows a relationship example with four component interface locations (LLC, MAC, Physical, and I/O) standardized in this supplement. It also shows the APIs consisting of "A" interfaces and "B" interfaces. "A" interfaces support data and real-time control. "B" interfaces support set-up and initialization from applications, other levels, and user interfaces. (In this context and for API definitions, real-time means in conjunction with the data transfer and non-real-time refers to controls and status independent of data transfer.)

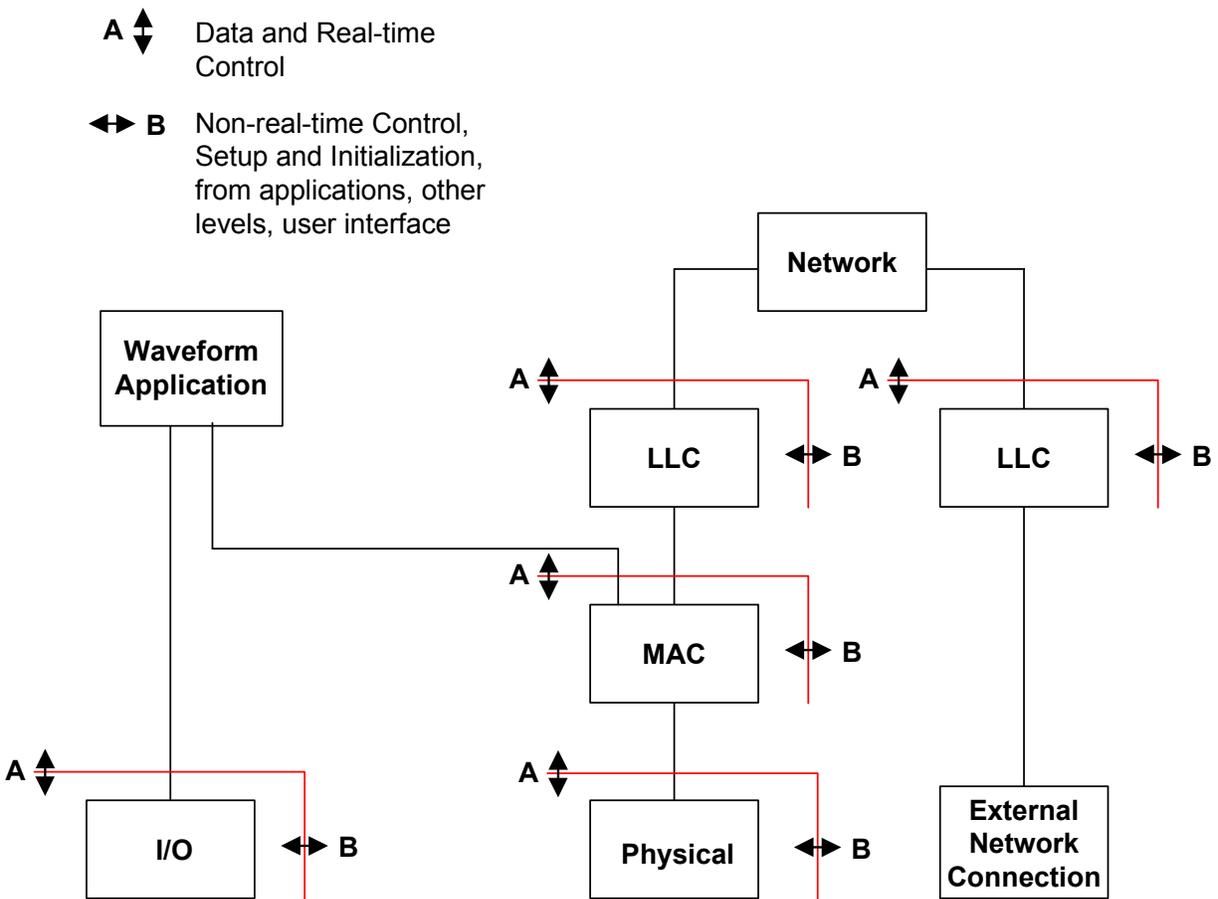


Figure 3-2. Relationship Example of APIs

3.2.2 Relationship of APIs to the SCA.

3.2.2.1 APIs, Components, and Ports.

Waveform components can be viewed as elements connected in series in an OSI type stack. Each component provides services in that stack to the layer above and as such is defined as a service provider. The user of those services is termed the service user. The interfaces for these services are defined in terms of the information supplied to and output from the service provider. These are the "A" interfaces shown in Figure 3-2. In addition, waveform components require configuration, control, and setup which are independent of layer-to-layer communication. These are the "B" interfaces shown in Figure 3-2. "B" interfaces can be thought of as parallel interfaces in that they do not require series connection with another standard JTRS component and may all be connected to a centralized controller component (e.g., a waveform controller).

Figure 3-3 shows a Physical Component and a MAC component for a waveform, each with a *CF::Resource* or a *CF::Device* interface. In addition, each component has ports. Ports are the vehicle by which components are connected together to produce a waveform application. The interfaces defined in this document, which form the APIs for a standard component, appear at some of these ports. In Figure 3-3, the service provider interface for Physical real time control

and data flow (downstream) appears at the top right port of the Physical component. The Physical component will implement the server for this interface. The service user interface for Physical real time control and data flow (upstream) appears at the lower left port of the MAC component. The MAC component will implement the server for this interface. Some waveforms will not require MAC functionality, in which case the service user would be some other waveform component. The service provider for the Physical non-real time control interface appears at the non-real time control port. The *CF::Port* interface appears at the top left port of the MAC component and the lower right port of the Physical component. In other words clients of component interfaces must implement the *CF::Port* interface.

3.2.2.2 APIs and the Software Component Descriptor.

The Software Component Descriptor (SCD) for a waveform component contains *port* and *interface* elements. A *port* element consists of the name of a *port*, identification of whether it is a provider or a user, a classification of the port and the repository ID of the interface it provides or uses. An interface element identifies the name of the interface, its unique repository ID and interfaces that it inherits.

Each IDL interface in an API will be assigned a repository ID by the ORB IDL compiler. This repository ID will appear in the port and interface elements of the Software Component Descriptor to associate the services with the ports that provide and use them.

3.2.2.3 APIs and the Software Assembly Descriptor.

The Software Assembly Descriptor (SAD) for a waveform contains *connectinterface* elements which identify the *providesports* and *usesports* that are to be connected together. These ports are identified by their port names in the SCD.

3.2.2.4 Connecting Components Together.

The *CF::ApplicationFactory* reads the SAD and the SCD. The SAD identifies the ports to which components should be connected. The SCD identifies the interfaces that the ports provide and use. The SCA does not require but does not prohibit the *ApplicationFactory* from using the interface information in the *SCD* to determine if the interfaces at the ports enumerated in the SAD align and aborting the connection process if there is a misalignment.

In Figure 3-3, the *CF::ApplicationFactory* would call *CF::Resource::getPort* on the Physical and MAC components for each port on those components. The *getPort* operation will return the object that is associated with the named port. The *CF::ApplicationFactory* will then connect the object that implements the provides port with the object that implements the uses port using the *CF::Port::connectPort* operation.

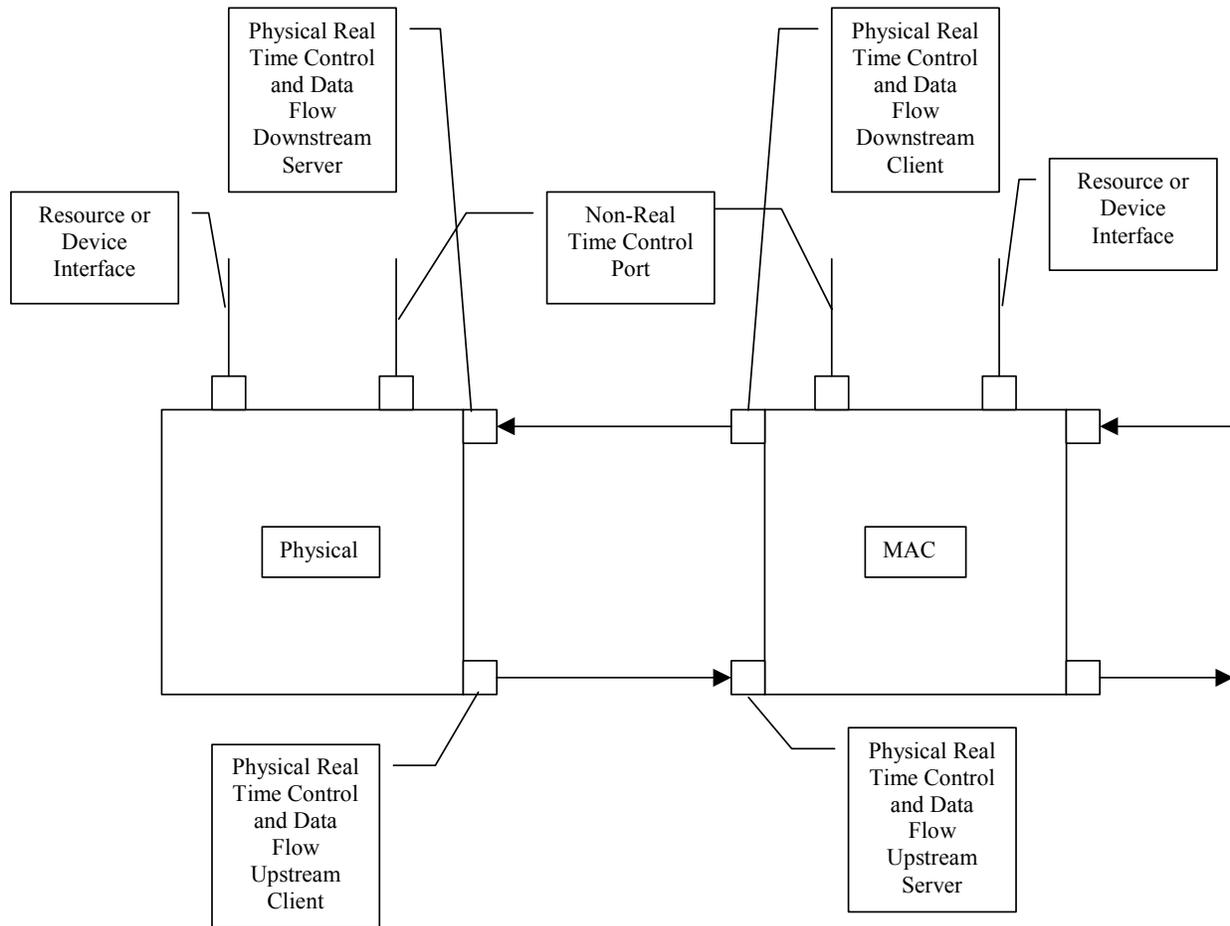


Figure 3-3. Connection of Physical and MAC Components

For example the port on the Physical component that is associated with the object that implements the downstream interface for Physical real-time control and data flow may have a name "Provider::Downstream". The port on the MAC component that is associated with the object that uses the downstream interface for Physical real-time control and data flow may have a name "User::Downstream". This using-object must implement the *CF::Port* interface. For this case, the following sequence of events occurs to connect the provider port to the user port.

1. The *ApplicationFactory* calls *getPort* on the Physical component with "Provider::Downstream" as the parameter.
2. The *getPort* operation returns the object reference for the service provider.
3. The *ApplicationFactory* calls *getPort* on the MAC component with "User::Downstream" as the parameter.
4. The *getPort* operation returns the object reference for the service user.
5. The *ApplicationFactory* narrows the object reference to the *CF::Port* interface.

6. The *ApplicationFactory* calls *connectPort* operation with the object reference of the service provider as the parameter.

3.3 API SERVICES.

It is desirable that APIs for JTRS waveforms be common, to promote the use of multipurpose devices and to support the long-term goal of portable / reusable components. The range and variety of services at the various interfaces, most notably the MAC and Physical, make a common API for all waveform applications large and burdensome for resource constrained implementations. API Building Blocks, as described in section 3.4, have been defined to bridge this dichotomy.

The services defined by an API at a particular interface include the data transfer, RT control, and NRT control and status that are used at that interface. The services provided at the interfaces defined in section 3.2 that are used by more than one waveform application are summarized in Appendix B. These services are collected as candidates for inclusion in the Building Blocks. Services used only by a single application are defined in the applicable API in addition to those included in Building Blocks.

3.4 BUILDING BLOCKS.

A goal would be to have a standard set of APIs for all current and future waveforms. As stated above, this is not practical because many waveforms have major differences in interface requirements. The JTRS-preferred approach is to provide a mechanism whereby standard building blocks are developed from which APIs may be created. These building blocks are conceptual elements intended to provide the API developer with a template that can be used to form IDL interfaces. APIs for each specific waveform are developed from the same building blocks to provide a standard set for each of those waveforms.

Future waveforms will have their APIs developed from the same standard building blocks. If a new service, not provided by the building blocks, is needed for a waveform, the building blocks can be extended to provide that service. Available services not required by a waveform would not become part of the APIs for that waveform, thus providing a more efficient API.

API Building Blocks are pre-defined analysis classes that define the various components that make up an API. Two types of Building Blocks are used. The first type, concrete Building Blocks, applies to all APIs and is well defined, needing only implementation for the specific API. The second type, generic Building Blocks, provide an analysis class in which each API defines the generic interface details making the API specific to that application. Generic Building Blocks are provided to facilitate an API framework for interfaces that are, by nature, application dependent. A Building Block is an abstract element that must be instantiated with concrete types to make a usable interface. Building blocks are used to define abstract services to foster reuse and commonality between differently implemented APIs. Each building block is then instantiated to form an interface. All of the interfaces at one defined interface are combined to form the API for that interface.

A Building Block is represented in UML by a parameterized class. An instantiated Building Block defines a callable Interface and is represented in UML with an INTERFACE stereotype.

This interface, however, may not define all of the data flowing into and out of a particular layer. The collection of interfaces at a particular layer forms an API.

The interfaces associated with some layers can be commonly defined. For example, the LLC interface, when used by current waveforms, is completely defined in an API. Likewise, completely defined I/O APIs are common to waveform applications used in a JTRS. Other interface definitions are dependent on the waveforms that are being used. For this reason, Building Blocks are generated which define the type of services that are needed but not the actual parameters for that service. When the Building Block is used to form an API, the Building Block parameters are tailored to the waveform being supported.

Thus, Building Blocks provide a basis for specific interfaces that is invariant with respect to waveform, platform, installation, etc. This encourages commonality between realizations of radios with varying discriminating characteristics. It restricts the architect of a new radio function to a certain template form for attributes and operations for the service group it defines.

Building Blocks may exist on either one layer (e.g., Physical, MAC, Link, and I/O) or may be common to multiple layers, like the Packet BB.

IDL is not generated for the generic Building Blocks since IDL cannot be generated from parameterized classes. The IDL for an interface is developed based upon Building Block model.

4 REQUIREMENTS.

4.1 GENERAL REQUIREMENTS.

4.1.1 Name Scoping.

APIs and building blocks should use scoped names. For example, the API set for LOS may have the following module hierarchy:

```
module LOS {  
    module Physical {  
    }  
    MODULE Mac {  
    }  
}
```

4.2 API REQUIREMENTS.

SCA-compliant APIs are defined in Service Definitions consisting of behavior, state, priority, and additional information that provide the contract between the Service Provider and the Service User. All SCA-compliant APIs shall have their interfaces described in IDL. IDL is used to foster reuse and interoperability. IDL provides a method to inherit from multiple interfaces to form a new API.

4.2.1 API Usage and Creation.

The structure and language requirements of the APIs have been selected to provide commonality between implementations to foster reuse and portability of applications. To further these ends, one of the following methods for creating APIs shall be used.

- A. Use existing API.
- B. Create a new API by inheriting an existing API and then extending its services.
- C. Translate an existing non-JTRS API to IDL to create a new JTRS API.
- D. Develop a new API based upon one or more Building Blocks. Use of Building Blocks should follow the order of using existing Building Blocks, extending existing Building Blocks, generating new Building Blocks.

For these identified methods, the order of preference shall flow from Item A to Item D.

4.2.2 API Transfer Mechanisms.

A Transfer Mechanism provides the communication mechanism between a service provider and a service user that may be co-located or distributed across different processors. Figure 4-1 shows the standard and alternate transfer mechanism structure for APIs.

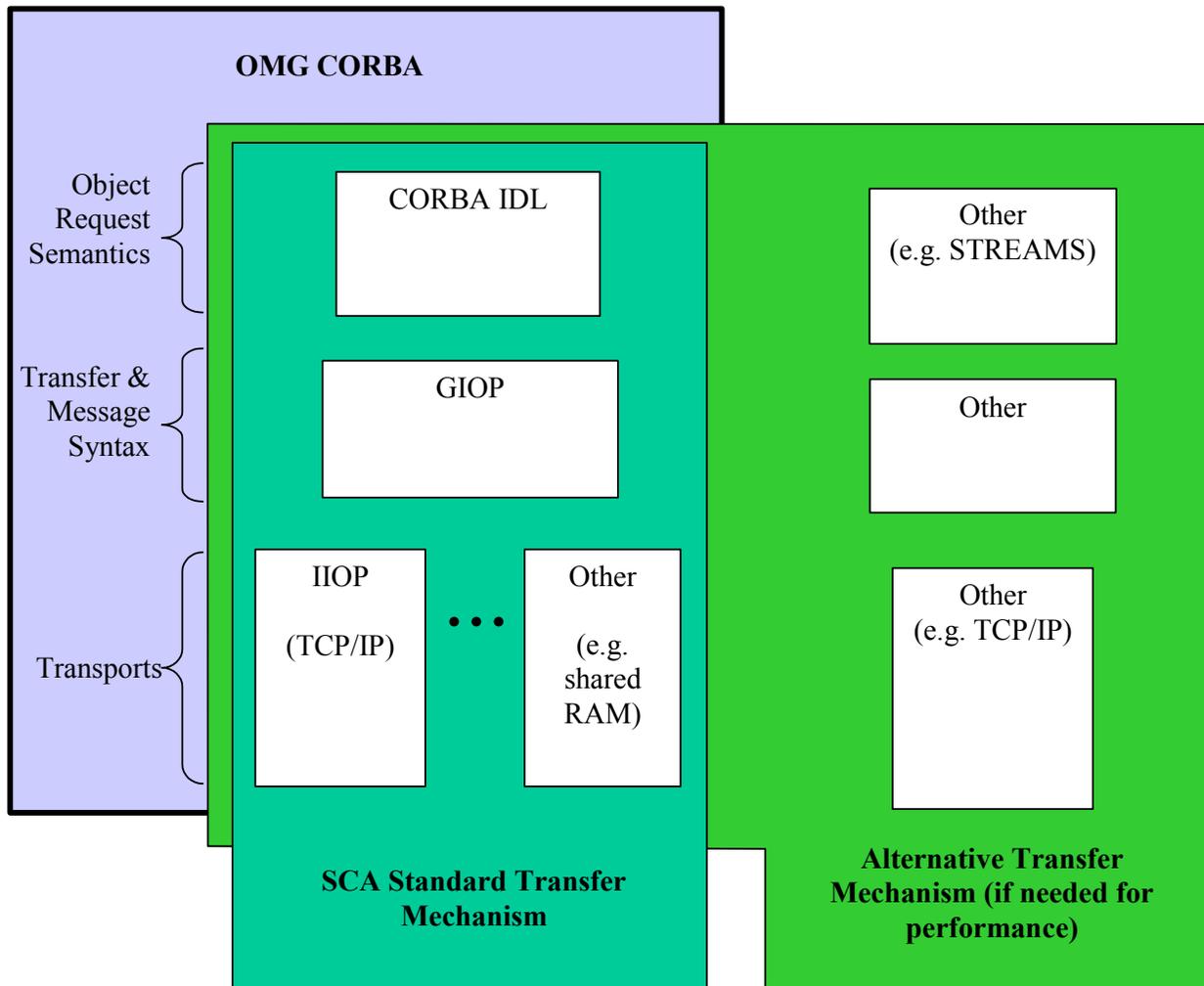


Figure 4-1. Standard and Alternate Transfer Mechanism

4.2.2.1 Standard Transfer Mechanism.

The standard transfer mechanism shall be CORBA except as allowed in 4.2.2.2.

4.2.2.2 Alternate Transfer Mechanism.

An alternate transfer mechanism is allowed when the Application performance requirements cannot be achieved with the standard transfer mechanism.

When an alternate transfer mechanism is used for real-time control and data flow, the transfer mechanism for initialization and non-real-time control shall use the standard transfer mechanism (if those controls can be separated).

When an alternate transfer mechanism is used, the transfer and message syntax of the alternate transfer mechanism shall be mapped to the IDL of the API Service Definition. This mapping shall be identified by a UUID (separate from the Service Definition UUID).

The description of the alternate transfer mechanism, an analysis supporting the performance need for the alternate mechanism, the mappings to the Service Definition, and the associated UUIDs shall be registered as defined in section 4.3.3.

4.2.2.2.1 API Instance Behavior.

Irrespective of the transfer mechanism used, all behavior including state transitions and priorities defined in the service definition shall be obeyed by an API Instance.

4.2.2.2.2 Alternate Transfer Mechanism Standards.

Transfer mechanisms shall be in accordance with commercial or government standards.

4.2.2.2.3 Alternate Transfer Mechanism Selection.

In addition to the above, transfer mechanism selection should consider the availability of supporting products that have wide usage, are available from multiple vendors, and are expected to have long-term support in the industry.

4.3 SERVICE DEFINITION REQUIREMENTS.

4.3.1 Non-JTRS Service Definitions.

It is not the intent of this document to force creation of new documentation for non-JTRS APIs that have commercial and/or government acceptance. Non-JTRS APIs that do not have IDL interfaces shall have a mapping to an IDL interface in a Service Definition as shown in Figure 4-2.

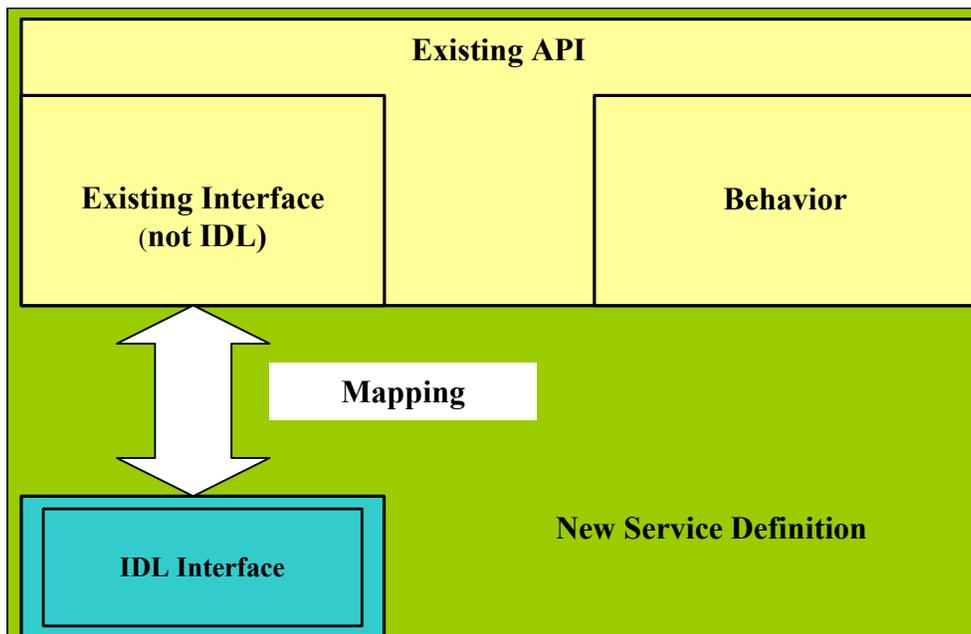


Figure 4-2. Reusing an Existing API Without an IDL Interface

4.3.2 JTRS API Service Definition Format.

SCA-compliant APIs shall be defined in Service Definitions conforming to the Service Definition Description (SDD) provided in Appendix A, except as allowed in 4.3.1.

SCA-compliant Service Definitions shall conform to the Service Definition Description (SDD) provided in Appendix A, except as allowed in 4.3.1.

4.3.3 Service Definition Identification and Registration.

Service Definition documentation of SCA-compliant APIs shall be submitted to a Registration Body to be established, initially, by the JTRS JPO.

Each Service Definition shall be identified by a Universally Unique Identifier (UUID). As used in this specification, the UUID is defined by the DCE UUID standard (adopted by CORBA). No centralized authority is required to administer UUIDs (beyond the one that allocates IEEE 802.1 node identifier MAC address).

Note: The UUID described in this requirement is needed for API registration purposes. It is not used for interface compatibility during run time. *{When a registration body is established for JTRS APIs, the identification methodology may be revised to be more user-friendly, under that body's control.}*

5 ACRONYMS, ABBREVIATIONS, AND OTHER DEFINITIONS.

5.1 ACRONYMS AND ABBREVIATIONS.

AM	Amplitude Modulation
API	Application Program Interface
BB	Building Block
CORBA	Common Object Request Broker Architecture
CSMA	Carrier Sense Multiple Access
DAMA	Demand Assigned Multiple Access
DASA	Demand Assigned Single Access
dBm	Decibels relative to one milliWatt
DCE	Distributed Computing Environment
FM	Frequency Modulation
FSK	Frequency Shift Keying
GUI	Graphical User Interface
HCI	Human-Computer Interface
HF ALE	High Frequency Automatic Link Establishment
HMI	Human-Machine Interface
Hz	Hertz
ID	Identifier, Identification
IDL	Interface Definition Language
I/O	Input and Output
ISO	International Standards Organization
JPO	Joint Program Office
JTRS	Joint Tactical Radio System
LOS	Line of Sight
MAC	Media Access Control
MSRC	Modular Software-Programmable Radio Consortium
NRT	Non-real-time
PDU	Protocol Data Unit
PPDU	Packet Protocol Data Unit
PSK	Phase Shift Keying
QoS, QOS	Quality of Service
RF	Radio Frequency
RT	Real-time
Rx	Receive
SCA	Software Communication Architecture
SD	Service Definition
SDD	Service Definition Description
SINCGARS	Single Channel Ground and Airborne Radio System
SW	Software
TDMA	Time Division Multiple Access

TM	Transfer Mechanism
TOD	Time of Day
TRANSEC	Transmission Security
TU	Transmission Unit
Tx	Transmit
UML	Unified Modeling Language
UUID	Universal Unique Identifier
WDW	Wideband Data Waveform
WF	Waveform
WOD	Word of Day

5.2 DEFINITIONS

The following definitions are in addition to those in Section 1.2.

Abstract Class

A class with specification only, and no implementation.

Concrete Class

A class whose implementation is complete and thus may be realized by an object.

Instantiation

A type-specific interface created from a parameterized class.

Non-Real-Time

Independent of user data transfer.

Packet

Logical grouping of information that includes a header containing control information and (usually) user data.

Parameterized Class

A class that serves as a template for other classes. A parameterized class must be instantiated before instances can be created.

Payload

Portion of a cell, frame, or packet that contains user information (data).

Priority Queue

Routing feature in which frames in an interface output queue are prioritized based on various characteristics such as packet size and interface type.

Real-Time

In conjunction with user data transfer.

Service Provider

An entity (hardware or software) at one layer of an interface that provides services to entities at a different layer of a system.

Service User

An entity (hardware or software) at one layer of an interface that uses the services at a lower layer of a system.