

# JOINT TACTICAL RADIO SYSTEM - APPLICATION PROGRAMMING INTERFACES

Cinly Magsombol, Chalena Jimenez, Donald R. Stephens  
Joint Program Executive Office, Joint Tactical Radio Systems Standards  
San Diego, CA

## ABSTRACT

In April 2007, the Joint Program Executive Office (JPEO) Joint Tactical Radio System (JTRS) released 15 Application Programming Interfaces (APIs) to the Software Communications Architecture (SCA) website [1]. The release of these APIs will allow for the public distribution, implementation, and collaboration with commercial software vendors, universities, and international programs. The APIs contained in this first release provide common API building utilities, I/O interfaces supporting data/voice processing, and an abstraction for the modem hardware. This release is the result of efforts to promote and encourage portability and reusability within the software defined radio community.

## INTRODUCTION

The standardization of APIs is essential to the overall JPEO JTRS goals of software portability and reusability. APIs facilitate application portability and reuse by providing a defined and documented host environment across all JTR platforms. JTRS APIs specify the language/semantics that guarantee that a waveform application and service on the set can communicate regardless of the implementation details (i.e., different languages or operating environment) of the JTR component. This allows applications developed on one platform to be reused on another platform of different size, mission, or deployment. A common API also delineates the components responsibility between the waveform and platform minimizing any duplication of effort.

The JTR infrastructure reuse model is shown in Figure 1. It is expected that all JTRS products available from the JPEO Information Repository (IR) (waveform and operating environment software) utilize these defined APIs to reduce costs, discovery, code rework, and improve product deliveries.

Portability and reuse are heavily dependent on the proper specification of the interface. The development of an interface that is too set specific or incompatible with the existing code base would require additional waveform modifications when porting. Initial API standardization involved balancing the need to provide

APIs that were backward compatible with the existing code base while ensuring the APIs were scalable and extensible to the various form factors and missions.

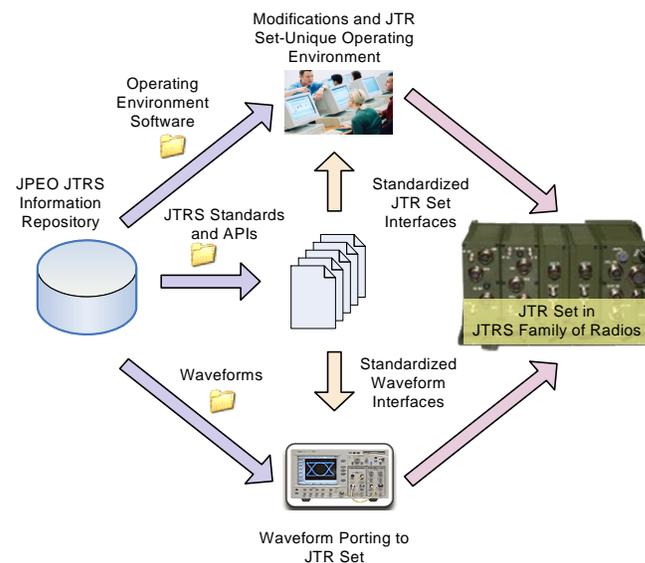


Figure 1 JTR Infrastructure Reuse Model

## JTRS API DEFINITION

The first JTRS APIs evolved from the software code base that was available from the JPEO IR. Early examination of this code base identified candidate APIs that were common across waveform developments. Other available interfaces were JTRS set specific and were not selected for standardization. The decision was to leverage this knowledge base and specify APIs that were backwards compatible to these products.

JTRS Standards acknowledged a need for specifying flexible APIs that would accommodate varying form factors and missions. Specifically, it was desired to provide a rich set of services for larger platforms while not burdening smaller resource constrained platforms. Toward this goal, JTRS Standards has defined the concept of API extensions. API extensions are one of many design patterns developed for the APIs defined in “Design Patterns of JTRS Infrastructure” [2]. It categorizes the specific capabilities within a specified API as a “base” or an “extension”. The API “base” specifies the minimum behavior needed for all JTRS platforms while API extensions identify additional (or optional) capabilities that can be used by larger

platforms. For example, the *Vocoder Service* provides a “base” capability for the control and transfer encoded data. It also defines “extension” capabilities to support different vocoders (e.g. MELP, CVSD, LPC). Platforms requiring a *Vocoder Service* would implement the base only those extensions required.

### JTRS API STRUCTURE

API documents provide a complete definition of the waveform to platform Interface Description Language (IDL) for the specified service (e.g. JTR set application component). JTR platform specific interfaces have not been defined and will be part of the service specific design. The service definition delineates what is to be provided by the service from the service user (e.g. waveform) and describes all service interfaces, methods, input and output parameters, data types, and error signals. Where applicable, sequence diagrams have been documented and included to clarify certain call behaviors.

To minimize the number of documents, base APIs and extensions are collocated within the same document. Base APIs are defined in section ‘A’ with API extensions following with section ‘B’ and beyond. There can be zero or more extensions for each API.

### JTRS API MANAGEMENT

The JPEO recognized the importance of involving the JTRS community and utilizing developed knowledge in the creation of JTR APIs. They also understood that new APIs must be developed and be capable of evolving as new technologies and missions emerge. To foster and manage APIs, the JPEO created the JTRS Standards Interface Control Working Group (ICWG) to serve as the technical and decision-making authority on the development and configuration management of all JTRS standards including APIs. Members of the ICWG include participants from across the JTRS enterprise. ICWG activities include the approval of all individual standards and the disposition of change requests on existing standards. It is a tenet of the ICWG to ensure that all standards are based upon mature use cases, follow defined processes, and consider the impact of these standards on the entire JTRS community.

### APIS WITHIN THE JTRS INFRASTRUCTURE

APIs are a subset of the specifications and standards that make up the entire JTRS infrastructure. Other public specifications include the *Software Communication Architecture* [3]. The collection of the JTRS APIs deployed on the JTRS infrastructure is illustrated in Figure 2. JTRS APIs fall into broad

categories consisting of primitive APIs, radio devices, and radio services.

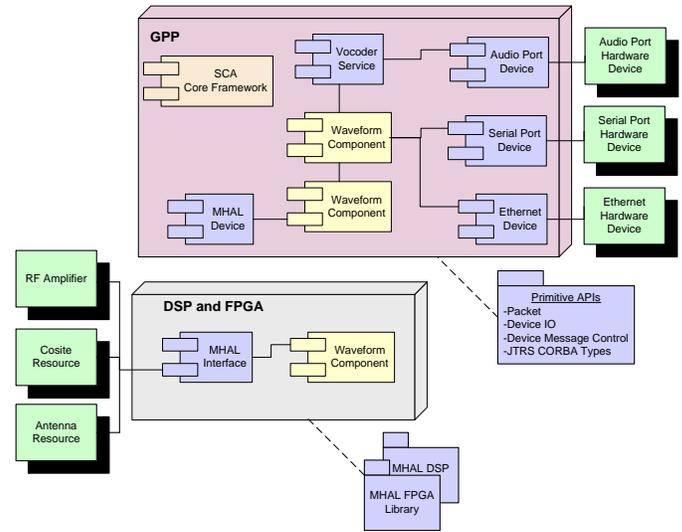


Figure 2 JTRS Infrastructure

Primitive APIs are a set of basic data types and data transfer interfaces that are common to a number of APIs. APIs requiring these utilities will use/inherit these types/interfaces in their design. This promotes reuse and design commonality between the different APIs. Primitive APIs are documented independently to avoid redefining their functionality in each API document. Figure 3 provides an example use of a primitive API. The *VocoderPacketProducer* and *VocoderPacketConsumer* interfaces (from *Vocoder Service API*) inherit the primitive *Packet API* interfaces for pushing octet packets. The *Vocoder Services* does not need to redefine these interfaces.

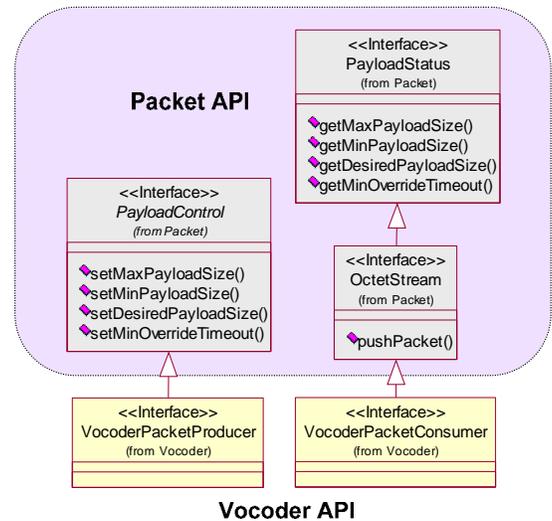


Figure 3 Vocoder Service Primitive Interfaces

A generic radio device is depicted in Figure 4. Radio devices provide a software abstraction for physical hardware devices. Radio devices expose a JTRS standard CORBA-based API to the user (e.g. waveform component) and abstract the hardware specific details from the waveform. The native device platform-side API is defined by the set for its particular architecture and mission.

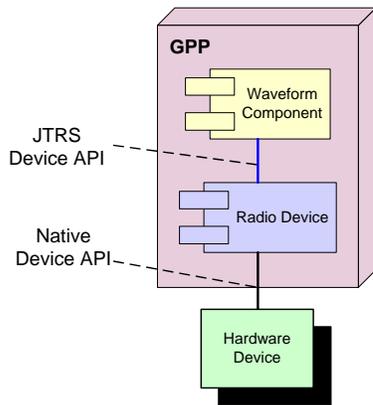


Figure 4 Radio Devices

Radio services provide functionality that is common to a number of software components. Unlike radio devices, they are not tied to specific hardware resources on the set.

### JTRS PRIMITIVE APIS

#### JTRS CORBA Types

*JTRS CORBA Types* provide a set of common JTRS types that are defined in a common JTRS product wide namespace. It is the intention that *JTRS CORBA Types* be used by JTRS APIs. The standard provides a collocated set of the common unbounded sequence types that are equivalent to the superset found in the SCA “Cf.idl” and “PortTypes.idl” files. This set does not include floating point types since it is recommended that they should be limited in JTRS APIs and avoided on resource restricted platforms. *JTRS CORBA Types* also define the JTRS extensible enumeration type which was defined to allow for the extensibility of additional enumerated values [2]. Lastly, this set includes common error exceptions related to the inclusion of the extensible enumeration types within an operation.

#### Packet API

The JPEO has standardized two sets of packet interfaces. The *Packet API* was designed as the preferred extensible packet interface to be used in future APIs. The older packet interfaces listed below are provided to maintain compatibility with legacy devices such as the *Ethernet Device*. These interfaces provide

similar but less extensible capabilities and are not recommended for future designs. The older packet interfaces will be maintained until they are no longer required by legacy devices.

- § Device Packet
- § Device Packet Signals
- § Device Simple Packet
- § Device Simple Packet Signals

The *Packet API* defines the basic building blocks for component messaging. It defines a common set of data transfer and control interfaces that can be used by component JTRS APIs such as the *Audio Port Device*. The majority of inter-component data communications are accomplished using these messaging interfaces.

Packet interfaces are based on the data producer and consumer model shown in Figure 5. As illustrated, data is sourced at the data producer and consumed at the data consumer. A component can be a data producer or a data consumer depending whether it is producing or consuming data.

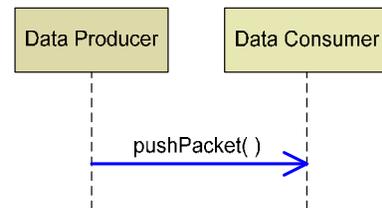


Figure 5 Data Producer and Consumer

The *Packet API* base provides methods to push packets of common data types to the device or service user. Packets are accompanied by a structure that identifies and sequences the streams provided. The base also provides other methods to negotiate packet payload sizes and delivery timeouts provided in Figure 6.

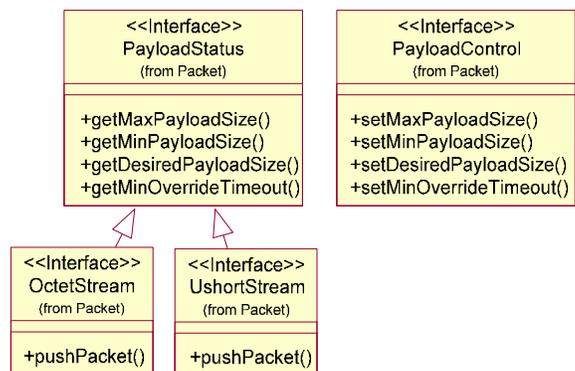


Figure 6 Base Packet Interfaces

The *Flow Control* extension and *Empty Signals* extensions provide added mechanisms to control and monitor the data stream defined in the base. These interfaces may be used in conjunction with other interfaces to create the device/service data producer and data consumer interfaces. The *Flow Control Extension* provides the ability to send flow controlled data to/from the device or service user. Data is controlled by either polling for space availability or waiting for a signal to resume. The *Empty Signal Extension* provides the ability to indicate when a specified stream has been processed.

### DeviceIo API

As with the *Packet API*, the *DeviceIo API* is also accompanied by older interfaces. *DeviceIo* contains a repackaging of the standards below to provide a common namespace and reduce the number of supporting documents. The interfaces below are provided to maintain compatibility and are not recommended for future designs. The older interfaces will be maintained until they are no longer required by legacy devices.

- § Device IO Signals
- § Device IO Control

The *DeviceIo* interfaces provide methods to enable a handshake before sending data. Specifically, these interfaces activate Request to Send (RTS) and Clear to Send (CTS) signals. The *DeviceIo* interfaces may be used in association with the *Packet API* interfaces to create the component data producer and data consumer interfaces.

### Device Message Control API

*Device Message Control* is the last of the common utilities released in this first set of APIs. This interface provides methods to monitor the communication in the transmit or receive direction. It also provides the ability to abort the transmission.

## JTRS DEVICES

### Audio Port Device

The *AudioPort Device* is a radio device that supports methods and attributes that are specific to the audio hardware it represents. The base *AudioPort Device* provided in Figure 7 provides operations to generate and control alert/alarm tones within an audio stream and to notify the device user of a Push-To-Talk (PTT) event.

The *AudioSampleStream* extension extends the functionality of the base to provide the ability to

consume and control audio samples to/from the audio hardware. The *AudioSampleStream* audio stream capability is accomplished by inheriting from the packet interfaces described in the *Packet API*.

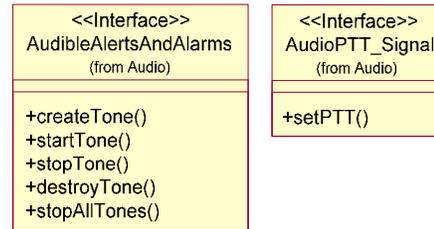


Figure 7 Base Audio Port Device

### Serial Port Device

The *Serial Port Device* is another radio device which supports the operation and configuration of serial port hardware devices. The *Serial Port Device* base provides the ability to generate and consume serial packets to/from the serial port device. The *Serial Port Device* base also provides a base configuration interface to select the specific synchronization mode to exercise, shown in Figure 8.

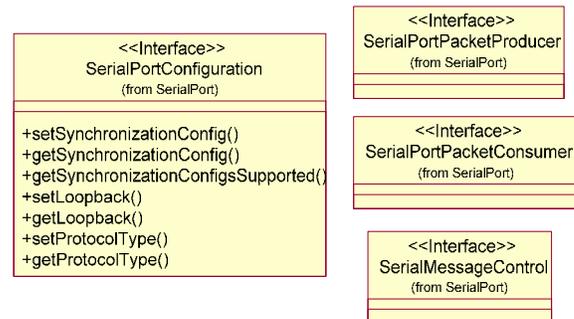


Figure 8 Base Serial Port Device

The *Asynchronous*, *Synchronous Raw* and *Synchronous HDLC* extensions extend the functionality of the base to support the configuration of each synchronization configuration type. Each extension provides a configuration interface that allows the user to set and retrieve the configuration for the specified synchronization configuration type. Figure 9 provides an example of the *Synchronous Raw* extension.

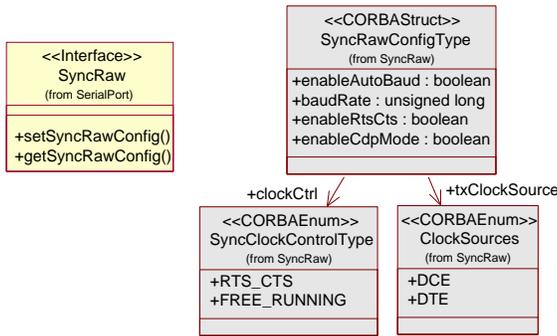


Figure 9 Synchronous Raw Extension

### Ethernet Device

The *Ethernet Device* abstracts the Ethernet hardware from the device user. It consists solely of a base which provides operations to generate and consume ethernet packets to and from the ethernet hardware. As with other devices, data transfer is achieved through inheritance of the common packet interfaces. However, unlike the other APIs, the *Ethernet Device* relies on the older *Packet* and *DeviceIo* interfaces.

### MHAL Device

The *MHAL* device abstracts the waveform application from the specifics of the JTR modem hardware. It also supports communications between waveform components hosted on different computational elements (CEs) (e.g. General Purpose Processors (GPPs), Modem Digital Signal Processors (DSPs) and/or Modem Field Programmable Gate Arrays (FPGAs)). Figure 10 shows a reference deployment of the *MHAL*. The *MHAL API* does not specify the number of computational elements a JTR platform will provide. It also does not specify the platform specific transport, implementation or hardware architecture. What the *MHAL* does specify is a defined and consistent method for routing messages between each CE. It achieves this through the definition *MHAL* interfaces and structures.

**Error! Objects cannot be created from editing field codes.**  
Figure 10 MHAL API Reference Deployment Diagram

The *MHAL* base defines the common *MHAL* message structure depicted in Figure 11. This message structure includes the information required to maintain an orderly processing of message buffers across CEs. Specifically, the *MHAL* structure specifies an In-Use (IU) bit for internal message flow control, the logical destination (e.g. end user) for the message, the payload to be sent, and the message length. It is important to note that the payload is an opaque message that is passed by the JTR

set's transport mechanism to the peer *MHAL* communication node.

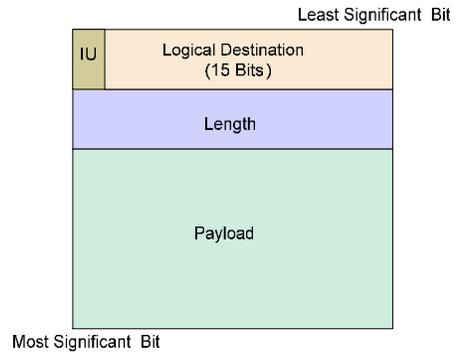


Figure 11 Standard Message Structure for MHAL Communication

The *MHAL GPP* extension is a CORBA-based device interface documented in IDL. It provides a defined and consistent interface to the waveform for access to one of the other CEs. The CORBA API for the *MHAL* is illustrated in Figure 12. As with the other APIs, data transfer is based on the packet interfaces and defines a *pushPacket* operation as well as operations for setting and getting message routes. CORBA-capable processors may utilize this interface for sending *MHAL* messages between processing elements.

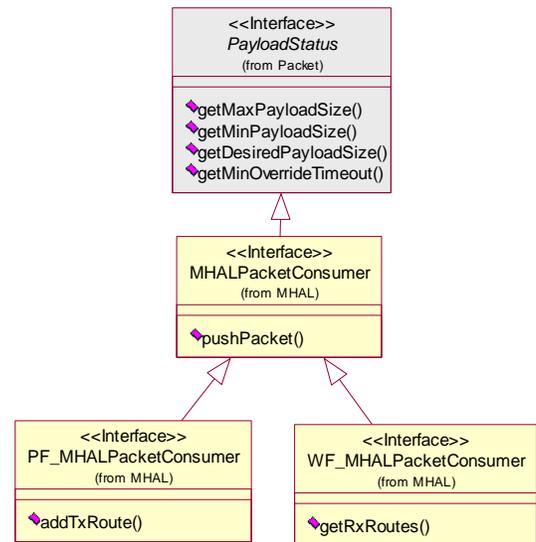


Figure 12 CORBA MHAL API

The concept for defining an infrastructure on the DSP and FPGA processors was similar to that of the *MHAL GPP*. Because the DSP environment does not readily support dynamic linkable objects, the *MHAL* interface on the DSP is a library of standardized components that are linked into the waveform code at build time. The API for the DSP extension is illustrated

in Figure 13. *MHAL DSP* messages are serviced through the function *MHAL\_Comm()*. This is equivalent to *pushPacket()* defined in the *GPP MHAL* extension. This C-level interface defines functions that communicate with other processor elements using the standardized *MHAL* messages.

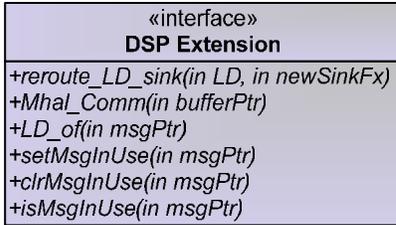


Figure 13 MHAL DSP Extension

The *MHAL FPGA* extension consists of FPGA entity libraries that are linked into a waveform build. In the FPGA interfaces, a receive node contains a signal interface that is asserted when a complete message is received. The JTR set's interfaces to the FPGA nodes are set-specific. The FPGA entities illustrated in Figure 14 are provided by the JTR set developer and compiled into the waveform's FPGA bit image. As with traditional APIs, not all waveforms will utilize all of the entities shown and the bit image will contain only the FPGA node resources required.

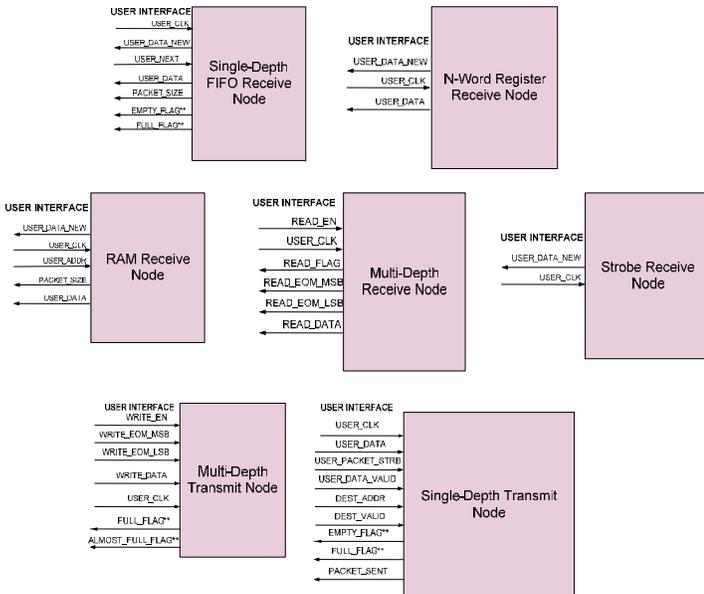


Figure 14 MHAL FPGA Extension

The *MHAL* APIs defined in the previous paragraphs allow detailed programming of the modem and RF capabilities using basic *MHAL* messages. To provide a higher-level abstraction of JTR set capabilities, the *MHAL Chain Coordinator* extension API is defined with

common functions expected of modems and power amplifiers. In Figure 15, the *Chain Coordinator* API provides functions such as channel frequency (*RFC\_ChannelFrequency*) and receiver automatic gain control (AGC) that control (*RFC\_RxAGCAttackTime*) which increases programming abstraction and subsequently, portability of waveform software.

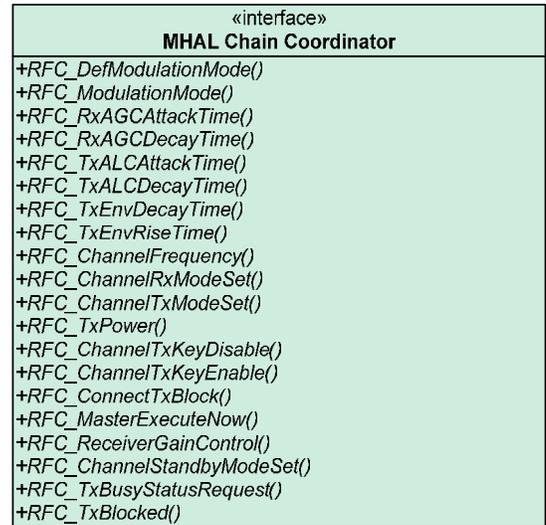


Figure 15 MHAL Chain Coordinator Extension

## JTRS SERVICES

### Vocoder Service

The *Vocoder Service* is the single radio service defined in this first release. It provides common voice encoding/decoding capabilities to application components. As depicted in Figure 16, the *Vocoder Service* base defines the transfer of raw bit streams between the application and the service. It also provides operations to control the bit stream. Specifically, this control supports loop back operations, provides for the query and the selection specific algorithms, and provides an option to abort the transmit stream.

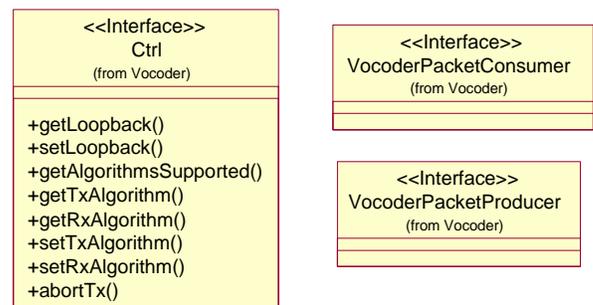


Figure 16 Base Vocoder Service

The *Vocoder Service* is extended for the specific algorithms types used to encode/decode the voice stream. The *Vocoder Service* currently supports Continuously Variable Slope Delta (CVSD), Mixed Excitation Linear Predictive (MELP), Linear Predictive Coding (LPC), and Speex, which is based upon CELP (Code Excited Linear Prediction). The individual algorithms are defined as extensions and provide the capability to query and configure the algorithm selected. Similar to the *Serial Port Device* synchronization configuration type extensions, the *Vocoder Service* algorithm type extensions provide methods to configure and get the current configuration. The interfaces and associated configuration structure for the *MELP* extension is shown in Figure 17. It should be noted that the *Speex* extension does not have a defined interface since it does not require specialized configuration services.

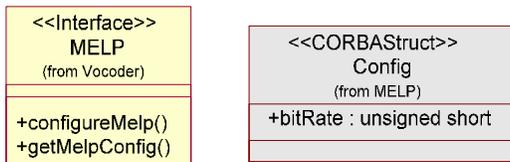


Figure 17 MELP Extension

The last extension in the *Vocoder Service* is the *Vocoder Audio Stream* extension. This extension provides an explicit connection of the service to the *Audio Port Device* to transfer audio samples as illustrated in Figure 18. The standard also allows for a JTR set defined connection to the audio port device. This is permitted to support legacy implementations which may have coupled the *Vocoder Service* and *Audio Port Device* in their design.

**Error! Objects cannot be created from editing field codes.**  
 Figure 18 Explicit Connections to Audio Port Device

### EXAMPLE API COLLABORATION DIAGRAM

Figure 19 illustrates an example interaction between a radio service or device and its user, using the defined interfaces and operations. In this example, the *Vocoder Service* provides encoded data to the *Service User* (i.e. waveform). The *Service User* selects CVSD as the transmit algorithm by calling the *setTxAlgorithm* operation defined in the *Ctrl* interface. When data is available (from the data source), the *Vocoder Service* will encode that data using the CVSD algorithm and pass it to the *Service User* by calling the *pushPacket* method defined in the *VocoderPacketConsumer* interface.

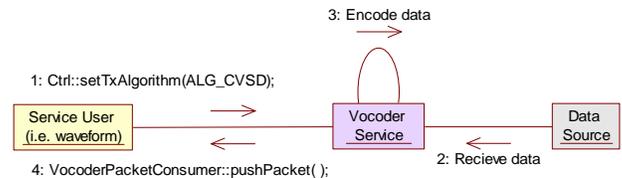


Figure 19 Vocoder Service Collaboration Diagram

### SUMMARY

The development of JTRS APIs was a collaborative effort based on design methodologies to allow the portability of application software and interchangeability of platform services. The current release consists of 5 component APIs and a number of primitive APIs. This paper provided a summary of each API included in this release, reviewed their origins, and discussed their importance to JTRS. It is expected that additional APIs will be released in the future to promote portability and reuse.

### REFERENCES

- [1] Software Communications Architecture Website, <http://jtrs.spawar.navy.mil/sca>
- [2] Stephens, D. R., "Design Patterns of JTRS Infrastructure", MILCOM 2007.
- [3] JTRS Standards, "Software Communication Architecture", Version 2.2.2., May 15, 2006.