Communications Research Centre Canada
An Agency of Industry Canada

Centre de recherches sur les communications Canada
Un organisme d'Industrie Canada

Software Communications Architecture v2.2 Reference Implementation Project

SDRF Forum Contract SDRF-04-A-0002-V0.00

Deliverable 3

Analog FM Application Developer Guide

Presented to:

Software Defined Radio Forum
Denver, Colorado
USA

Prepared by:

Communications Research Centre Canada
Advanced Radio System Research Group
Ottawa, Ontario

1 February 2005

Document Prepared By	M. Phisel, CRC / VPSAT / RARS
Approved By	C. Bélisle Research Management Authority Research Manager CRC / VPSAT / RARS
Submitted To	Software Defined Radio Forum
Date	February 1, 2005

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This document is part of the third deliverable of the SCAv2.2 reference implementation contract issued by the SDR Forum to the Communications Research Centre Canada.

In this document, the software design of the Analog FM application developed as part of this contract is described.

This document is a companion to the "AnalogFM Application User Guide.pdf" also provided at www.crc.ca/scari-open

2. APPLICATION DEVELOPMENT LIFE-CYCLE

This section contains a description of the Life-Cycle development of an application. As shown in Figure 1, the different steps of creating and application are being presented.

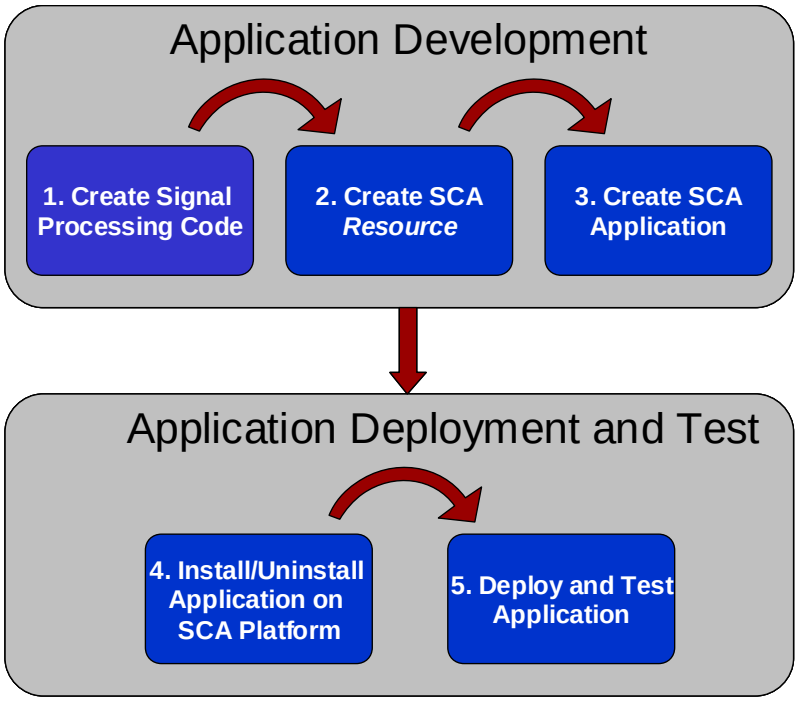


Figure 1 – Application Development Life-Cycle.

2.1. Create Signal Processing Code

In the first step, the developer creates the software to perform signal processing (ex: an FM Modulator). That software must be implemented in a language supported by a compiler for the target processor. It can be developed in tools like MATLAB®.

2.2. Create an SCA Resource

In the second step, the signal processing code of the previous step is intergraded into a *Resource* as shown in Figure 2. The software can also be compiled for a number of different target platforms. When deployed, the Core Framework will automatically choose the proper implementation.

The *Resource* must be CORBA-enabled and implement standard interfaces (specified in CORBA IDL): PortSupplier, LifeCycle, PropertySet, TestableObject, and Resource. The *Resource* can be a proxy to non-CORBA native code (DSP code). Each component must be described with meta-data used for deployment (domain profile database).

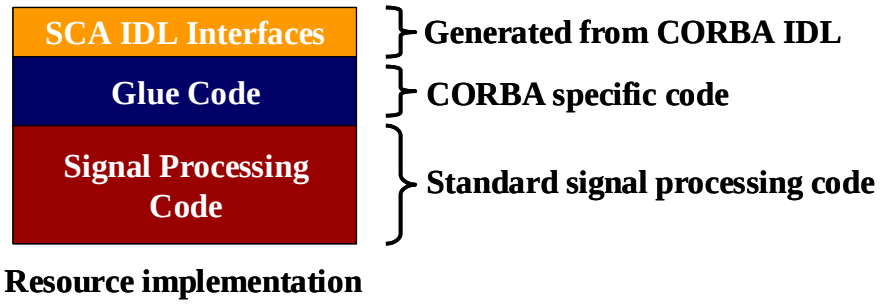


Figure 2 – SCA Resource Implementation

The meta-data for SCA components must be provided using 3 types of SCA XML files as shown in Figure 3:

1. Software Package Descriptor (SPD):
   - Describes component implementations
   - Describes capacity and capability requirements
2. Software Component Descriptor (SCD):
   - Describes CORBA interfaces
   - Describes ports for interconnections
3. Property Descriptor (PRF):
   - Describes execution parameters
   - Describes configuration properties

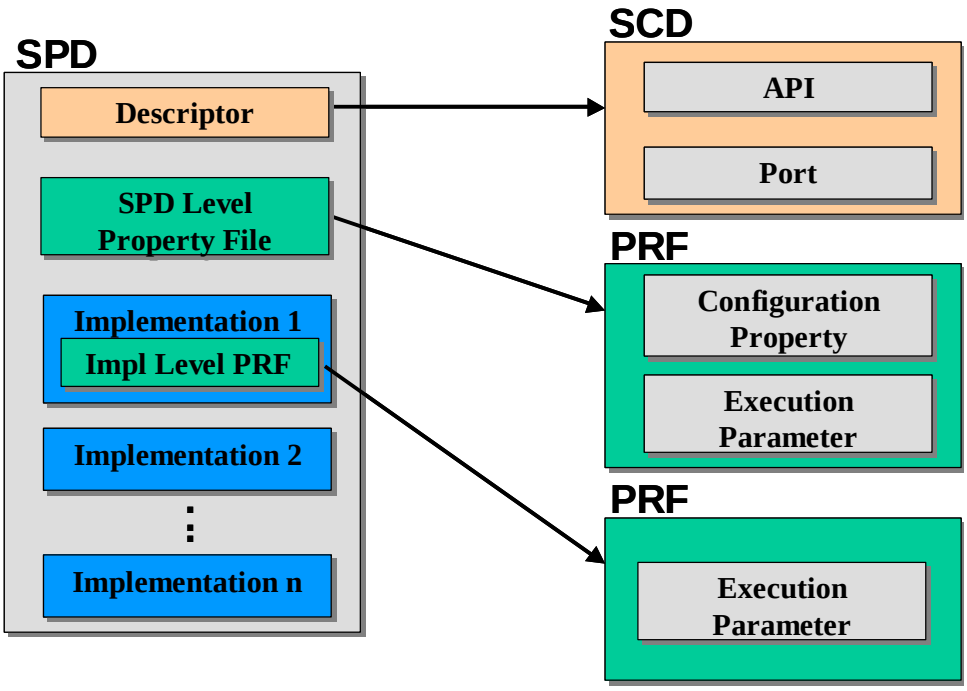
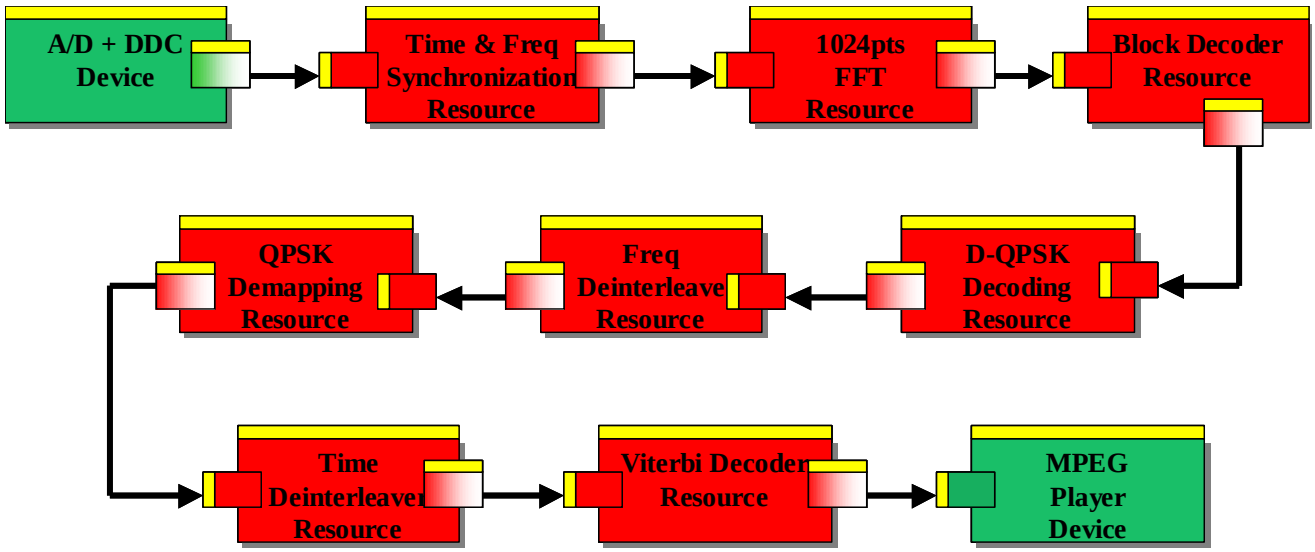


Figure 3 – SCA Resource XML Files.

2.3. Create an SCA Application

In the third step, an SCA Application is created. It is composed of a number of *Resources* exchanging data using Ports. Figure 4 is showing an example on how different *Resources* (in red) are link together.



CRC's Digital Audio Broadcast™ (DAB) SCA Application (Simplified block diagram)

Figure 4 – SCA Application.

For an application to be deployed/executed on an SCA radio, it must be described using meta-data. The description is done in the XML file Software Assembly Descriptor (SAD) (see Figure 5) which:

- Identifies each component of an application
- Specifies how many instantiations of each component must be created
- Specifies any co-location restrictions for groups of instantiations
- Specifies which component is the assembly controller
- Specifies how the connections between component instantiations must be established

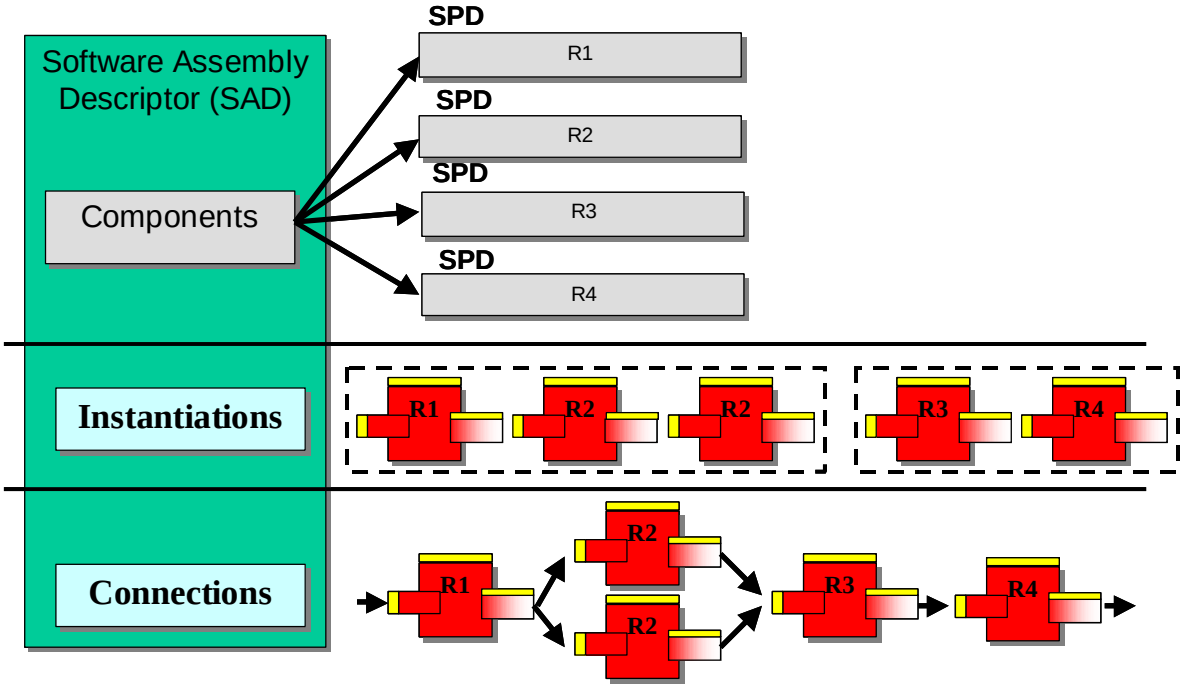


Figure 5 – Software Assembly Descriptor (SAD)

To be installed, an application can be packaged into a single file for easy transportation

- Contains domain profile descriptors for each component and the assembly
- Contains binary code for Resources implementations and libraries

2.4. Install/Uninstall Application on SCA Platform

In the fourth step, the SCA Application is installed which is done as follows:

- Copying the following Application files on the SCA radio:
  - One Software Assembly Descriptor (SAD)
  - n Software Package Descriptor (SPD)
  - m Software Component Descriptor (SCD)
  - k Property Files (PRF)
  - x binary files (compiled code and shared libraries)
- Finalizing the application installation by informing the Radio Management

2.5. Deploy and Test an SCA Application

In the fifth and final step, the SCA Application is deployed and consists in:

- Reading the application assembly meta-data (i.e. SAD)
  Identifying components to be deployed
- Selecting Devices of the SCA radio for each application component
- Making capacity reservations against selected devices based on component requirements
- Deploying code files onto selected devices
- Initializing and configuring deployed components
- Establishing connections between deployed components

3. ANALOG FM APPLICATION

The goal of the Analog FM application is to communicate with radios no mater which algorithm of communication they're using. The Analog FM can thus be a radio bridge or quite simply a computer radio. The radio bridge decodes the mode of communication of a radio. Then, encode its data into another mode of communication for another radio. The computer radio can be in decoding mode or encoding. It decodes the received signal of a radio to listen to it on the speakers or it encodes the signal of the microphone to transmit it to the other radio.

In this demonstration application, the transform *Resources*, used for the FM waveform, along with the assembly controller have been implemented. Also, the XML domain profile files describing the waveform application have been created.

This section contains a description of the Analog FM application. Section 3.1 illustrates the radio and application configuration. Section 3.2 explains how to boot the radio and run the application. Section 3.3 and section 3.4 describes the components needed by the Analog FM Application which performs FM Modulation or FM Demodulation on an audio stream that were developed for this application by extending the SCARI-Open core framework.

3.1. Application Deployment on a Single-Node Radio

Figure 6 shows the radio used in this demonstration that is composed of one node (personal computer with a full-duplex sound board). On this node are running the *DomainManager* and a *DeviceManager* which reports to the *DomainManager*. The node is also running an *ExecutableDevice*, an *AudioDevice*, an *RFDevice* and a *Log*.

As its name indicates, the ModulationFM *Resource* modulates its input audio stream into an FM signal. The DemodulationFM *Resource* demodulates its input FM audio stream into an audio stream. The FM Modulation and FM Demodulation processing are explained in APPENDIX A. The FMTransmitterReceiverAssemblyController controls which of these two *Resources* is running based on the value of the property RF_MODE explained in section 3.4.3.4.

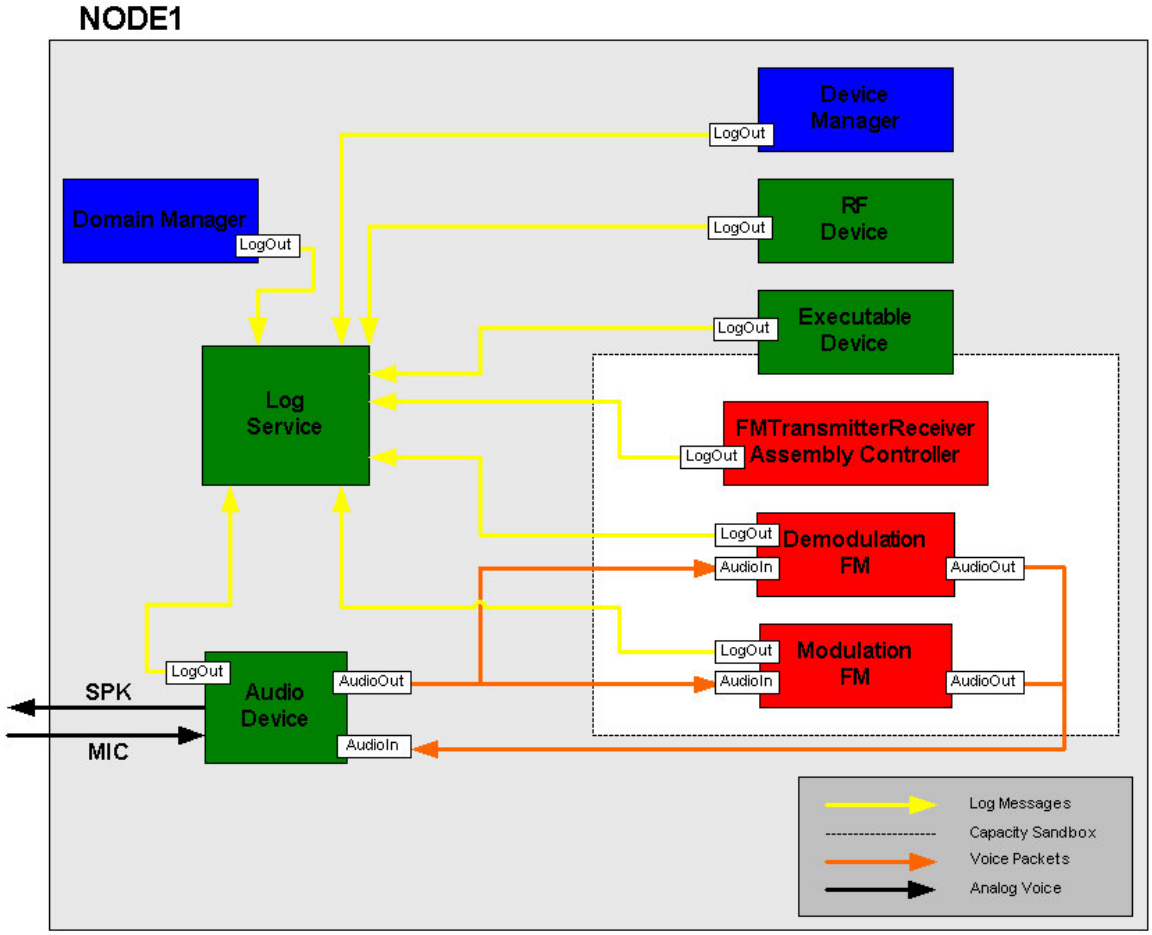


Figure 6 – SCA Reference Implementation: Application on a single node.

3.2. Demo Procedures

For the demonstration procedures please see the "AnalogFM Application User Guide.pdf" document for more information.

3.3. Node Components

3.3.1. devices.AudioDevice

3.3.1.1.Class Diagram

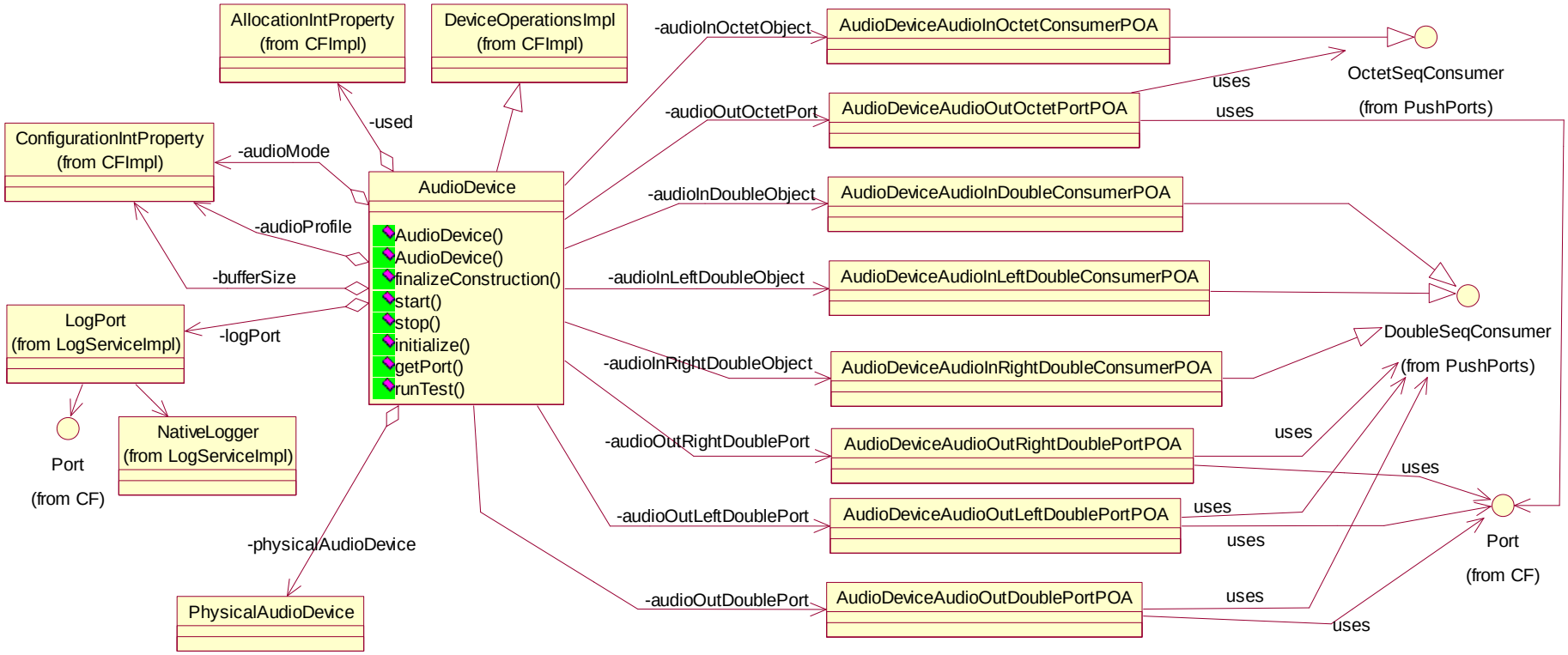


Figure 7 – AudioDevice Class Diagram

3.3.1.2. Description

The AudioDevice class is an SCA Device implementation that wraps the audio class (e.g. PhysicalAudioDevice) which provides full-duplex access to a soundcard.

This class provides inputs and an outputs port to the soundcard. The input ports are used to play sound by the soundcard. The output ports are used to get data from the microphone attached to the soundcard.

3.3.1.3. Ports

The Audio Device has nine available ports. The code shown below (from AudioDevice.java) defines those ports:

```
594   public org.omg.CORBA.Object getPort(String portName)
595          throws SCA.CF.PortSupplierPackage.UnknownPort
596   {
597     org.omg.CORBA.Object portObject = null;
598
599     if(portName.compareTo("AudioIn") == 0)
600     {
601       portObject = audioInOctetObject;
602     }
603     else if(portName.compareTo("AudioInDouble") == 0)
604     {
605       portObject = audioInDoubleObject;
606     }
607     else if(portName.compareTo("AudioInLeftDouble") == 0)
608     {
609       portObject = audioInLeftDoubleObject;
610     }
611     else if(portName.compareTo("AudioInRightDouble") == 0)
612     {
613       portObject = audioInRightDoubleObject;
614     }
615     else if(portName.compareTo("AudioOut") == 0)
616     {
617       portObject = audioOutOctetPort;
618     }
619     else if(portName.compareTo("AudioOutDouble") == 0)
620     {
621       portObject = audioOutDoublePort;
622     }
623     else if(portName.compareTo("AudioOutLeftDouble") == 0)
624     {
625       portObject = audioOutLeftDoublePort;
626     }
627     else if(portName.compareTo("AudioOutRightDouble") == 0)
628     {
629       portObject = audioOutRightDoublePort;
630     }
631     else if(portName.compareTo("LogPort") == 0)
```

```
632     {
633       portObject = logPort;
634     }
635     else
636     {
637       String msg = "[AudioDevice:getPort] Port '";
638       msg += portName + "' is unknown";
639       throw new SCA.CF.PortSupplierPackage.UnknownPort(msg);
640     }
641
642     return portObject;
643
644   } // getPort()
```

PROVIDES PORTS:

The implementation of all the following ports are done in AudioDevice.java.

- AudioIn: Invoking getPort() with "AudioIn" returns an object of type *OctetSeqConsumer* and it is used to send play sound by the soundcard. The data is received from a connected *Resource* and is written directly to the soundcard.

```
688 class AudioDeviceAudioInOctetConsumerPOA extends OctetSeqConsumerPOA
689 {
690   protected PhysicalAudioDevice physicalAudioDevice;
691   protected boolean isRunning = false;
692   protected SCA.LogServiceImpl.NativeLogger logger;
.....
733   public void processOctetMsg(byte[] msg, SCA.CF.DataType[] options)
734   {
735     if(isRunning)
736     {
.....
741     }
742   }
743 } // class AudioInPort
```

- AudioInDouble: Invoking getPort() with "AudioInDouble" returns an object of type *DoubleSeqConsumer* and it is used to play sound by the soundcard. The data is received from a connected *Resource* and converted from a double to a byte (1 double = 2 bytes).

```
746 class AudioDeviceAudioInDoubleConsumerPOA extends DoubleSeqConsumerPOA
747 {
748   protected PhysicalAudioDevice physicalAudioDevice;
749   protected boolean isRunning = false;
750   protected SCA.LogServiceImpl.NativeLogger logger;
.....
793   public void processDoubleMsg(double[] msg, SCA.CF.DataType[] options)
```

```
794   {
795     if(isRunning)
796     {
……
812     }
813   }
814 } // class AudioInDoublePort
```

- AudioInLeftDouble: Invoking getPort() with "AudioInLeftDouble" returns an object of type *DoubleSeqConsumer* and it is used to play sound by the soundcard. The data is received from a connected *Resource* and converted from a double to a byte (1 double = 2 bytes). Since the soundcard is in stereo mode the data has to be written only on the left channel as shown in Table 1.

```
1552 class AudioDeviceAudioInLeftDoubleConsumerPOA extends DoubleSeqConsumerPOA
1553 {
1554   protected PhysicalAudioDevice physicalAudioDevice;
1555   protected boolean isRunning = false;
1556   protected SCA.LogServiceImpl.NativeLogger logger;
1557   protected ConfigurationIntProperty audioMode;
……
1616   public void processDoubleMsg(double[] msg, SCA.CF.DataType[] options)
1617   {
1618     if(isRunning)
1619     {
……
1637     }
1638   }
1639 } // class AudioInLeftDoublePort
```

msg[0]	msg [1]	msg [2]	msg [3]
1 Byte	1 Byte	1 Byte	1 Byte
Data	Data	0	0
Left Channel		Right Channel	

Table 1 – Byte Buffer of the AudioInLeftDouble port

- AudioInRightDouble: Invoking getPort() with "AudioInRightDouble" returns an object of type *DoubleSeqConsumer* and it is used to play sound by the soundcard. The data is received from a connected *Resource* and converted from a double to a byte (1 double = 2 bytes). Since the soundcard is in stereo mode the data has to be written only on the right channel as shown in Table 2.

```
1641 class AudioDeviceAudioInRightDoubleConsumerPOA extends DoubleSeqConsumerPOA
```

```
1642 {
1643   protected PhysicalAudioDevice physicalAudioDevice;
1644   protected boolean isRunning = false;
1645   protected SCA.LogServiceImpl.NativeLogger logger;
1646   protected ConfigurationIntProperty audioMode;
.....
1706   public void processDoubleMsg(double[] msg, SCA.CF.DataType[] options)
1707   {
1708     if(isRunning)
1709     {
.....
1727     }
1728   }
1729 } // class AudioInRightDoublePort
```

msg[0]	msg [1]	msg [2]	msg [3]
1 Byte	1 Byte	1 Byte	1 Byte
0	0	Data	Data

Left Channel Right Channel

Table 2 – Byte Buffer of the AudioInRightDouble port

USES PORTS:

The implementation of all the following ports are done in AudioDevice.java.
When a *Resource* is connecting to a uses port that outputs data (see the connectPort() function in the code), the port adds this *Resource* in a consumerObjectList List. Several *Resources* can be connected to the same Out Port. When the first *Resource* is added to the list, the Port is being added to the consumerListeners List of the AudioDeviceReader. Several Ports can be connected to the AudioDeviceReader as shown in Figure 8. The AudioDeviceReader is reading the data (a byte buffer) from the soundcard and calls process() to all the Ports that are in it's consumerListeners List to give them the data. Then each Ports, is converting the received data as explained below and gives it to all the *Resources* that are in their individual List.

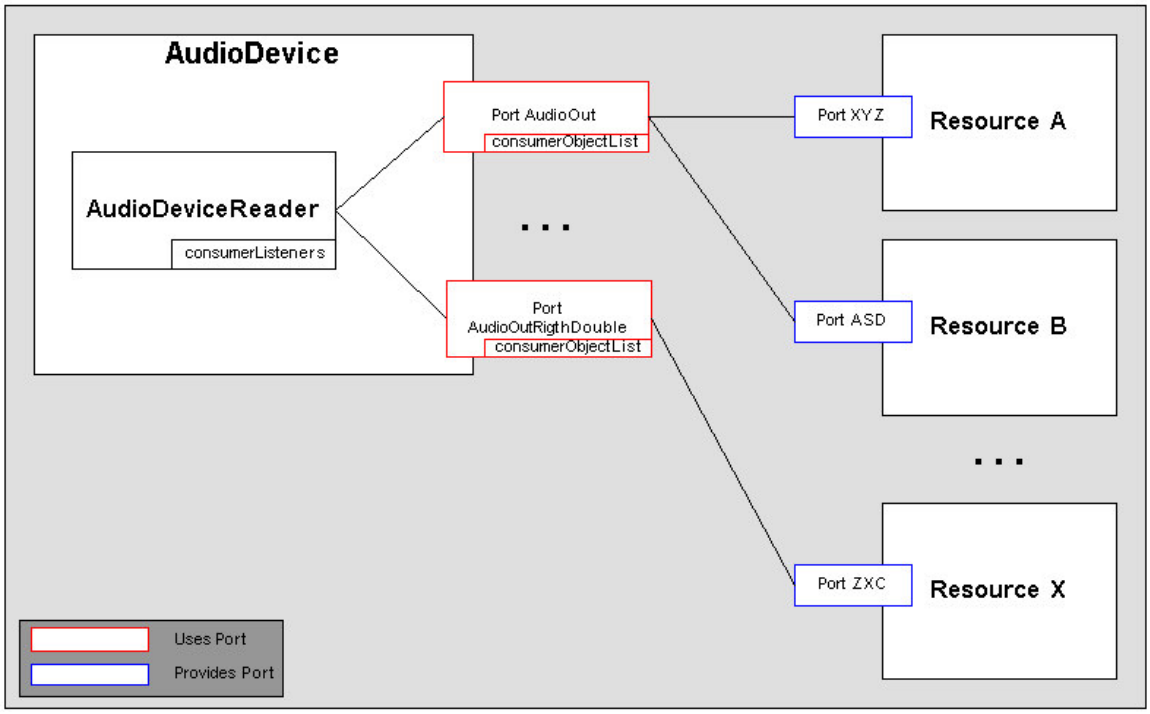


Figure 8 – AudioDevice Provides Ports Connections

- AudioOut: Invoking getPort() with "AudioOut" returns an object of type *OctetSeqProducer* and it is used to get data from the soundcard. No modification is done to the received data.

```
924 class AudioDeviceAudioOutOctetPortPOA
925       extends SCA.CF.PortPOA
926       implements AudioDeviceReader.ConsumerListener
927 {
928   protected DeviceOperationsImpl associatedDevice;
929   protected AudioDeviceReader audioDeviceReader;
930   protected java.util.Hashtable consumerObjectList;
931   protected boolean isRunning = false;
932   protected SCA.LogServiceImpl.NativeLogger logger;
……
973   public synchronized void connectPort(org.omg.CORBA.Object connection,
974                                        String connectionID)
975         throws SCA.CF.PortPackage.InvalidPort,
976                SCA.CF.PortPackage.OccupiedPort
977   {
……
995   }
……
1025   public void process(byte[] audioBuffer)
1026   {
……
```

```
1047    } //end of run
1048 } // class AudioDeviceAudioOutOctetPortPOA
```

- AudioOutDouble: Invoking getPort() with "AudioOutDouble" returns an object of type *DoubleSeqProducer* and it's used to get data from the soundcard. The received data is converted from a byte to a double buffer (2 bytes = 1 double).

```
1051 class AudioDeviceAudioOutDoublePortPOA
1052       extends SCA.CF.PortPOA
1053       implements AudioDeviceReader.ConsumerListener
1054 {
1055   protected DeviceOperationsImpl associatedDevice;
1056   protected AudioDeviceReader audioDeviceReader;
1057   protected java.util.Hashtable consumerObjectList;
1058   protected boolean isRunning = false;
1059   protected SCA.LogServiceImpl.NativeLogger logger;
.....
1100   public synchronized void connectPort(org.omg.CORBA.Object connection,
1101                                        String connectionID)
1102          throws SCA.CF.PortPackage.InvalidPort,
1103                 SCA.CF.PortPackage.OccupiedPort
1104   {
.....
1121   }
.....
1152   public void process(byte[] audioBuffer)
1153   {
.....
1187   } //end of run
1188
1189 } // class AudioDeviceAudioOutDoublePortPOA
```

- AudioOutLeftDouble: Invoking getPort() with "AudioOutLeftDouble" returns an object of type *DoubleSeqProducer* and it's used to get data from the soundcard. The data received is converted from a byte to a double buffer (2 bytes = 1 double). Since the soundcard is in stereo mode the data has to be read only from the left channel as shown in Table 3.

```
1193 class AudioDeviceAudioOutLeftDoublePortPOA
1194       extends SCA.CF.PortPOA
1195       implements AudioDeviceReader.ConsumerListener
1196 {
1197   protected DeviceOperationsImpl associatedDevice;
1198   protected AudioDeviceReader audioDeviceReader;
1199   protected java.util.Hashtable consumerObjectList;
1200   protected boolean isRunning = false;
1201   protected SCA.LogServiceImpl.NativeLogger logger;
1202   protected ConfigurationIntProperty audioMode;
```

```
.....
1251   public synchronized void connectPort(org.omg.CORBA.Object connection,
1252                                        String connectionID)
1253           throws SCA.CF.PortPackage.InvalidPort,
1254                  SCA.CF.PortPackage.OccupiedPort
1255   {
.....
1272   }
.....
1329   public void process(byte[] audioBuffer)
1330   {
.....
1367   } //end of run
1368
1369 } // class AudioDeviceAudioOutLeftDoublePortPOA
```

audioBuffer[0]	audioBuffer[1]	audioBuffer[2]	audioBuffer[3]
1 Byte	1 Byte	1 Byte	1 Byte
Data	Data	Data	Data

Left Channel Right Channel

Table 3 – Byte Buffer of the AudioOutLeftDouble port

- AudioOutRightDouble: Invoking getPort() with "AudioOutRightDouble" returns an object of type *DoubleSeqProducer* and it's used to get data from the soundcard. The data received is converted from a byte to a double buffer (2 bytes = 1 double). Since the soundcard is in stereo mode the data has to be read only from the right channel as shown in Table 4.

```
1372 class AudioDeviceAudioOutRightDoublePortPOA
1373       extends SCA.CF.PortPOA
1374       implements AudioDeviceReader.ConsumerListener
1375 {
1376   protected DeviceOperationsImpl associatedDevice;
1377   protected AudioDeviceReader audioDeviceReader;
1378   protected java.util.Hashtable consumerObjectList;
1379   protected boolean isRunning = false;
1380   protected SCA.LogServiceImpl.NativeLogger logger;
1381   protected ConfigurationIntProperty audioMode;
.....
1432   public synchronized void connectPort(org.omg.CORBA.Object connection,
1433                                        String connectionID)
1434           throws SCA.CF.PortPackage.InvalidPort,
1435                  SCA.CF.PortPackage.OccupiedPort
1436   {
.....
1453   }
```

```
.....
1510   public void process(byte[] audioBuffer)
1511   {
.....
1546   } //end of run
1547
1548 } // class AudioDeviceAudioOutRightDoublePortPOA
```

audioBuffer[0]	audioBuffer[1]	audioBuffer[2]	audioBuffer[3]
1 Byte	1 Byte	1 Byte	1 Byte
Data	Data	Data	Data

Left Channel Right Channel

Table 4 – Byte Buffer of the AudioOutRightDouble port

- LogPort: Invoking getPort() with "LogPort" returns an object of type Port that can be connected to a Log.
  The LogPort, which is the port accepting a connection to a Log, is created in AudioDeviceServer.java before the *Device* is created. The created LogPort is then passed to the constructor in AudioDevice.java:

```
296   public void createObject()
297          throws Exception
298   {
.....
305       SCA.LogServiceImpl.LogPort logger =
306              new SCA.LogServiceImpl.LogPort(execDeviceID,
307                                      "AudioDevice_" + execDeviceName);
308       implementation =
309                new devices.AudioDevice(deviceMgr,
310                                        execDeviceSpd,
311                                        execDeviceID,
312                                        execDeviceName,
313                                        parentDevice,
314                                        logger);
.....
341   }
```

Then we create the servant of the LogPort implementation (logger) to make it accessible through CORBA:

```
397   public void finalizeConstruction(SCA.CF.Device device,
398                                    org.omg.PortableServer.POA newPoa)
399           throws InstantiationException
400   {
.....
470       SCA.CF.PortPOATie logPortPOATie =
471                   new SCA.CF.PortPOATie((SCA.CF.PortOperations) logger);
472       logPort =
473      SCA.CF.PortHelper.narrow(newPoa.servant_to_reference(logPortPOATie));
.....
483   } // finalizeConstruction()
```

3.3.1.4. Configurable Properties

This Device has three configuration properties of type "long" (e.g. ConfigurationLongProperty) and one allocation property of type "long" (e.g. AllocationLongProperty):

- AUDIO_PROFILE: readable and writeable property of type "long" used to indicate to the soundcard device driver which kind of application will be using it. The following values are Linux dependent. Refer to the soundcard device driver "HowTos" for more information.
  - 0 = this value must be used for normal applications
  - 1 = this value must be used for applications that may experience under runs caused by an "external" delay
  - 2 = this value must be used for applications that may experience under runs caused by an "overheating" the CPU

  The lines code from 218 to 263 shows the definition of the property and line 265 shows how the property is added.

```
218     // Property audioProfile
219     // -----------------
220     audioProfile = new ConfigurationIntProperty("AUDIO_PROFILE",
221                                                 AnyProperty.READ_WRITE,
222                                                   AnyProperty.VISIBLE)
223     {
224       public boolean setValue(int newValue)
225       {
226         if((newValue < 0) || (newValue >= 3))
227         {
228           //out of range
229           return false;
230         }
.....
262       }
```

```
263     }; // audioProfile
264
265     addProperty(audioProfile.getName(), audioProfile);
```

- AUDIO_MODE: readable and writeable property of type "long" used to indicate how to sample the input signal to the soundcard. The following values are valid:
  - 0 = Mono, 8 bits, 8000Hz
  - 1 = Mono, 8 bits, 16000Hz
  - 2 = Mono, 8 bits, 22050Hz
  - 3 = Mono, 8 bits, 44100Hz
  - 4 = Mono, 8 bits, 48000Hz
  - 5 = Mono, 8 bits, 96000Hz
  - 6 = Mono, 16 bits, 8000Hz (default value)
  - 7 = Mono, 16 bits, 1600Hz
  - 8 = Mono, 16 bits, 22050Hz
  - 9 = Mono, 16 bits, 44100Hz
  - 10 = Mono, 16 bits, 48000Hz
  - 11 = Mono, 16 bits, 96000Hz
  - 12 = Stereo, 8 bits, 8000Hz
  - 13 = Stereo, 8 bits, 16000Hz
  - 14 = Stereo, 8 bits, 22050Hz
  - 15 = Stereo, 8 bits, 44100Hz
  - 16 = Stereo, 8 bits, 48000Hz
  - 17 = Stereo, 8 bits, 96000Hz
  - 18 = Stereo, 16 bits, 8000Hz
  - 19 = Stereo, 16 bits, 16000Hz
  - 20 = Stereo, 16 bits, 22050Hz
  - 21 = Stereo, 16 bits, 44100Hz
  - 22 = Stereo, 16 bits, 48000Hz
  - 23 = Stereo, 16 bits, 96000Hz

  The lines code from 267 to 312 shows the definition of this property and line 313 shows how it's added to the device.

```
267     // Property audioMode
268     // ------------------
269     audioMode = new ConfigurationIntProperty("AUDIO_MODE",
270                                              AnyProperty.READ_WRITE,
271                                              AnyProperty.VISIBLE)
272     {
273       public boolean setValue(int newValue)
274       {
275         if((newValue < 0) || (newValue >= 24))
276         {
277           //out of range
278           return false;
279         }
```

```
.....
311         }
312       }; // audioMode
313       addProperty(audioMode.getName(), audioMode);
```

- AUDIO_BUFFER_SIZE: readable and writeable property of type "integer" used to set/get the buffer size of the device.
  The lines code from 315 to 370 shows the definition of this property and line 371 shows how it's added to the device.

```
315       // Property bufferSize
316       // -------------------
317       bufferSize = new ConfigurationIntProperty("AUDIO_BUFFER_SIZE",
318                                                 AnyProperty.READ_WRITE,
319                                                 AnyProperty.VISIBLE)
320       {
321         public boolean setValue(int newValue)
322         {
323           if(newValue < 0)
324           {
325             //out of range
326             return false;
327           }
.....
359         }
.....
370       }; // bufferSize
371       addProperty(bufferSize.getName(), bufferSize);
```

- USED: read-only property of type long that indicates how much capacity is allocated. This property can only allocate one unit. It is used as a Boolean property for determining if the Device is in use or not.
  The lines code from 373 to 378 shows the definition of this property and line 379 shows how it's added to the device.

```
373       // Used Allocation Property
374       // --------------------
375       used = new AllocationIntProperty("USED",
376                                        AnyProperty.READ_ONLY,
377                                        AnyProperty.VISIBLE,
378                                        1);
379       addProperty(used.getName(), used);
```

3.3.2. devices.PhysicalAudioDevice

3.3.2.1.Class Diagram

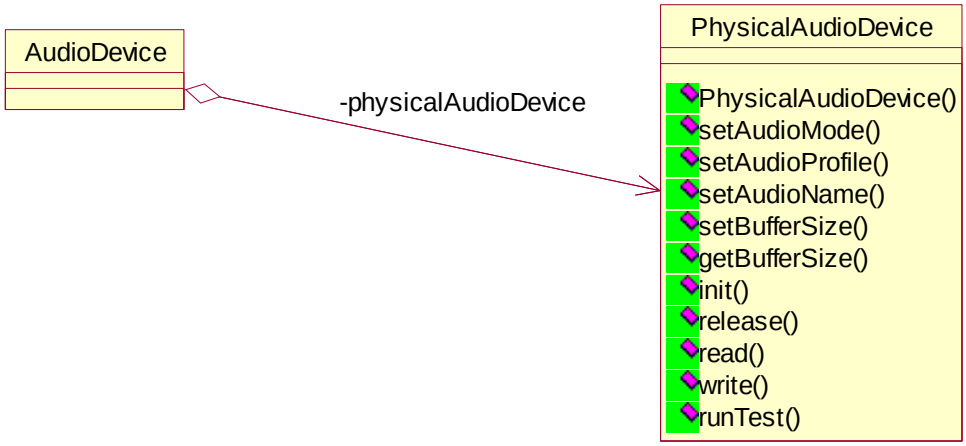


Figure 9 – PhysicalAudioDevice Class Diagram

3.3.2.2.Description

The PhysicalAudioDevice class implements the Java Native Interface calls used by the AudioDevice class to access the soundcard.

3.3.3. devices.RFDevice

3.3.3.1. Class Diagram

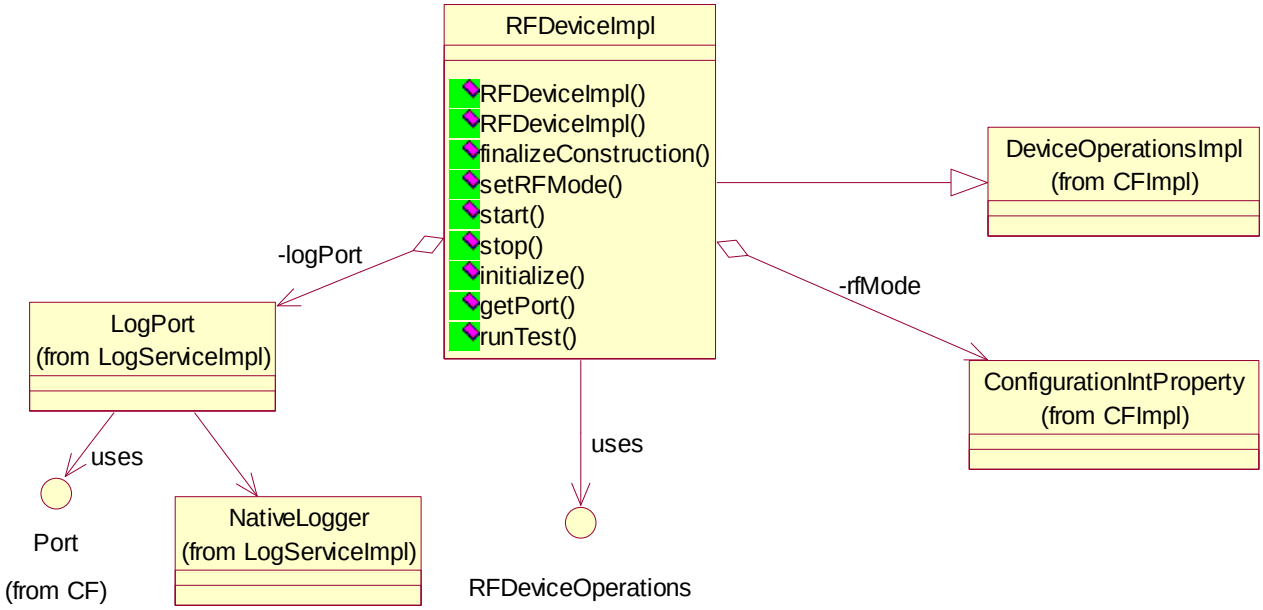


Figure 10 – RFDevice Class Diagram

3.3.3.2. Description

The RFDevice class is an SCA Device implementation that controls the CRC RF board, see the "Analog FM Hardware Design Document" for more information.

3.3.3.3. Ports

This *Device* has only one port named LogPort. The code shown below (from RFDeviceImpl.java) defines those ports:

```
368   public org.omg.CORBA.Object getPort(String portName)
369          throws SCA.CF.PortSupplierPackage.UnknownPort
370   {
371     org.omg.CORBA.Object portObject = null;
372
373     if(portName.compareTo("LogPort") == 0)
374     {
375       portObject = logPort;
376     }
377     else
378     {
```

```
379       String msg = "[RFDeviceImpl:getPort] Port '";
380       msg += portName + "' is unknown";
381       throw new SCA.CF.PortSupplierPackage.UnknownPort(msg);
382     }
383
384     return portObject;
385
386   } // getPort()
```

Invoking getPort() with "LogPort" returns an object of type Port that can be connected to a Log.
The LogPort, which is the port accepting a connection to a Log, is created in RFDeviceImplServer.java before the *Device* is created. The created LogPort is then passed to the constructor in RFDeviceImpl.java:

```
295   public void createObject()
296           throws Exception
297   {
.....
304       SCA.LogServiceImpl.LogPort logger =
305               new SCA.LogServiceImpl.LogPort(execDeviceID,
306                                              "RFDeviceImpl_" + execDeviceName);
307       implementation =
308                 new devices.RFDeviceImpl(deviceMgr,
309                                          execDeviceSpd,
310                                          execDeviceID,
311                                          execDeviceName,
312                                          parentDevice,
313                                          logger);
.....
338   }
```

Then we create the servant of the LogPort implementation (logger) to make it accessible through CORBA:

```
197   public void finalizeConstruction(SCA.CF.Device device,
198                                     org.omg.PortableServer.POA newPoa)
199           throws InstantiationException
200   {
.....
212       SCA.CF.PortPOATie logPortPOATie =
213                     new SCA.CF.PortPOATie((SCA.CF.PortOperations) logger);
214       logPort =
215      SCA.CF.PortHelper.narrow(newPoa.servant_to_reference(logPortPOATie));
.....
225   } // finalizeConstruction()
```

3.3.3.4. Configurable Properties

This *Device* has one configuration properties of type "long" (e.g. ConfigurationLongProperty). The property is named RF_MODE. It's a readable and writeable property of type integer used to set the radio in receive or transmit mode:

- RF_MODE = 1: the RF board is configured in receive mode.
- RF_MODE = 2: the RF board is configured in transmit mode.

The lines code from 128 to 176 shows the definition of this property and line 177 shows how it's added to the *resource*.

```
128     // Property rfMode
129     // -----------------
130     rfMode = new ConfigurationIntProperty("RF_MODE",
131                                           AnyProperty.READ_WRITE,
132                                           AnyProperty.VISIBLE)
133     {
134       public boolean setValue(int newValue)
135       {
136         boolean sucess = true;
137         if((newValue < 0) || (newValue >= 3))
138         {
139           //out of range
140           return false;
141         }
.....
173       }
174     }; // rfMode
175
176     rfMode.setValue(devices.RFDevicePackage.RFMode._ReceiveOnly);
177     addProperty(rfMode.getName(), rfMode);
```

3.4. Analog FM Application Components

The components needed by the application are: DemodulationFM Resource, ModulationFM Resource, FMTransmitterReceiver Assembly Controller

3.4.1. resources.AnalogFM.DemodulationFMResource

3.4.1.1. Class Diagram

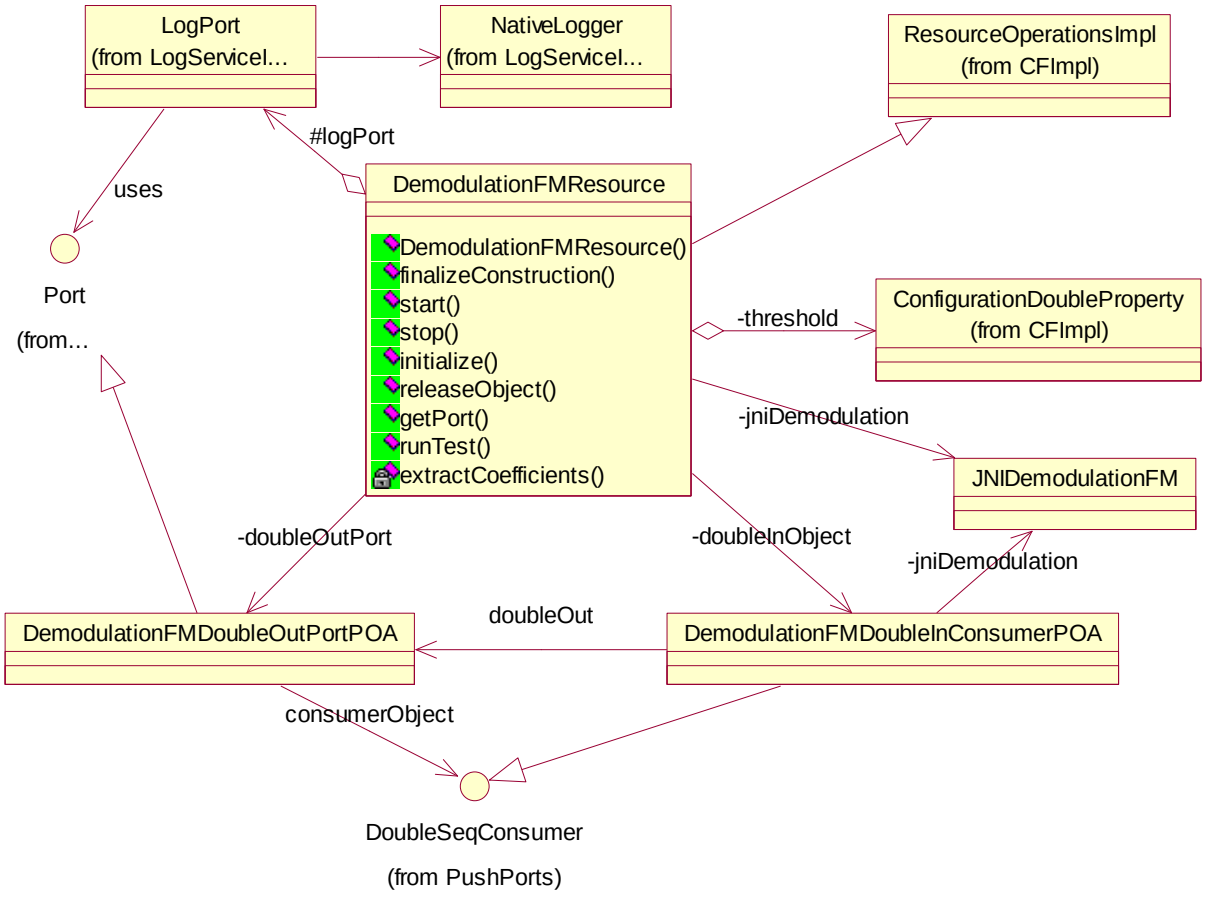


Figure 11 – DemodulationFMResource Class Diagram

3.4.1.2. Description

The DemodulationFMResource class is an SCA Resource implementation that takes audio input port, performs an FM Demodulation transform then forwards it to the audio output port.

3.4.1.3. Ports

This *Resource* has three available ports. The code shown below (from DemodulationFMResource.java) defines those ports which are supported by this *Resource*.

```
347   public org.omg.CORBA.Object getPort(String portName)
348          throws SCA.CF.PortSupplierPackage.UnknownPort
349   {
350     org.omg.CORBA.Object portObject = null;
351
352     if(portName.compareTo("DoubleSeqIn") == 0)
353     {
354       portObject = doubleInObject;
355     }
356     else if(portName.compareTo("DoubleSeqOut") == 0)
357     {
358       portObject = doubleOutPort;
359     }
360     else if(portName.compareTo("LogPort") == 0)
361     {
362       portObject = logPort;
363     }
364     else
365     {
366       String msg = "[DemodulationFMResource:getPort] Port '";
367       msg += portName + "' is unknown";
368       throw new SCA.CF.PortSupplierPackage.UnknownPort(msg);
369     }
370
371     return portObject;
372   } // getPort()
```

- DoubleSeqIn: This port returns an object of type DoubleSeqConsumer and it is used to send the data to the consumer. The data received from the soundcard is then process in processDoubleMsg() as explained in section "A.2 FM Demodulation" of the APPENDIX A.

```
419 class DemodulationFMDoubleInConsumerPOA extends DoubleSeqConsumerPOA
420 {
421   DemodulationFMDoubleOutPortPOA doubleOut;
422   private JNIDemodulationFM jniDemodulationFM;
423   private boolean isRunning;
.....
471   public void processDoubleMsg(double[] data, SCA.CF.DataType[] options)
```

```
472   {
……
505   }
506 } // class DemodulationFMDoubleInConsumerPOA
```

- DoubleSeqOut: This port returns an object of type DoubleSeqProducer and it is used to receive data from a producer.

- LogPort: LogPort. Invoking getPort() with "LogPort" returns an object of type Port that can be connected to a Log.
  The LogPort, which is the port accepting a connection to a Log, is created in DemodulationFMResourceServer.java before the *Resource* is created. The created LogPort is then passed to the constructor in DemodulationFMResource.java:

```
286   public void createObject()
287          throws Exception
288   {
……
295       SCA.LogServiceImpl.LogPort logger =
296              new SCA.LogServiceImpl.LogPort(resourceID, resourceName);
297       implementation = new DemodulationFMResource(resourceSpd,
298                                                   resourceID,
299                                                   resourceName,
300                                                   logger);
……
318   }
```

Then we create the servant of the LogPort implementation (logger) to make it accessible through CORBA:

```
166   public void finalizeConstruction(SCA.CF.Resource resource,
167                                    org.omg.PortableServer.POA newPoa)
168          throws InstantiationException
169   {
……
196       SCA.CF.PortPOATie logPortPOATie =
197                                 new SCA.CF.PortPOATie((LogPort)logger);
198
199       logPort = SCA.CF.PortHelper.narrow(
200                            newPoa.servant_to_reference(logPortPOATie));
……
210   } // finalizeConstruction()
```

3.4.1.4. Configurable Properties

This *Resource* has one configuration properties of type "double" (e.g. ConfigurationDoubleProperty) the property is named THRESHOLD. It's a readable and writeable property of type double that acts as a type of squelch triggering demodulation only if the RF signal detected has enough energy. The value must be between 0.000001 and 0.0 inclusively.
The lines code from 112 to 146 shows the definition of this property and line 147 shows how it's added to the *resource*.

```
111     // -----------------
112     threshold = new ConfigurationDoubleProperty("THRESHOLD",
113                                                 AnyProperty.READ_WRITE,
114                                                   AnyProperty.VISIBLE)
115     {
116       public boolean setValue(double newValue)
117       {
.....
145       }
146     }; // threshold
147     addProperty(threshold.getName(), threshold);
```

3.4.2. resources.AnalogFM.ModulationFMResource

3.4.2.1. Class Diagram

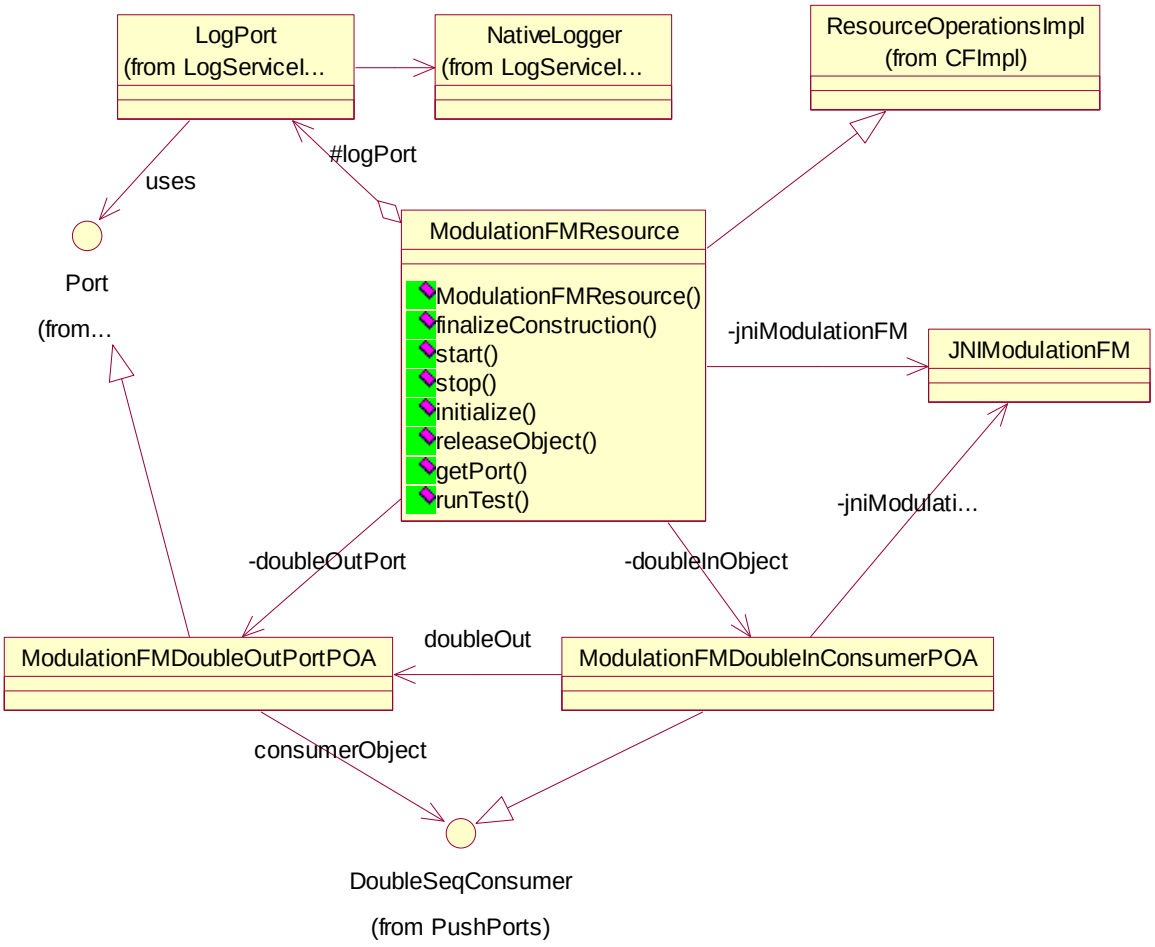


Figure 12 – ModulationFMResource Class Diagram

3.4.2.2. Description

The ModulationFMResource class is an SCA Resource implementation that takes audio input port, performs a FM modulation transform then forwards the dat to the audio output port.

3.4.2.3. Ports

This *Resource* has three available ports. The code shown below (from ModulationFMResource.java) defines those ports which are supported by this *Resource*.

```
278   public org.omg.CORBA.Object getPort(String portName)
279          throws SCA.CF.PortSupplierPackage.UnknownPort
280   {
281     org.omg.CORBA.Object portObject = null;
282
283     if(portName.compareTo("DoubleSeqIn") == 0)
284     {
285       portObject = doubleInObject;
286     }
287     else if(portName.compareTo("DoubleSeqOut") == 0)
288     {
289       portObject = doubleOutPort;
290     }
291     else if(portName.compareTo("LogPort") == 0)
292     {
293       portObject = logPort;
294     }
295     else
296     {
297       String msg = "[ModulationFMResourceNameResource:getPort] Port
'";
298       msg += portName + "' is unknown";
299       throw new SCA.CF.PortSupplierPackage.UnknownPort(msg);
300     }
301
302     return portObject;
303   } // getPort()
```

- DoubleSeqIn: This port returns an object of type DoubleSeqConsumer and it is used to send the data to the consumer. The data received from the soundcard is then process processDoubleMsg() as explained in section "A.1 FM Modulation" of the APPENDIX A.

```
331 class ModulationFMDoubleInConsumerPOA extends DoubleSeqConsumerPOA
332 {
333   ModulationFMDoubleOutPortPOA doubleOut;
334   private JNIModulationFM jniModulationFM;
335   private boolean isRunning;
……
381   public void processDoubleMsg(double[] data, SCA.CF.DataType[] options)
382   {
……
394   }
395 } // class ModulationFMDoubleInConsumerPOA
```

- DoubleSeqOut: This port returns an object of type DoubleSeqProducer and it is used to receive data from a producer.

- LogPort: LogPort. Invoking getPort() with "LogPort" returns an object of type Port that can be connected to a Log.
  The LogPort, which is the port accepting a connection to a Log, is created in ModulationFMResourceServer.java before the *Resource* is created. The created LogPort is then passed to the constructor in ModulationFMResource.java:

```
246   public void createObject()
247          throws Exception
248   {
……
255       SCA.LogServiceImpl.LogPort logger =
256              new SCA.LogServiceImpl.LogPort(resourceID,
resourceName);
257
258       implementation = new ModulationFMResource(resourceSpd,
259                                                 resourceID,
260                                                 resourceName,
261                                                 logger);
……
276   } // createObject
```

Then we create the servant of the LogPort implementation (logger) to make it accessible through CORBA:

```
115   public void finalizeConstruction(SCA.CF.Resource resource,
116                                     org.omg.PortableServer.POA newPoa)
117          throws InstantiationException
118   {
……
145       SCA.CF.PortPOATie logPortPOATie =
146                                new SCA.CF.PortPOATie((LogPort)logger);
147
148       logPort = SCA.CF.PortHelper.narrow(
149                               newPoa.servant_to_reference(logPortPOATie));
……
159   } // finalizeConstruction()
```

3.4.3. applications.FMTransmitterReceiverAssemblyController

3.4.3.1. Class Diagram

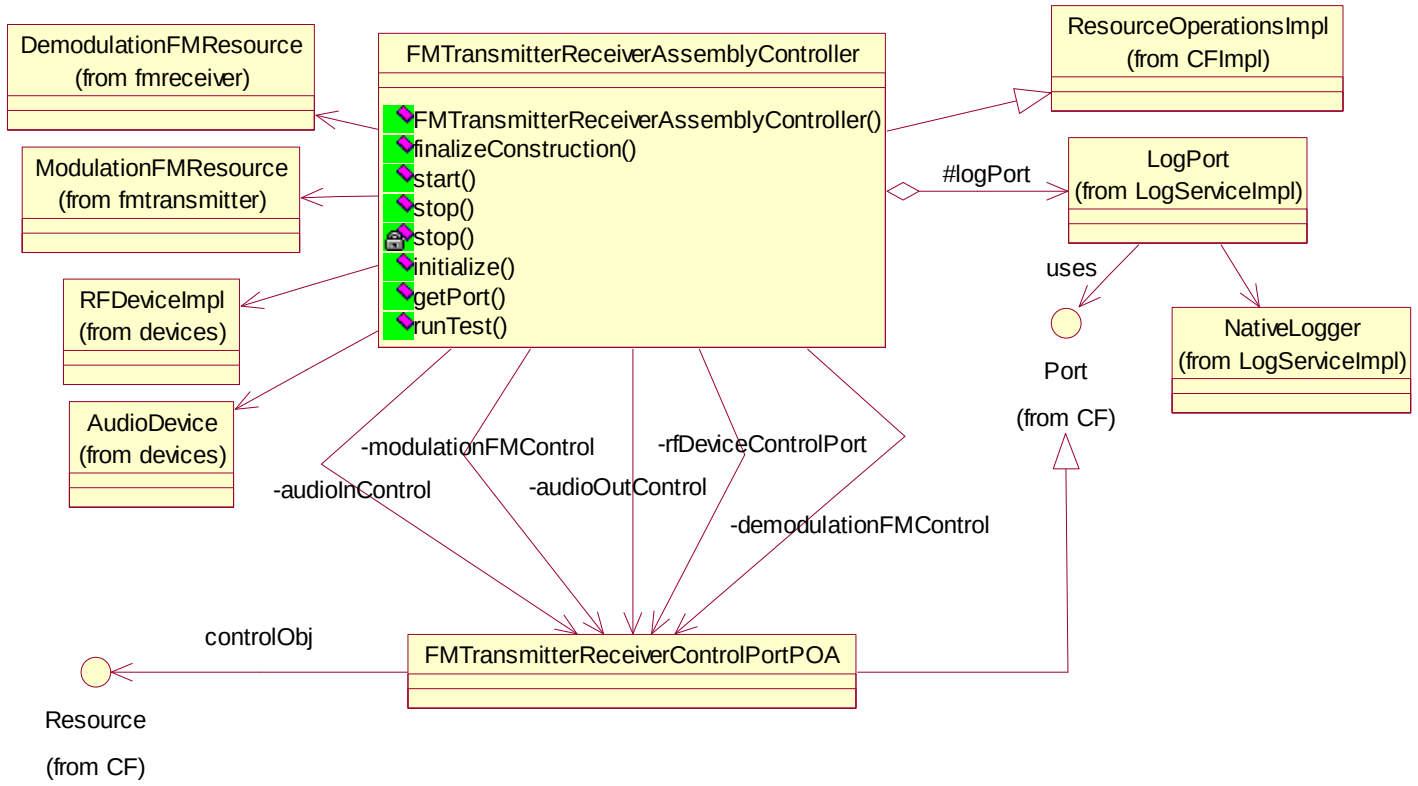


Figure 13 – FMTransmitterReceiverAssemblyController Class Diagram

3.4.3.2. Description

The FMTransmitterReceiverAssemblyController class is an SCA Resource implementation that controls the *Resources* of the application. The resources controlled are the ModulationFM *Resource* and the DemodulationFM resource.

3.4.3.3. Ports

This *Assembly Controller* has eight available ports. The code shown below (from FMTransmitterReceiverAssemblyController.java) defines those ports which are supported by this *Assembly Controller*.

```
594   public org.omg.CORBA.Object getPort(String portName)
595          throws SCA.CF.PortSupplierPackage.UnknownPort
596   {
597     org.omg.CORBA.Object portObject = null;
598
599     if(portName.compareTo("LogPort") == 0)
600     {
601       portObject = logPort;
602     }
603     else if(portName.compareTo("InputDeviceControl") == 0)
604     {
605       portObject = audioInControl;
606     }
607     else if(portName.compareTo("OutputDeviceControl") == 0)
608     {
609       portObject = audioOutControl;
610     }
611     else if(portName.compareTo("RFDeviceControl") == 0)
612     {
613       portObject = rfDeviceControlPort;
614     }
615     else if(portName.compareTo("DemodulationFMControl") == 0)
616     {
617       portObject = demodulationFMControl;
618     }
619     else if(portName.compareTo("ModulationFMControl") == 0)
620     {
621       portObject = modulationFMControl;
622     }
623     else
624     {
625       String msg = "[FMTransmitterReceiverAssemblyController:getPort]
Port '";
626       msg += portName + "' is unknown";
627       throw new SCA.CF.PortSupplierPackage.UnknownPort(msg);
628     }
629     return portObject;
630
631   } // getPort()
```

- InputDeviceControl: This port returns an object of type Port that can be connected to a *Device* from which the application will get its input data.
- OutputDeviceControl: This port returns an object of type Port that can be connected to a *Device* to which the application will send its output data.
- ModulationFMControl: This port returns an object of type Port that can be connected to a *Resource* to which the application will configure the modulation.
- DemodulationFMControl: This port returns an object of type Port that can be connected to a *Resource* to which the application will configure the demodulation.
- RFDeviceControl: This port returns an object of type Port that can be connected to a *Device* to which the application will configure the RF board.

```
655 class FMTransmitterReceiverControlPortPOA extends SCA.CF.PortPOA
656 {
657   /***/
658   public SCA.CF.Resource controlObj;
659   private String objectType;
……
694   public synchronized void connectPort(org.omg.CORBA.Object
connection,
695                                        String connectionID)
696         throws SCA.CF.PortPackage.InvalidPort,
697                SCA.CF.PortPackage.OccupiedPort
698   {
……
721   }
……
740 } // class FMTransmitterReceiverControlPortPOA
```

- LogPort: LogPort. Invoking getPort() with “LogPort” returns an object of type Port that can be connected to a Log.
  The LogPort, which is the port accepting a connection to a Log, is created in FMTransmitterReceiverAssemblyControllerServer.java before the *Resource* is created. The created LogPort is then passed to the constructor in FMTransmitterReceiverAssemblyController.java:

```
288   public void createObject()
289         throws Exception
290   {
……
297       SCA.LogServiceImpl.LogPort logger =
298             new SCA.LogServiceImpl.LogPort(resourceID,
299                                        serverName + "_" + resourceName);
300
301      implementation = new FMTransmitterReceiverAssemblyController(resourceSpd,
302                                                                   resourceID,
303                                                                 resourceName,
304                                                                    logger);
```

```
.....
322    }
```

Then we create the servant of the LogPort implementation (logger) to make it accessible through CORBA:

```
216    public void finalizeConstruction(Resource resource,
217                                     org.omg.PortableServer.POA _poa)
218           throws InstantiationException
219    {
.....
233     SCA.CF.PortPOATie logPortPOATie = new SCA.CF.PortPOATie((LogPort)logger);
234
235     logPort = SCA.CF.PortHelper.narrow(_poa.servant_to_reference(logPortPOATie));
.....
279    } // finalizeConstruction()
```

3.4.3.4. Configurable Properties

This *Assembly Controller* has two configuration properties of type "long" (e.g. ConfigurationLongProperty) that are defined in FMTransmitterReceiverAssemblyController.java:

- THRESHOLD: readable and writeable property of type double that acts as a type of squelch triggering demodulation only if the RF signal detected has enough energy. The value must be between 0.000001 and 0.0 inclusively.
  The lines code from 167 to 172 shows the definition of this property and line 196 shows how it's added to the *resource*.

```
165      // Property squelchThreshold
166      // ------------------
167      squelchThreshold = new ConfigurationDoubleProperty("THRESHOLD",
168                                                         AnyProperty.READ_WRITE,
169                                                          AnyProperty.VISIBLE)
170      {
171         public boolean setValue(DataType dataType)
172         {
.....
187         }
.....
195      }; // squelchThreshold
196      addProperty(squelchThreshold.getName(), squelchThreshold);
```

- RF_MODE: readable and writeable property of type integer used to set the radio in receive or transmit mode:
  - RF_MODE = 1: the RF board is configured in receive mode, the ModulationFMResource is stopped and the DemodulationFMResource is started.
  - RF_MODE = 2: the RF board is configured in transmit mode, the DemodulationFMResource is stopped and the ModulationFMResource is started.

  The lines code from 117 to 161 shows the definition of this property and line 162 shows how it's added to the *resource*.

```
114     /***
115       Property RF_MODE
116      ***/
117     rfMode = new ConfigurationIntProperty("RF_MODE",
118                                           AnyProperty.READ_WRITE,
119                                           AnyProperty.VISIBLE)
120     {
121       public boolean setValue(int newValue)
122       {
……
127           switch(newValue)
128           {
129             case 1:
130             {
131               modulationFMControlPOA.controlObj.stop();
132
RFDeviceHelper.narrow(rfDeviceControlPortPOA.controlObj).setRFMode(devices.
RFDevicePackage.RFMode.ReceiveOnly);
133               demodulationFMControlPOA.controlObj.start();
134               break;
135             }
136             case 2:
137             {
138               demodulationFMControlPOA.controlObj.stop();
139
RFDeviceHelper.narrow(rfDeviceControlPortPOA.controlObj).setRFMode(devices.
RFDevicePackage.RFMode.TransmitOnly);
140               modulationFMControlPOA.controlObj.start();
141               break;
142             }
……
160       }
161     }; // rfMode
162     addProperty(rfMode.getName(),  rfMode);
```

APPENDIX A. "ANALOG FM WAVEFORM DESCRIPTION"

A.1. FM Modulation

A.1.1. Block diagram for a FM Modulation

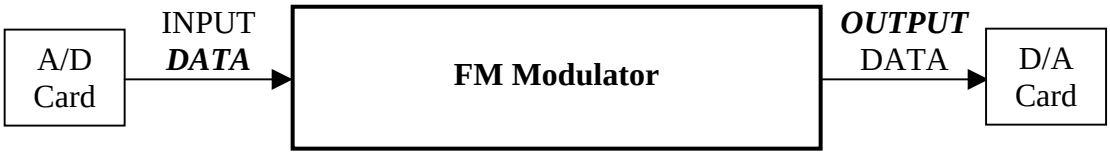


Figure 14 – Block diagram of the FM Modulation

The A/D card reads the input coming from the microphone. This signal is in band pass and is treated by the "INPUT DATA" port of the FM Modulator. The signal is then modulated on the "OUTPUT DATA" port of the FM Modulator and sent to the D/A card.

A.1.2. Block diagram of the FM Modulator

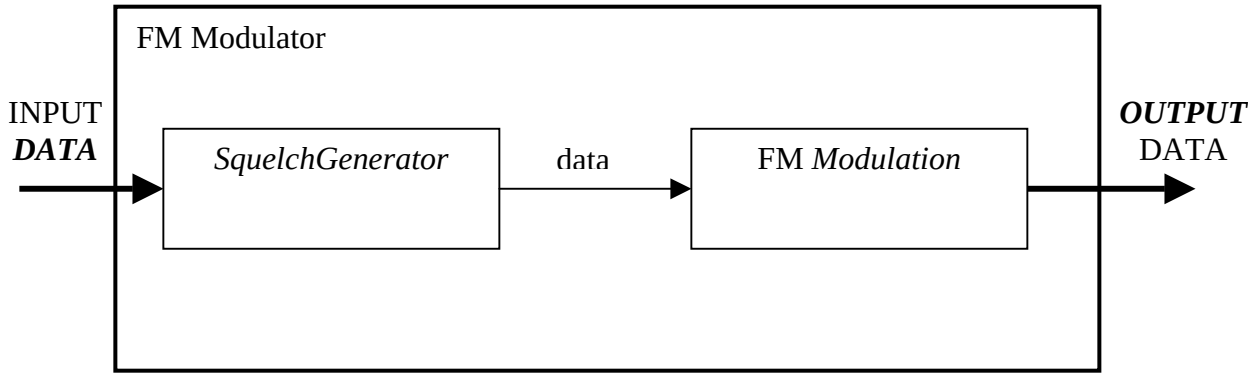


Figure 15 – Block diagram of the FM Modulator

Description:

Input:
The "INPUT DATA" port of the FM Modulator contains a base band signal.

SquelchGenerator:
A CTCSS (Continuous Tone Coded Squelch System) is added to the signal.

FM Modulation:
A FM Modulation of the interpolated signal.

Output:
The treated signal is sent on the "OUTPUT DATA" data port of the FM Modulator.

A.2. FM Demodulation

A.2.1. Block diagram for a FM Demodulation

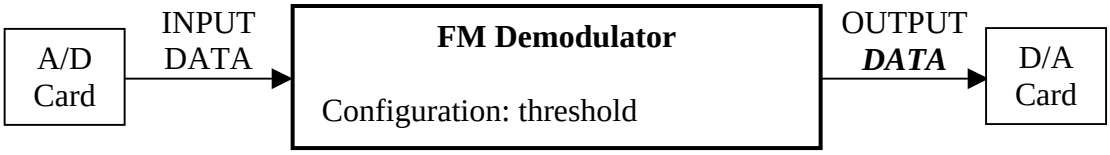


Figure 16 – Block diagram of the FM Demodulation

The A/D card reads the input coming from the microphone. This FM modulated signal is treated in the "INPUT DATA" port from the FM Demodulator. The signal is then demodulated on the "OUTPUT DATA" port of the FM Demodulator. Finally, this base band signal is sent to the D/A card.

Configuration parameters:

threshold: the limited value to determine if a signal is being detected.

A.2.2. Block diagram of the FM Demodulator

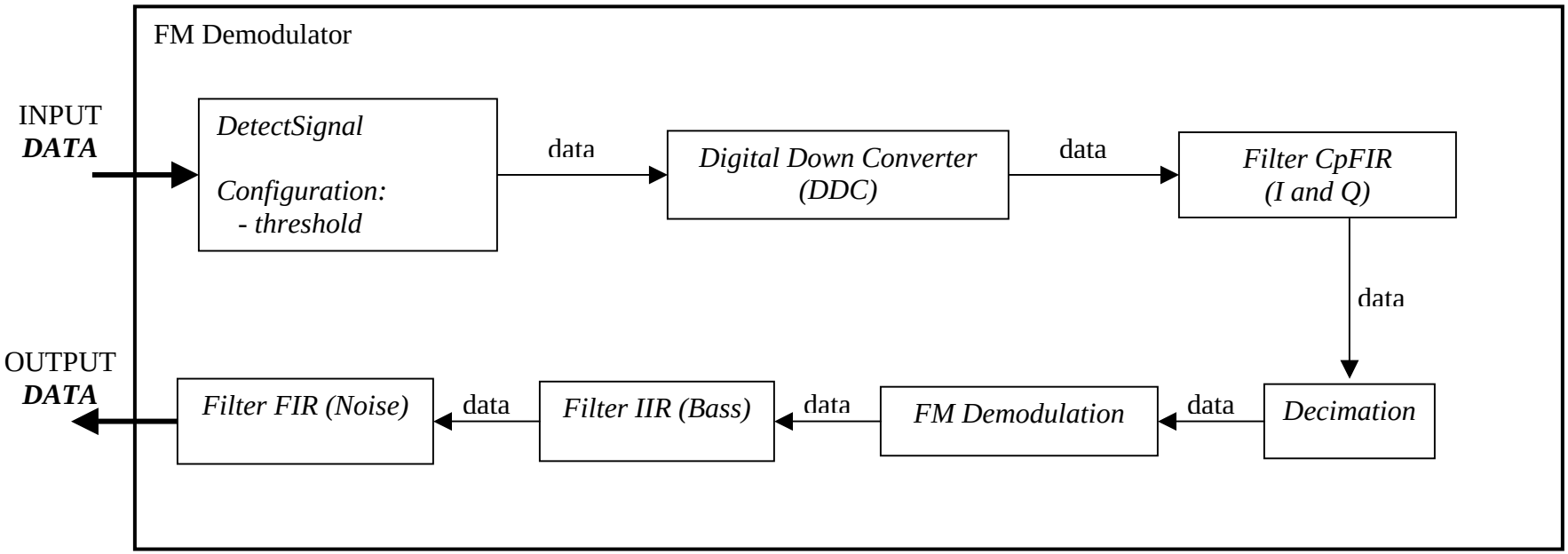


Figure 17 – Block diagram of the FM Demodulator

Description:

Input:
The "INPUT DATA" port of the FM Demodulator is a modulated FM signal.

DetectSignal:
The signal detector verify that there's a signal at the input. If true, then the data is sent to the DDC.

DDC:
The DDC break up the signal into I and Q in baseband.

Filter CpFIR:
The high frequency of I and Q are filtered in order to remove the high frequencies of the carrier.

Decimation:
Decimation is done in order to decrease the sampling rate.

FM Demodulation:
The FM Demodulation is done in order to obtain the signal in base band.

Filter FIR and Filter IIR:
The signal is passed through a low-pass filter in order to filter the low frequencies as well as the squelch. Then the signal is passed through a high-pass filter in order to filter the high frequencies as well as the infiltrated noise.

Output:
The modulated signal is sent on the "OUTPUT DATA" port of the FM Demodulator.