aiDevices Framework

Programming Guide

Contents

1. Introduction

The aiDevices framework represents the second generation Application Programming Interface (API) for automating and controlling Advent Instruments hardware products from a .Net programming environment. Unlike simplistic C-style drivers, this .Net assembly exposes a feature rich network of classes which enable application developers to write fine-tuned applications without sacrificing code clarity or flexibility.
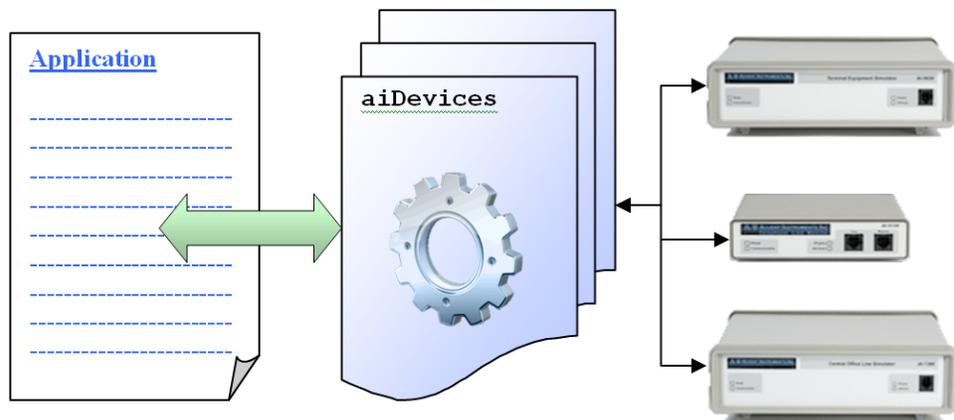
At its core the aiDevices framework contains a set of device classes which communicate with and manage the features of Advent Instruments' hardware products. These classes provide an abstract interface which allows applications access the rich feature set of each instrument.

To make things easier, the aiDevices framework also contains a periphery of supporting classes which make constructing, generating, and analyzing complicated signals very straight forward. These classes manage representations of physical quantities, complex impedances, signal level representations and even perform automatic conversion between different representations. The framework is capable of automatic Caller ID message construction and decoding based on industry standard formats while still enabling applications to customize down to the bit level.

The aiDevices project is coded to take full advantage of the powerful object-oriented programming features of the .Net environment. Re-usable device support classes results in abstract code which can be easily ported between instruments. The multi-threading features are utilized to produce "smart" objects which manage the details of generating and detecting complicated signals in the background while parent application code tends to other tasks. Delegation is also leveraged to deliver asynchronous information to applications which can be written using an event-driven structure.

This manual provides a detailed guided tour through the vast majority of the classes, features, and behaviors of the aiDevices assembly.

1.1. Document Scope

This manual is intended to serve as a primer for project managers and application developers to design successful applications that interact with Advent Instruments hardware products through the aiDevices framework.

While great care has been taken to ensure this document can be read by non-programmers, the vast majority of this document assumes the reader is experienced or at least familiar with most basic and some advanced object oriented programming concepts; especially as they pertain to .Net programming languages. These concepts include:

- Classes, inheritance, sub-classing, and class diagrams
- Delegation
- Abstraction
- Polymorphism
- Multi-threading

Readers are also expected to have a basic understanding of circuit theory and some analog telephony concepts.

This document is not intended to be an exhaustive API reference for each class within the assembly but rather a guide book to direct developers to the correct features of the aiDevices assembly. For specific documentation on particular classes, functions, or parameters please refer to the XML documentation described in section 6. This documentation is distributed with the assembly and appears automatically within Visual Studio.

1.2. Related Documents

The contents of this document pertain to the aiDevices framework and its interoperation with Advent Instruments products. For instrument specific documentation, the reader is referred to the following documents which are available at www.adventinstruments.com.

- ➢ **AI-5620 User Guide** – contains AI-5620 product overview, USB driver installation, and performance specification.

- ➢ **AI-7280 User Guide** – contains AI-7280 product overview, USB driver installation, and performance specification.

1.3. Document Organization

General Information

➢ **Introduction** – a general introduction to the assembly and this manual

➢ **Project Roadmap** – information about the future direction of the aiDevices project

➢ **Versioning and Compatibility** – information regarding the version numbers and compatibility issues

➢ **XML Documentation** – information about the XML documentation distributed with the assembly.

➢ **Programming Guidelines** – rules and tips regarding programming styles and design patterns used throughout aiDevices

Device Classes

➢ **Device Class Fundamentals** – reference for features and behaviors common to all classes that manage instruments

➢ **Instrument Time Management** – describes how timing is managed within instruments and the supporting software

➢ **Notifications** – contains documentation for the behavioral pattern used to report asynchronous information to parent applications

➢ **Exception Handling** – documents how exception conditions are handled throughout the framework

➢ **Debugging and Tracing** – describes the debugging and tracing features built into each device class.

Common Classes and Concepts

➢ **Descriptor Classes** – describes the supporting cast of descriptor classes which are used throughout the framework

➢ **Signal Descriptor Classes** – describes the set of classes devoted to describing signals which can be generated or detected by instruments

➢ **Caller ID (FSK)** – describes the set of classes devoted to describing Caller ID messages based on FSK signaling.

➢ **Device Support Classes** – describes the set of classes which are reused by several different instrument's device classes to implement particular features

Instrument Specific Classes

➢ **AI-5620 TE Simulator** – documents the classes specific to the AI-5620

➢ **AI-7280 CO Simulator** – documents the classes specific to the AI-7280

Reference

➢ **Terminology** – contains a list of terms and definitions which are commonly used within this manual and the aiDevices framework.

1.4. Documentation Conventions

1.4.1. Language, Style, and Normative Terms

This document is intended as a guide for project managers and developers to design successful applications with aiDevices. It is intended to be easy to read and uses a mixture of colloquial examples and formal language. When required, this manual will convey recommendations and requirements using the following normative language:

- **Must, Must Not** – these terms indicates strict rules. Non-compliance with such statements may result in serious errors or malformed applications.

- **May, Should, Can** – these terms indicate an optional requirement and should be treated as a guideline.

Some statements **will be written in bold faced text** to draw attention to important distinctions of relative importance.

1.4.2. Code Examples

The document includes may code snippets in C# and VB.net which are used to convey information relevant to the section in which they are found. Please note:

- All code examples are only short snippets of code and are not expected to function as complete programs. Several examples show multiple examples of the same operation and are not expected to execute sequentially.

- Some variable declarations may be omitted for the sake of brevity. Variables with names like _7280 are assumed to be of a type which is associated with an instrument of matching product number (in this case AI7280_CO_Simulator).

- Exception handling is omitted for the sake of clarity. Application developers should **<u>always</u>** write programs with suitable try/catch statements.

Each snippet of example code will appear as shown below.

Example:

```csharp
// C# examples will appear within this style of box
// All examples assume the following statements…

using System;
using Advent.aiDevices;
```

Example:

```vbnet
' VB.Net examples will appear within boxes like
' All such examples assume the following statements…

Imports Advent.aiDevices
```

1.4.3. Warnings and Notes

Important information will be highlighted within the documentation using one of the formats demonstrated below.

This format identifies important information or a tip which is important in understanding either the software or further documentation

This format is used to convey information which is vital to constructing a successful application. This may highlight a feature or requirement which may not be immediately obvious

This format is used to convey serious warnings. If you do not heed the instructions in these boxes you may risk serious application errors and possible damage to the connected instrument.

This format is used to indicate a potential compatibility issue or information on future features being considered that may affect application designs.

2. What's New in This Version

The following sections highlight any significant changes between the latest version of the assembly (to which this document applies) and the previous assembly version. Please see Section 20 for a complete and detailed project revision history

2.1. ArraySampleWritter class (Version 1.1.1)

The ArraySampleWriter class was added (in Version 1.1.1) which allows recorded samples to be downloaded into arrays (instead of only .wav files). Please see the xml documentation for information on the public interface.

2.2. SignalGenerator.IsBusy (Version 1.1.2)

All Signal generators now have an IsBusy property which indicate if the signal generator is either

- Scheduled to generate a signal, or

- Generating a signal

This property helps applications determine if a call to Generate() will fail without needing to deal with Notifications

2.3. AI-7280 Expanded DC Feed Support (Version 1.1.8)

aiDevices now supports the expanded DC Feed option which may be installed on AI-7280 models which allows the DC voltage and current to range up to 105 Volts and 105 milliamps respectively.

3. Project Roadmap

The aiDevices framework is by no means a static project. At Advent Instruments we are constantly working to improve our software and hardware products and integrate new features; largely in response to customer feedback. As a result aiDevices will continue to change and expand over time, and while we cannot anticipate every possible feature, the following list outlines some features we plan to support in future releases:

- Support for the AI-5120 Telephone Line Monitor instrument

- Limited access to instrument flash file systems

- Support for abstract signal generation sequences

- Support for scripting

- Support for saving and restoring instrument states

- Native support for DTMF Caller ID, NTT Caller ID

- Support for SMS messages

- Support for FSK dropouts and other impairment generation

- Support for custom filters

- Support for AI-5620 recording and playback

- Support for AI-7280 playback

If you require a particular feature which is not addressed by aiDevices or have any suggestions to improve the existing software please do not hesitate to contact us at www.adventinstruments.com or any means listed in the Technical Support section.

4. Installation and Dependencies

4.1. Microsoft .Net Framework 3.5

The aiDevices framework is based on the Microsoft .Net Framework 3.5 which is available for download from Microsoft's website. This framework must be installed before any of the assemblies or example programs will function.

4.2. USB Drivers

USB drivers may need to be installed in order to communicate with Advent Instruments hardware products. Please refer to the product specific documentation listed in section 1.2 for proper driver installation procedures.

4.3. Firmware Versions

Each device class within the aiDevices project may require a minimum firmware version to be installed in the associated instrument. Each Advent Instruments product can be easily upgraded to the latest firmware version using the firmware update utility which can be downloaded from www.adventinstruments.com. Please refer to the product specific documentation listed in section 1.2 for proper firmware update procedures.

4.4. Files Included

The following files are included with the aiDevices framework:

➢ **aiDevices.dll** – this is the primary assembly which contains all the device classes and supporting classes and should be referenced by applications.

➢ **aiDevices.xml** – this file contains the API documentation which will appear automatically within Visual Studio. Application developers should place this file in the same directory as the assembly in order to view the documentation within Visual Studio.

5. Versioning and Compatibility

The .Net framework defines two separate version numbers which are included in each assembly.

- **Assembly Version** – this version is actually used by the common language runtime to determine the "correct" version to link to a particular application

- **File Version** – this version is not used by the language runtime but is visible in the file properties through the Windows file properties.

Each version is composed of four numbers in the format

> **<Major>.<Minor>.<Build>.<Revision>**

The aiDevices project will

1. Zero the revision number.

2. Synchronize the Major and Minor versions for all releases so the assembly version can be determined from Windows file properties.

3. Increment the Build number of the File Version for each "transparent" change that does not require dependent applications to be recompiled (such as a bug fix)

4. Increment the assembly version for changes which require dependent applications to be re-compiled.

5. Increment the major version for changes which modify framework features or integrate a significant number of new features.

Application developers are urged to recompile each dependent project when upgrading to a new version of aiDevices.

6. XML Documentation

The number of classes, methods and properties within aiDevices are far too numerous to make an exhaustive API reference manual economical. Instead documentation for each class, member, function, and property is available through the XML documentation feature in Visual Studio. This documentation has been carefully maintained during the development process and will track all changes to the project. The XML documentation is distributed along with the assembly in the file aiDevices.xml.

> To enable XML documentation place the aiDevices.xml file is in the same directory as the aiDevices assembly referenced by your project. Also be sure to update this xml file whenever updating the assembly to ensure the documentation is correct.

When selecting properties or methods using auto-complete a description of the method or property will appear as shown below.
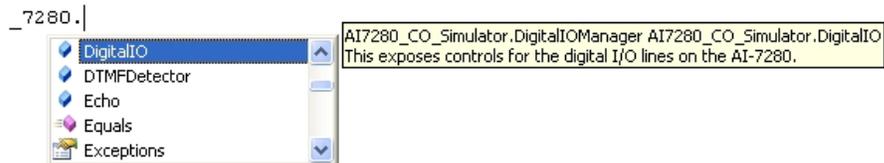
Descriptions for each function argument will also appear as you type which give you an idea of their proper usage.
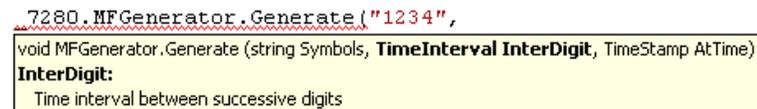
Descriptions of variables and classes will also appear when you mouse over variables or code segments as shown below.

The XML documentation also appears in the Object Browser window which is a very handy tool to browse all the available features of an assembly.

7. Programming Guidelines

The following sections highlight some general requirements and expected usages which are not always obvious from the source code or API documentation. Proper understanding of these requirements is essential in order to develop robust applications

7.1. Immutable Descriptor Objects

Many of the classes within the framework are used as "descriptors" in that they are a placeholder which represents a particular entity or configuration. Some of these classes include:

- Signal definitions documented (see section 14)

- Quantity and measure definitions (see section 13.1)

- Communications medium descriptors (see section 8.5)

Unless specifically stated to the contrary, all descriptor objects are "immutable" which means that their members cannot be modified after they are created. At first glance this may seem an imposing restriction however one must consider the consequences imposed by multi-threading and multiple object references. By enforcing this simple restriction each descriptor can be safely referenced by multiple classes simultaneously (possibly on separate threads) or used as a base class without concern for member values changing. Typically when a different descriptor is required it is straight forward to create another either through a constructor or operator.

7.2. ToString Overloads

Nearly every class within the aiDevices framework overloads the ToString function and will return a well formatted description of the object or whatever the object represents. For example the SignalLevel class will actually print out the signal level in SI units (i.e. "1.25 dBm"). This behavior can be very handy when debugging, update status displays, or when adding objects to controls like combo boxes which typically display the ToString value.

.

7.3. Thread Safety

Most of the classes within the aiDevices framework are designed from their inception to operate in a multi-threaded environment. However application developers must be very careful to note some of the key limitations within the project with respect to multi-threading. The core communications system within device classes is completely thread-safe and allows low-level communication transactions to be issued from multiple threads simultaneously. This ability allows each device object to interact with the connected instrument in parallel (from worker threads) without interfering with the application software as illustrated below.
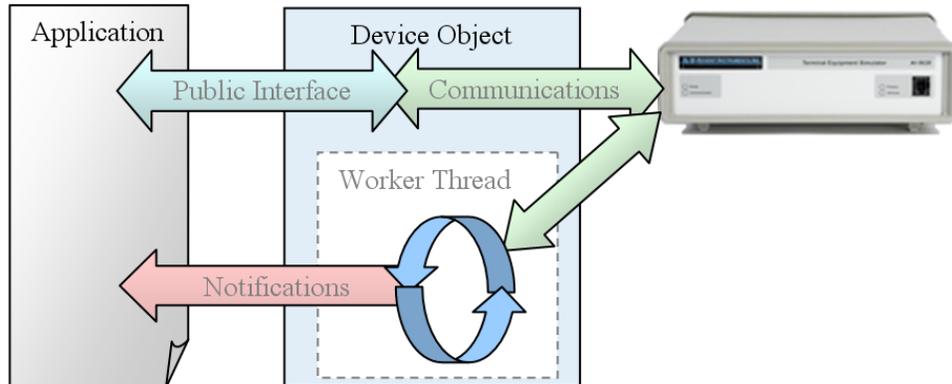
Application developers should note however that **device classes and device support classes themselves are not inherently thread-safe with respect to the parent application** (as most public methods access internal member variables).

Application developers must adhere to the following threading rules and guidelines (unless explicitly specified otherwise).

1. Separate device objects can be created and managed on completely separate threads without conflict. Device objects can be considered to be independent entities and share no common internal references.

2. A device object **may** be accessed on threads other than the one on which it was created; however **the parent application must limit access to exactly one thread at a time.**

3. Notifications (see section 10) are always delivered on a different thread than the one used to create a device object. If applications choose to respond to notifications they must ensure exclusive access to the device object and all supporting objects before accessing any members.

> **Concurrent access of a device object or support objects members may result in erratic operation and/or serious application errors.**

8. Device Class Fundamentals

Each instrument supported by the aiDevices framework is managed by a class with a representative name (i.e. AI7280_CO_Simulator). These classes are referred to collectively as "device classes" (instances are "device objects") within the framework and this documentation. Each device class is responsible for:

- Establishing, managing, and terminating communications with the instrument.

- Automatically synchronizing with the instrument and maintaining the instrument state.

- Automatically retrieving key status information from each instrument which is made available without latency.

- Manage instrument resources to prevent and report any potential conflicts.

- Expose a set of supporting objects which implement the instrument's features and provide the majority of the public interface. Manages ASCII based command/response transactions

- Exposes a communications trace and debugging methods which allow applications to gain visibility into problems or performance issues.

> Device classes have no publicly accessible constructors and can only be created through a successful call to a static Connect method (see section 8.4).

8.1. Common Members

Devices classes make extensive use of inheritance to incorporate common features and this common ancestry also ensures consistent behavior and a uniform interface for basic instrument properties. Most device classes expose the following members

- **Connect** – these **static functions** will search for and connect to instruments and return the corresponding device object (see section 8.4).

- **Close** – this method will immediately terminate communications and release any internal resources associated with communications (see section 8.6.1).

- **CloseAndReset** – this method will immediately termination communications and issue a hardware reset command (see section 8.6.2)

- **DeviceModel** – returns an object which describes the model of the connected instrument.

- **SerialNumber** – returns the serial number of the connected instrument. If the serial number is not known it will return null.

- **UnitID** – returns an object which represents the unique ID number for the connected instrument.

- **Info** - returns basic information regarding the instrument's hardware and firmware versions.

- **FirmwareVersion** – returns a descriptor containing information regarding the instrument's firmware version (if known). If the firmware version could not be determined this will return null/nothing.

- **ConnectedVia** – returns the communication medium descriptor that describes the medium on which the instrument is connected (see section 8.5).

- **IsConnected** – returns true if communications are established with an instrument. (Note: This may return true if communication are established but suspended due to an exception condition)

- **IsCorrupted** – returns true if a serious configuration problem has been detected within the instrument (such as missing or invalid calibration information). If this is the case communications will also typically be halted.

- **IsActive** – returns true so long as communications are established and have not been suspended due to an exception condition. If this returns false a serious problem may have occurred and applications should check the passive exception reporting mechanisms listed in 11.3 or the communications trace to determine the nature of the issue.

- **SupportedBaudRates** – if the instrument associated with the device class is capable of COM port (RS-232) communication then this will return a list of supported baud rates, otherwise this will return an empty list.

- **ChangeToBaudRate** – if the instrument associated with the device class is capable of COM port communications then this method will change the baud rate to the value specified.

- **ChangeToMaximumBaudRate** – If the instrument associated with the device class is capable of COM port communications then this method will change the baud rate to the maximum supported value.

- **Exceptions** – returns a list of exceptions which have been passively reported by the device class (see section 11.3). This list should be checked when exception conditions are detected.

- **DebugTraceFileName** – This is the default file name used by the WriteDebugTraceFile method.

- **WriteDebugTraceFile** – this set of methods will write debug and trace information to a text file which can then be used by Advent Instruments to fix problems or performance issues.

- **Trace** – this exposes the communications trace object through which application developers can insert items into the communications trace which will appear in debugging files and in trace windows (see section 12).

- **NotificationRecipient** – this specifies a delegate to which notifications will be delivered. Great care must be taken when accessing this property (see section 10).

- **ResetToDefaultSettings** – this method will reset all instrument settings back to default settings (see section 8.7).

8.2. Device Support Objects

While each instrument is represented by a device class, these classes do not directly manage many features aside from basic communications and exception handling. Once communications are established, applications will interact almost exclusively through supporting objects which manage the instrument's features and are accessed through read-only properties of the device object (see section 16 for information on common support classes). Each supporting class is typically responsible for managing a single feature of an instrument. For application developers this design yields the following benefits:

- New features can be added to device classes by adding new support objects without affecting any existing code

- Support classes logically separate and group sets of features and functionality. The resulting application code is much easier to read and write.

- Device classes with similar capabilities reuse the exact same support classes (for example nearly every instrument supports tone generation through instances of the ToneGenerator class). This means code written for one device can be easily abstracted or ported to another device with the same support objects.

8.3. Determining Instrument Capabilities

Most device classes implement a Capabilities property which returns an object that reports the collective capabilities of the instrument's hardware, firmware, and the device class. This capability reporting mechanism enables applications to

- Discover the particular abilities of a device object in an abstract manner.

- Automatically take advantage of improvements or changes in instrument functionality

- Avoid "parameter out of range" exceptions.

> The capabilities of hardware, firmware, and the aiDevices software may change with different versions (generally to improve performance). By writing applications which reference this capabilities object they can avoid future "hardcoded" problems when capabilities change!

Example:

```
' Sweep a tone over the maximum supported frequency range
Const NumPoints As Integer = 10
With  7280.Capabilities      ' Access capabilities info
        For i = 0 To NumPoints
                Dim F = i * ((.ToneMinFrequency + .ToneMaxFrequency) / NumPoints)
                7280.ToneA.Frequency = F
        Next
End With
```

Each type of capabilities class will report information in a consistent format for each type of capability being reported. The following sections outline the common capabilities which can be reported. Instrument specific capabilities are documented in the corresponding device class section.

8.3.1. Tone Generator Capabilities

The tone generation capabilities are always reported by capabilities objects using the following properties:

- **ToneLevelMaximum** – reports the maximum possible signal level for each tone generator

- **ToneFrequencyMinimum / ToneFrequencyMaximum** –reports the range of frequencies which can be produced by a tone generator

8.3.2. FSK Generator Capabilities

The FSK generation capabilities are always reported by capabilities objects using the following properties:

- **FSKTransmitterBitsMaximum** – reports the maximum number of consecutive FSK bits which can be transmitted

8.3.3. Noise Generator Capabilities

The white noise generator capabilities are always reported by capabilities objects using the following properties:

- **NoiseLevelMaximum** – reports the maximum possible signal level for the white noise generator

8.3.4. Echo Generator Capabilities

The echo generator capabilities are always reported by capabilities objects using the following properties:

- **EchoDelayMinimum / EchoDelayMaximum** – reports the supported range of echo delays for each tap

- **EchoGainMinimum / EchoGainMaximum** – reports the supported range of echo gains for each tap

- **EchoTapsMaximum** – reports the maximum number of echo taps for the echo generator

8.3.5. Ring Generator Capabilities

The ringing generator capabilities are always reported by capabilities objects using the following properties:

- **RingDCMinimum / RingDCMaximum** – reports the range of acceptable range of DC offsets which can be produced by the generator

- **RingFrequencyMaximum / RingFrequencyMinimum** – reports the range of acceptable frequencies that can be produced by the generator.

- **RingLevelMaximum** – reports the maximum signal level which can be produced by the ringing generator.

8.4. Establishing Communications

Each device class contains a set of static Connect functions which establish communications with the corresponding instrument and returns an instance of the device object which can then be used to control the instrument.

Each Connect function will behave as follows:

- If an instrument is found which is supported by the class and communications are established successfully, then an instance of the device object will be created and returned. This object can then be used to interact with the instrument.

- If no supported instruments are found then null/nothing is returned.

- If an instrument is found but communications are not established correctly or the instrument is not supported by the software then the function will raise an Exception which must be handled by the calling application.

The aiDevice base class contains a special variant of the Connect functions in that

- Each aiDevice Connect function will connect to **any supported instrument** and return an instance of the handling class. This is especially useful when attempting to connect to multiple types of instruments without regard to order.

Each device class defines three variants of the static Connect function.

- **.Connect()** – this will connect to any instrument which is supported by the device class.

- **.Connect(String SerialNumber)** – this will connect to a device supported by the class name with the serial number specified.

- **.Connect(CommunicationMediumDescriptor m)** – this will connect to any available device supported by the device class which is connected on the communication medium specified by the descriptor.

Examples:

```
//Connect to any avilable AI-7280
AI7280_CO_Simulator CO = AI7280_CO_Simulator.Connect();

//Connect to the AI-5620 with a specific serial number
AI5620_TE_Simulator TE = AI5620_TE_Simulator.Connect("SN140059");

// Connect to any device connected on COM 1
aiDevice Dev = aiDevice.Connect(new ComPortDescriptor("COM1"));
```

8.5. Discovering Communication Mediums

Each device class exposes a static function called AvailableCommunicationsMediums that returns a list of descriptors which correspond to the available communication mediums compatible with that particular device class at the instant when the function was called. The aiDevice base also class exposes a special version of this function which returns the list of descriptors that are compatible with **any** supported instruments. Each communication medium descriptor derives from the CommunicationMediumDescriptor class and occurs in two types:

- **USBDescriptor** – describes an available instrument connected on USB. Note: the serial number and model description are also reported by this class.

- **ComPortDescriptor** – describes an available COM (RS-232) port on the host computer.

These descriptor objects can then be supplied to the static Connect methods for device classes to attempt to establish communications on that medium.

Example:

```csharp
// search through each available communication medium
foreach (CommunicationMediumDescriptor m in
                    aiDevice.AvailableCommunicationMediums)
{
      try
      {       // attempt to connect to any instrument
              if ((Dev = aiDevice.Connect(m)) != null) break;
      }
      catch (Exception ex)
      {
              MessageBox.Show("Error while connecting:" + ex.Message);
      }
      if (Dev != null)
              MessageBox.Show("Connected to " + Dev.ToString());
      else
              MessageBox.Show("No instruments found");

}
```

8.6. Terminating Communications

Once an application has finished using a device object it **must** call one of the Close methods before setting the object variable to null. The Close methods will terminate communications immediately and release internal resources within the device object.

> **Applications must call one of the Close methods before setting a device object variable to null!** Failing to call a close method will cause the device object to not shut down properly and will not release communication resources associated with the instrument until your application is terminated!

After communications are terminated through any of the Close methods the instrument will no longer be connected to the device object and cannot affect the instrument behavior. However please note:

- The communications trace will remain until the object is actually destroyed which allows for debugging after communications are terminated (see section 12).

- Many device object properties may still be accessed but will return default values

8.6.1. Close

Each device class contains a Close method which can be used to terminate communications. This method will:

- Release any internal resources within the device object and supporting objects.

- Terminate any automated behavior managed by the device object (tone patterns, background tasks).

- Terminate communications with the instrument.

After the Close method is called communications can generally be immediately re-established using a Connect method (see section 8.4).

> The Close method will terminate automated behaviors but **will not affect "static" instrument settings and behaviors** (such as telephone interface settings, basic tone generators, and signal routing) in order to minimize the impact of Close on the behavior of the instrument. Please note:
> - Instruments settings can be configured with particular signal routing and telephone interface settings (possibly required for a particular test configuration) which will remain in place after Close is called.
> - When a device object reconnects to the instrument it will synchronize with the instrument and appear with most of the same settings that were present when Close was called. This will prevent transient conditions on connection.
> - If desired, the unit can be reset to default settings before close by calling ResetToDefaults before Close or alternatively calling CloseAndReset

8.6.2. CloseAndReset

Each device class exposes a CloseAndReset method which terminates communications with the associated instrument in the same manner as the Close manner except:

- Just prior to terminating communications this method will issue a command which will cause the connected instrument to perform a hardware reset that will ensure the hardware and firmware are restored to the initial power-on state.

The hardware reset command issued by the CloseAndReset method has the following effects:
- The hardware reset cycle **will take a significant amount of time to complete** during which communications may not be able to be reestablished.
- The hardware reset cycle may cause a reset of the instrument's USB controller which may cause the device to "disappear" from the list of available devices until the reset is complete and the host computer can re-enumerate the instrument

If these behaviors are undesirable please consider using the Close method.

8.7. Resetting to Default Settings

In many applications it is desirable to reset the instrument settings to defaults in order to return the device to a known operating condition. Each device object implements a ResetToDefaultSettings method which will generally:

- Call the ResetToDefaults method of each support object. Please refer to each particular support class specification for details.

- Stop all active signal generators and reset all generator settings to nominal defaults

- Reset all detector settings to nominal defaults

- Reset all telephone interface settings to defaults.

- Reset all digital outputs to "Output Low" and disable all special functions

- Reset all signal routing, measurement settings, and filters to defaults

- Reset protection mechanisms within the instrument.

This ResetToDefaultSettings **does not initiate a hardware reset of the instrument** but rather reconfigures the instruments with "safe" default values.

Most support objects also have a ResetToDefaults method which only affects the settings relevant to one feature. This can be very helpful when needing to reset only certain instrument features to defaults.

9. Instrument Time Management

Each device object is responsible for establishing and maintaining a relative time base with the connected instrument. This time base is then used:

- To record and report timing of events within the instrument or the device object (such as signals being detected or generators starting/stopping)

- To schedule future actions

- To compare events reported by instruments to the host time or events reported by other instruments.

In general, each device time base operates as follows:

- When communications are first established with an instrument

  o A timer is simultaneously reset within the instrument and the device object.

  o The host time (wall time) is recorded.

  o This instant in time is referred to as the device time **epoch** and corresponds to device time of zero.

- All time information reported by the instrument is reported to the device object in seconds relative to epoch. The device object then reports timing information to the application using TimeStamp objects which report both the instrument time and corresponding host time.

- The device object is responsible for maintaining synchronization between its internal timer and the device timer and will periodically adjust for any slippage.

A simplified illustration of the time base system is show below.
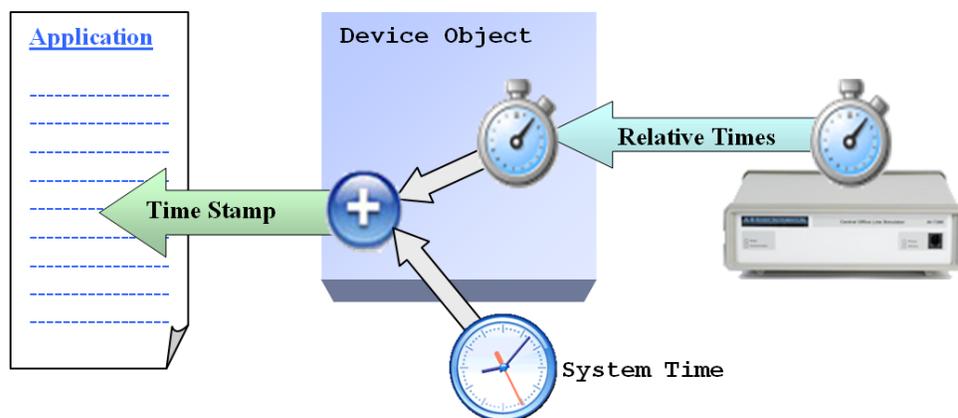
Figure 1 Time Base Structure

9.1. Time Stamps and Timing Calculations

All descriptions of a particular instant in time are managed by the TimeStamp class. Each time stamp object exposes two properties that report timing information:

- **DeviceTime** –reports the instant in time in seconds relative to the device time epoch.

- **HostTime** –reports the instant in time corresponding to the host time as a DateTime structure.

- **ToString** – returns an easy-to-read device time format (i.e. "1m 6s 72ms")

An illustration of the relationship between the host time, device time, time base, and TimeStamp objects are illustrated below.
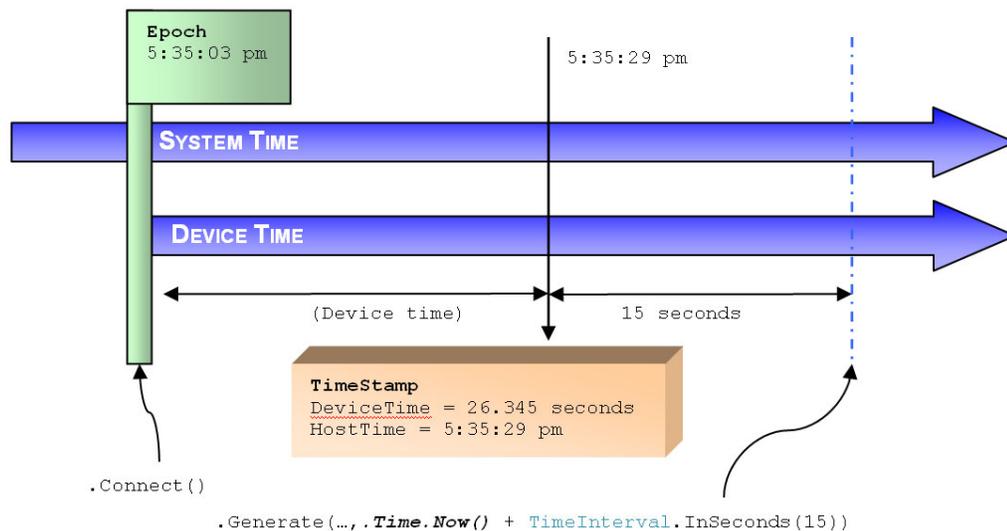
Figure 2 Device Time and Time Stamps

TimeStamp objects are primarily used in Notifications (section 10) and as arguments in methods that support scheduling. The TimeStamp class also defines a set of operators which make accurate timing calculations and comparisons very easy to implement.

> All operations and comparisons using TimeStamp objects are abstracted such that even time stamps reported from different devices can be compared against each other.

Example:

```vb
Dim Time1 As TimeStamp ' These must be created by aiDevices
Dim Time2 As TimeStamp

' Difference between time stamps
Dim Diff As TimeInterval = Time1 - Time2

' Can calcaulate other times by adding time intervals

Dim NewTime As TimeStamp = Time1 + TimeInterval.InSeconds(4)
```

10. Notifications

Applications are able to configure and control instruments by means of methods and properties exposed through the device object and its cast of supporting objects. The notification system "closes the loop" by providing a mechanism to asynchronously report information from the instrument back to the higher level application in a timely manner.
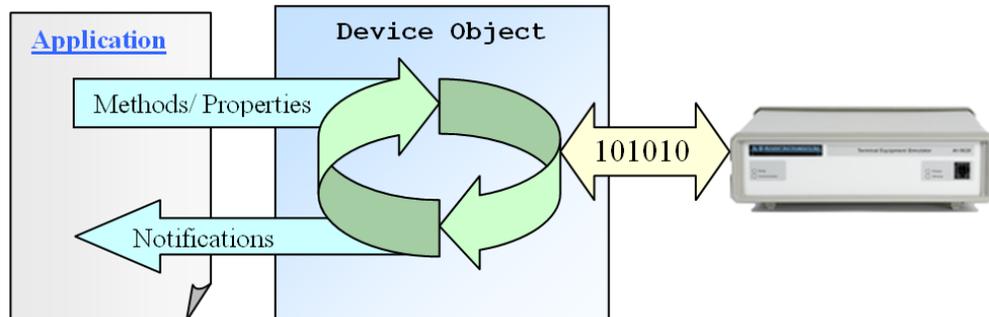
The term Notification is used within aiDevices to avoid confusion with events which have their own meaning and usage within the .Net framework.

All asynchronous information is delivered from a device object to the application by means of Notification objects. These notifications can convey information such as:

- Signals being detected (DTMF, CAS, FSK, etc)

- Actions that have occurred (signal generation complete, etc)

- Measurement information

- Exceptions that have occurred

This notification system

- Minimizes polling and other wasteful programming structures by delivering information as soon as it's available

- Enables applications to respond to detected signals and conditions with minimal latency.

- Enables applications to be designed using an Event Driven Architecture (EDA) which simplifies the code to simulate behaviors that may not be easily realized in a top-down program structure.

It is important to note that notifications are reported by an instrument **as soon as the corresponding event or information is available**. Please note:
- Notifications from different detectors may appear out of temporal order with respect to each other due to detection delays. For example: a DTMF digit may be reported slightly after a metering pulse although they may have actually occurred in the opposite order. The actual order of notifications should always be determined by comparing the reported TimeStamp information if available.
- In general, notifications of signals of a similar signal type are reported in temporal order
- Due to communication and processing delays, each notification may be slightly delayed with respect to the event or information which caused it.

Please note that these notifications can be handled in a more simplistic manner depending on the application requirements:
- To wait for a particular notification see section 16.3
- To record detected signals see section 16.4.

10.1. Notification Objects

Notification objects can be thought of as "memos" passed by the device object to a parent application to inform it of important information. All notifications classes have descriptive names which convey their meaning (similar the subject line of a memo) and end with 'Notification' (For example: SignalDetectedNotification informs an application that a signal has been detected). Notifications are intended to be informative in nature and do not require any particular action on the part of handling applications. Applications may choose to:

- Monitor notifications to detect particular sequences

- Detect particular notifications and take some action

- Ignore notifications altogether

Notification classes are immutable and organized by meaning and function through their inheritance pattern. All notification objects are derived from the Notification abstract base class and can thus be handled by the device notification system. This base class exposes a signal Sender member which references the device object from which the notification originated.

10.1.1. Exception Notification

The ExceptionNotification class is used to report internal exceptions to applications in a timely manner. This class exposes the following members:

- **Exception** – returns the exception which has occurred

Once communications are established, exception notifications can be generated at nearly any time. See section 11 for more details on what conditions can cause these notifications to be reported. Depending on the severity of the exception the device object may no longer be able to communicate with the instrument!

10.1.2. Signal Started Notification

The SignalStartedNotification class is used to inform an application that the **beginning** of a signal has been detected. This class exposes the following members:

- **Time** – returns a TimeStamp indicating the time at which the start of the signal was detected

- **Signal** –returns an ISignal descriptor (see section 14)

This notification is useful when applications need to respond to signals which may be very long in duration but cannot wait until the end of the signal is detected.

> The SignalStartedNotification object indicates that the **start** of a particular signal has been detected. However:
> - These notifications are **not guaranteed to occur** for all signals – especially for those of very short duration.
> - These notifications convey preliminary information about the signal in question and disregard information such as duration (since it is generally not known).
> - If a SignalStartedNotification is generated, a SignalDetectedNotification will occur at a later time with an updated signal description that will contain additional information such as duration.

10.1.3. Signal Detected Notification

The SignalDetectedNotification class is used to inform an application that a particular signal has been detected and has **completed**. This class exposes the following members:

- **Time** – returns a TimeStamp indicating the time at which the start of the signal was detected

- **Signal** –returns an IDetectedSignal descriptor (see section 14)

This notification is the primary mechanism for reporting information to applications from the signal detection features of instruments.

> SignalDetectedNotification objects convey that a signal has been **detected and has completed**. However:
> - These notifications are only generated **after the detected signal has finished**.
> - Signals of long duration **may** be preceded by a SignalStartedNotification which conveys preliminary information regarding when the signal started and the nature of the signal detected.
> - Each fully detected signal is **guaranteed** to generate exactly one SignalDetectedNotification which contains a complete set of measurements, timing, and details related to the signal which was detected.

10.1.4. Action Notification

In addition to signals that have been detected by the instrument, it is often crucial for an application to be aware of the nature and timing of actions being performed by the instrument itself (whether initiated directly or as a result of automated behavior). In general such actions can include:

- Activation of a signal generator

- Deactivation of signal generator

- Change in telephone interface hook-switch or connection status

These notifications can be very useful to

- Schedule signals for transmission based on the relative timing of prior signals

- Monitor the current state of signal generation within the instrument

- Verify the timing of signals being transmitted

The ActionNotification class contains the following members:

- **Time** – returns a TimeStamp indicating the instant at which the action occurred.

- **Action** – returns an enumerated value which indicates the particular action that occurred. Each enumerated value is descriptively named. Please see the relevant XML documentation for more information.

Applications can also wait for ActionNotifications using the WaitManager (see section 16.3) in a much simpler manner.

10.1.5. Protection Notification

ProtectionNotification objects are used to inform an application that an instrument has detected a hazardous condition and has taken corrective action to prevent physical damage to the instrument.

Upon reception of a ProtectionNotification object an application can assume that the **behavior of the instrument may not reflect the current settings or state expected by the application program** however the specific differences depend on the protection mechanism that was invoked. These protection mechanisms can include:

- Disconnection of the telephone interface circuitry or terminations
- Disconnection of ringing loads
- Disconnection of AC termination circuitry

Each protection mechanism may make temporary changes to the instrument state which will reset automatically while others may **make long-lasting changes that must be reset by the application software**. Please see the device specific documentation for specific details.

Regardless of the specific protection mechanism, the ProtectionNotification class exposes the following properties which can be used by the application to determine the nature of the error.

- **Message** – returns a description of the hazardous condition that was detected and which action has been taken to protect the device.

- **Action** – returns an enumerated value which indicates the nature of the protective action taken by the instrument (Note: not all devices will support all actions listed). The supported values are:

    o **Telephone_Interface_Disconnected** – indicates the telephone interface of the instrument has been disconnected internally

    o **AC_Terminations_Disconnected** – indicates that the AC termination circuitry internal to the device has been disconnected from the telephone interface

    o **Ringing_Loads_Disconnected** – indicates that ringing loads have been disconnected from the telephone interface

- **Persistence** – returns an enumerated value that describes how long the protection mechanism will persist, and whether action must be taken to reset the protection mechanism. The supported values are:

    o **Temporary** – indicates that the protection mechanism is only temporary and will be automatically reset after a suitable amount of time has elapsed.

    o **Conditional** – indicates that the protection mechanism will be engaged so long as the hazardous condition persists. The mechanism will automatically disengage once this condition is removed.

    o **Fixed** – indicates that the protection mechanism will remain engaged until the application takes application to reset this condition.

10.2. Handling Notifications

Each device object within the aiDevices framework implements a very simple notification delivery mechanism using a single delegate. Applications must "register" for notification delivery by supplying a compatible function to a delegate property available in each device object. Application code will appear in the following form:

> ***DeviceObject.*** NotificationRecipient = ***AddressOfHandler***

Notifications are <u>always</u> delivered on a thread other than the one on which the device object was created. Application designers must take suitable precautions to avoid race conditions or threading conflicts.

Notification information can be handled in a more simplistic manner depending on the application requirements:
- To wait for a particular notification see section   16.3
- To record detected signals see section 16.4.

The following example demonstrates how an application can be structured which can receive notifications and respond with some programmed action. In the following example the AI-7280 CO simulator waits until the telephone line goes off hook and then generates a tone. **Note: This program assumes that only the OnNotification method will access the _7280 object which prevents thread conflicts.**

Example:

```csharp
class Simulator
{
  AI7280_CO_Simulator _7280 = null;

  // Constructor
  public Simulator(){}

  // Connects to AI-7280 and registers for notifications
  public void Start()
  {
      Close();
      // Connect to any available AI7280 CO simulator
      _7280 = AI7280_CO_Simulator.Connect();
      if (_7280 == null) return;
      _7280.ResetToDefaultSettings();
      _7280.NotificationRecipient = OnNotification;
  }
  // Disconnects from AI-7280
  public void Close()
  {
      if (_7280 != null) _7280.Close();
      _7280 = null;
  }
  // Continued on next page...
```

```csharp
    // Called by _7280 object to deliver notifications
    private void OnNotification(Notification N)
    {
        //----------------------------------------
        // Filter out only line state changes
        //----------------------------------------
        SignalNotification S = N as SignalDetectedNotification;
        if (S == null) return;
        LineStateChange LS = S.Signal as LineStateChange;
        if (LS == null) return;

        switch (LS.State)
        {
            case Telephone_Line_State.Off_Hook:
                // turn on a 440 Hz tone when off hook
                _7280.ToneA.Update(new Tone(SignalLevel.IndBV(0),
                                            Frequency.InHz(440)));
                _7280.ToneA.Generate();
                break;
            case Telephone_Line_State.On_Hook:
                _7280.ToneA.StopGenerator();
                break;
        }
    }
}
```

11. Exception Handling

While not desirable, exception conditions may occur within the classes in the aiDevices framework. The following sections document the general exception handling behavior built into aiDevices which can be leveraged to create robust fault-tolerant applications. Exceptions can occur within aiDevices for a variety of different reasons and are broken into categories and discussed in the following sub-sections.

11.1. aiDeviceException

All Exception classes which relate to instrument operation derive from the aiDeviceException base class and can be reported through each of exception reporting mechanisms described in section 11.3. This class defines two severities of exceptions which are reported through the Severity property of aiDeviceException:

- **ExceptionSeverity.Critical** – indicates that an error condition occurred from which recovery was not possible. In general this means that communications with the instrument is suspended and further operations are not possible. Applications should call the Close method of the device object as soon as possible.

- **ExceptionSeverity.Minor** – indicates that an error occurred from which recovery was possible. This type of exception is used to report conditions which could only slightly impair performance.

11.2. Automatic Communication Suspension

Whenever a serious error is detected within a device object further communications with the instrument are suspended; but not terminated. Such error conditions can include communication errors, instrument file system errors, or unexpected internal exceptions. It is important to note the general device class behavior when such an error occurs:

1. Further communications with the instrument are suspended (commands will not be sent) although (if possible) the **communication medium is kept open**. This ensures that the communications trace remains valid and can be used for debugging the exception condition. **<u>Applications still must call Close before setting the device object to null!</u>**

2. The exception is reported using each of the passive reporting mechanisms discussed in the section 11.3.

3. The IsActive property of the device object returns false which can be detected by parent applications (if polling is desirable)

4. Each property or function which communicates with the instrument will return **default values**.

11.3. Passive Exception Reporting Mechanisms

Not all exceptions with the aiDevices assembly can or should be thrown in a manner that can be handled using try-catch statements. Possible conditions where conventional throw/catch mechanisms are not appropriate include:

- Exceptions which occur in worker threads which are not accessible to the calling application. Throwing exceptions in this circumstance could be fatal to the entire project (which is obviously not desirable).

- Exceptions which are noteworthy and should be reported to the user, but so minor they should not interrupt the flow of execution.

- Exceptions which report that normal functionality may be impaired, but not completely.

The aiDevices framework supports two independent methods for passive exception reporting:

- All device classes contain an **Exceptions property** which maintains a thread-safe list of all Exceptions which have been reported by the application. The calling application can access this list at any time to check for passively reported exceptions.

- The **Notification** system (see section 10) can deliver ExceptionNotification objects which inform applications of exception conditions.

Example:
```
// aiDevice Dev;


// Check the list of exceptions in the device dev
foreach (aiDeviceException ex in Dev.Exceptions)
{
    switch (ex.Severity)
    {
        case ExceptionSeverity.Critical:
            // Something really bad happened!
            break;
        case ExceptionSeverity.Minor:
            // Worthy of note
            break;
        default:
            break;
    }
}
```

11.4. Exception Conditions and Behaviors

11.4.1. Invalid Arguments

In some circumstances, an application may pass an invalid argument to an aiDevices class. When such invalid arguments are detected

- An Exception is raised to alert the calling application of the offending argument and possibly indicate the range of acceptable arguments.

- The exception is **not** reported using a passive mechanism since applications can and should easily recover from such errors.

When invalid arguments are detected the following exceptions may be thrown:

- System.ArgumentNullException

- System.NullReferenceException

- System.IO.InvalidDataException

- Advent.aiDevices.ArgumentToHighException

- Advent.aiDevices.ArgumentToLowException

> Wherever possible applications should reference limits reported by the Capabilities object returned from device objects to avoid argument exceptions (see section 8.3).

11.4.2. Resource Conflicts

Each device object and each device support object is responsible for managing, reserving, and releasing instrument resources to prevent erratic operation which may result from multiple objects competing for system resources. Whenever applications or support objects attempt to concurrently access a resource

1. A ResourceConflictException will be thrown

2. The Exception will **not** be reported using a passive mechanism since applications can easily recover from such errors.

When resource conflicts are detected the following exceptions may be thrown

- Advent.aiDevices.ResourceConflictException

11.4.3. Unsupported Features

If an application requests a feature which is not supported by an instrument or the device class

1. A NotSupportedException will be thrown

2. The exceptions will **not** be reported using a passive mechanism since applications can easily recover from such errors.

When an unsupported feature is requested the following exceptions may be thrown

- System. NotSupportedException

11.4.4. Communication Errors

Communication errors can occur in many different forms for many reasons. Such reasons can include:

- A cable is disconnected or instrument power is lost

- A response from an instrument is malformed or not understood

- An instrument did not respond to a command within a prescribed timeout

Regardless of the source, all serious communication errors result in the same behavior.

1. The exception is reported using each of the passive reporting mechanisms (see section 11.3)

2. Communications with the instrument are suspended. When communications are suspended the device object's **.**IsActive property returns false.

3. All further accesses of methods or properties of supporting objects will return default values or will be ignored.

11.4.5. Unexpected Errors

Inevitably some exceptions may occur due to bugs or rare conditions which were not anticipated. Depending on the location of the exception, the exception will either be handled in an identical manner as Communication Errors, or Invalid Arguments. If any such error occurs in your application please do not hesitate to contact technical support at www.adventinstruments.com or any method listed in the Technical Support section and we will quickly resolve the issue.

12. Debugging and Tracing

As applications grow in complexity often times it becomes exceedingly difficult to track down the exact source of errors or performance problems. Without visibility into the inner workings of each software component the task may verge on impossible. Each device class within the aiDevices framework exposes a set of methods and objects which assist application developers in debugging and tracing. More specifically:

- A public trace interface which allow developers to insert their own comments into the internal communications trace. These comments will then appear graphically in the debug form and in trace files

- A debugging form which can be created independently for each device object which can:

    o Display general device information and communications statistics

    o Display a live graphical representation of the communications trace so the developer can watch his/her comments inline with the communications between the instrument and the device class

- A mechanism for writing a debug trace file containing the state of the entire device object and the communications trace. This file can be sent to Advent Instrument technical support to resolve complicated problems encountered by developers.

12.1. Communication Trace

A trace object is maintained within each device object which keeps a short list of recent communications and other important information as illustrated in Figure 3.This trace will:

- Record all communications to and from the connected instrument

- Record all serious exceptions reported by the device

- Record comments inserted by the device object which report important information

- Record all notifications generated by the device object regardless of whether the application is notified.

- Record comments added by the parent application
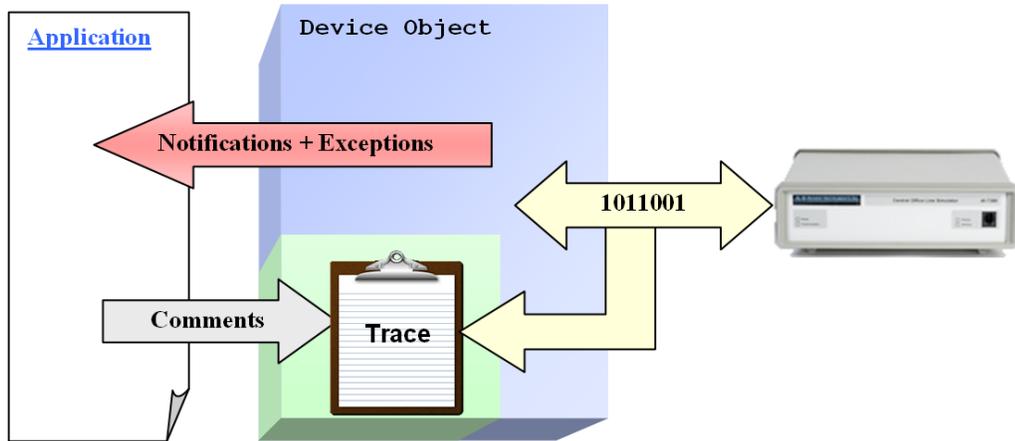
Figure 3Communications Trace Structure

Each device class exposes the trace features through the Trace property which can be accessed by applications to insert customized comments into the trace. This trace object exposes the following interface:

- **Add** – this method allows application developers to add their own custom comments to the internal communication trace which will appear in the debug trace files and graphically in the debug window.

- **IsFrozen** – when set to true, the internal communications trace is frozen such that no new items can be added. This may become useful when application developers need visibility for a particular point in time but do not want to stop the current execution of the project.

Example:

```
// This will add the comment directly to the trace
_7280.Trace.Add("Starting ringing");

//Generate a ringing burst and wait until it's done
_7280.Ring.Generate(RingBurst);
_7280.Wait.Until(ActionType.Ringing_Pattern_Stopped,
                 TimeInterval.InSeconds(5));

_7280.Trace.Add("Ringing Complete");
```

The contents of the trace can be made visible in several ways:

1. The application can write a debug trace file which contains the contents of the trace along with internal state information (see section 12.3).

2. The application can create a debug form for the device object which contains a tab which displays the live contents of the trace as the application executes (see section 12.2).

12.2. Debug Forms

When developing test and simulation applications with custom components it is often essential to obtain deep visibility into the internal workings of the software without complicated and unnecessary inline debugging code. To this end each device object exposes the CreateDebugForm function which will create and return a new form containing controls which display instrument specific debugging information. Please note:

- The form is not shown by default. Applications will need to call the .Show method to make it visible.

- These forms can be created on a thread other than the one on which the device object was created. This ensures that the window does not "freeze" when your application is busy.

Once created these forms will:

- Automatically update and display the state of the device object including all communications trace information and applicable instrument states.

- Persist after the device object has terminated communications in order to display the final state of the device object.

Example:

```
// Construct a debug form for this device but do not show it
Form DebugForm = _7280.CreateDebugForm();

// display this form to the user
DebugForm.Show();
```

12.2.1. Communications Trace Display

Each debug form contains a tab which displays the live contents of the communications trace in a manner which is simple to navigate:

- Items within the trace are color coded to identify their respective meanings

- Where possible communications are displayed with their high-level meaning or property name to make the contents easy to read.

- The display can be filtered to exclude items which may not be of interest (like status update communications which are very frequent)

An example of the communications trace display is shown in Figure 4.

The trace can be frozen manually by clicking here

The display of the trace can be filtered to reduce clutter

The display format for each item can be selected here

Each item is color coded with exceptions appearing in red

Items in the trace are shown in order from most recent to least recent
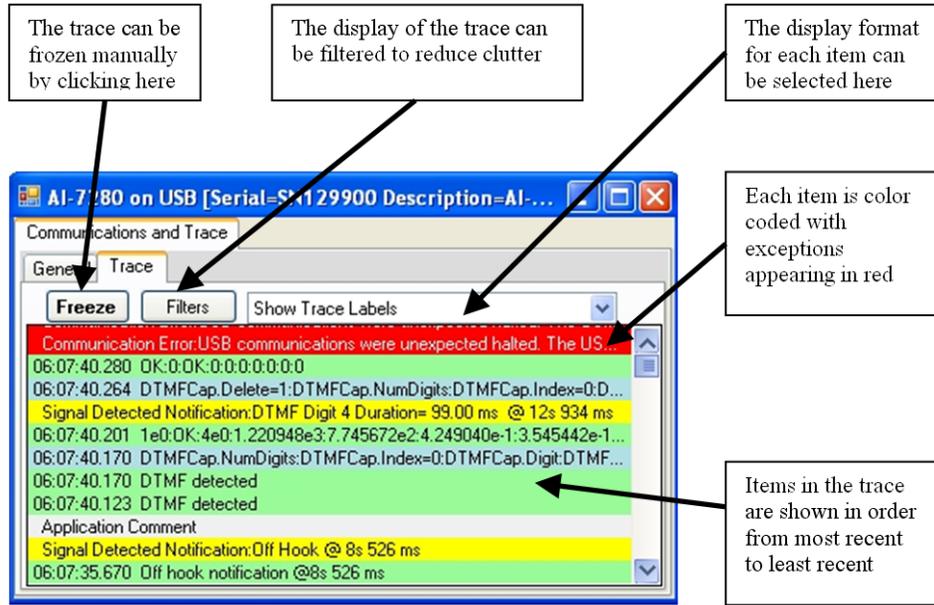
Figure 4 Debug Form Communication Trace

The contents of the trace are displayed graphically within the trace from top to bottom in order from most recent to least recent. As items are added to the trace the contents will automatically update. The format of the trace window is color coded to separate elements in the trace by type:

- Comments are generally not highlighted and appear with a **white** background

- All communications sent to the instrument are highlighted in **light blue**

- All communications received from the instrument are highlighted in **light green**

- Exceptions are highlighted in **red** to make them stand out.

- Notifications are highlighted in **yellow**

Since even normal communications between a device object and an instrument can involve a significant amount of data transfer, the trace display offers a set of simple filters to ensure the display is easy to read while debugging. These filter settings are annotated in Figure 5.

Displays all responses from the instrument

Shows all commands sent to the instrument

Shows/hides all polling or status update communications which may be quite frequent

Enables display of the actual byte values which are written to and read from the instrument. This filter may not be useful for most developers.

Enables display of all packet based communications to and from the device.

Figure 5Communication Trace Filters

12.3. Debug Trace Files

While not desirable or intended sometimes problems may arise within the framework which are very difficult to debug without detailed information. For this reason each device class can create a detailed text file containing the current state of the device object and the communications trace. Each device object exposes the following methods:

- **DebugTraceFileName** – this specifies the default filename for debug trace files written by the object. This can be assigned immediately once communications are established to ensure that any future calls to WriteDebugTraceFile will write to this location.

- **WriteDebugTraceFile** – this method immediately writes a debug trace file based on the current device object state. If a specific filename is not supplied, then the default filename will be used which is specified by the DebugTraceFileName property.

> When contacting technical support for assistance with a particular problem with a device object contained within the aiDevices framework please email a copy of this debug trace file to technical support to the email address listed in section 21.

13. Descriptor Classes

The aiDevices framework contains an assortment of descriptor classes which help manage details related to signal generation, measurements, signal representations, etc.

13.1. Quantities, Units, and Measures

Representations of different physical quantities, measures, and unit systems are required to perform may common tasks such as

- Synthesizing signals

- Assigning telephone interface parameters

- Returning measurements

All physical quantities, units, and measures within the aiDevices framework are managed using descriptor classes with representative names. Each of these classes

- Manage different representations and unit systems for physical measures

- Support creation of quantities in different scales (i.e. Amps, milliamps, etc)

- Support conversions between each unit system or representation

- Overload the ToString function to return a "display friendly" string which represents the described quantity. For example "2.13 mA" instead of 0.00213

- Support common operations (addition, subtraction, multiplication, etc) which allow applications to perform abstract calculations without regard to unit systems or internal representation. For example: a developer could write a level sweep algorithm without needing to know if the levels are specified in Vrms, dBV, or dBm.

These descriptor classes

- Result in more readable code since class names and methods are much more descriptive than double, int, and float.

- Result in more portable, abstract, and stable code since incompatible operands or operations can be detected by the compiler

Each class which represents a quantity or measure follows a common design pattern outlined below:

- Most descriptors are created using static functions which are named to specify the representation of the quantity being created. In general your code will look somewhat like the following:

    ClassName.In***UnitsAndScale***(Value)

    where ***UnitsAndScale*** will describe of the units and scale of the definition. There also may be more than one creation function if the physical quantity has multiple representations or scales. For example, each of the following defines a frequency of 1000 Hz

    Frequency.InHz(1000)
    Frequency.InkHz(1)

- Each descriptor object is immutable (meaning the internal values cannot be changed once created). To create a new physical quantity representation you must create a new object through a creation function, operator, or conversion function.

- If multiple representations of a quantity are possible the object will expose one or more conversion functions in the form

    ObjectVariable.To***AlternateRepresentation***

    where ***AlternateRepresentation*** describes the representation of the object which is returned. For example Level.TodBV expresses that the representation within the variable Level will be returned in a dBV unit system.

- Each object can be compared natively using the standard comparison operators ($>$, $>=$, $<$, $<=$, $==$, $!=$) for objects of the same class. Attempting to compare objects of incompatible class (for example Frequency and Voltage) will result in an exception.

- Where logical, arithmetic operators are defined for quantities which make calculations easier. For example: you can add and subtract Frequency objects to calculate new values.

- Each class may define a set of static or instance methods which assist in common conversions or calculations. For example SignalLevel helps with twist calculations.

- Each object returns the numeric value of a quantity in its current representation by means of the read-only Value property.

- If multiple scales are possible, the numeric value of a quantity will be returned through read-only properties in the form

    ObjectVariable.ValueIn***Represenation***

    For example Current.ValueInMilliamps is an example where the numerical value of a current measurement is being returned in milliamps.

- The ToString function of each object will return a well formatted string describing the contained quantity. (i.e. "1.25 mA", or "-6.45 dBV")

13.1.1. Signal Levels

Measurement and specification of AC voltages are managed by the SignalLevel class. This class can represent signals in Volts RMS, dBV, and dBm and convert between each representation. This class also exposes several static functions to assist with common level calculations.

To create a signal level you must call one of the following static functions:

- **InVrms** – creates a level specified in Volts RMS (Vrms)

- **IndBV** – creates a level specified in decibels relative to 1 Volt RMS (dBV)

- **IndBm** – creates a level specified in decibels relative to the equivalent voltage generated by dissipating 1 milliWatt across a 600 Ω load. (dBm)

- **InRepresentation** – creates one of the above level specifications based on an enumerated argument.

Once a SignalLevel object is created it can be converted to any particular representation through one of the member functions.

- **ToVrms** – returns the signal level converted to Volts RMS

- **TodBm** – returns the signal level converted to dBm

- **TodB** – returns the signal level converted to dBV

- **ToRepresentation** – returns the signal level converted to the representation specified by an enumerated argument. This is very handy when it is desirable to programmatically change representations when displaying signal level results.

The numerical value corresponding to the signal level in the current representation is accessible through the following read-only properties

- **Value** – returns the numeric signal value expressed in the current representation.

- **Specification** – returns an enumerated value which indicates the specification system used to represent the signal level.

The SignalLevel class also implements a range of arithmetic operators which make manipulations and sweeps very easy to implement and understand.

All signal generator levels are specified as **open circuit levels** and are accurate if the generator is not connected to any AC load (termination). To obtain the desired signal level when the generator is terminated you will need to adjust the generator level to compensate for the source and termination impedance as discussed below.

To avoid unnecessary exceptions any zero absolute signal levels will be converted to extremely small non-zero values when converting to logarithmic representations such as dBV or dBm. For example 0 Vrms may convert to -1000 dBV.

Examples:

```csharp
SignalLevel Level = SignalLevel.InVrms(0.5);  // 0.5 Volts RMS
Level = Level.TodB();            // Converts to -6.02 dBV
Level = Level.TodBm();           // Converts to -3.80 dBm
Level = Level.ToVrms();          // Converts back to 0.5 Vrms

// Prints "500 mVrms" to the immediate window

Debug.Print(Level.ToString());

SignalLevel Min = SignalLevel.IndBV(-40);
SignalLevel Inc = SignalLevel.InVrms(0.1);
SignalLevel Max = SignalLevel.IndBV(12);

for (SignalLevel Level = Min; Level <= Max; Level += Inc)
{ // sweep from Min to Max incrementing  by Inc
}
for (SignalLevel Level = Min; Level <= Max; Level *= 2)
{     // sweep from Min to Max doubling each loop

}
```

When specifying signals levels to be generated on a transmission line (such as the telephone line) applications must be careful to compensate for source and load impedances which can affect the resulting signal level on the line. The two common cases are:

- **Un-Terminated** – when the signal generator is not loaded (or no equipment attached) the signal generator level will match the line level with no compensation required.

- **Terminated** – when the signal generator is loaded (terminated) the line level will be reduced by a factor which depends on the source and load impedances.

The situation illustrated in Figure 6 is representative of a telephone network when a phone goes off hook. In this scenario:

- From the perspective of the Central Office (CO), the telephone has terminated the far end of the telephone line and loads the signal generator within the CO. The signals measured on the telephone line which are generated by the CO will be proportionally less than the levels generated internally within the CO (or if the line had not been terminated)

- From the perspective of the telephone, the CO has terminated the far end of the telephone line and loads the signal generator within the phone. The signals measured on the telephone line which are generated by the phone will be proportionally less than the levels generated internally within the phone.

Unless explicitly specified otherwise all **measured** signal levels are measured directly from the transmission line labeled as "line level" (see Figure 6)
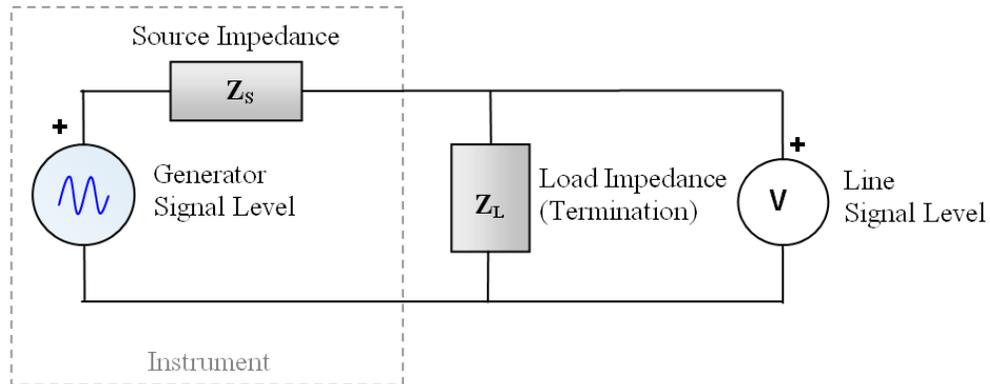
Figure 6 Signal Levels and Terminations

The resulting signal level (at a particular frequency f) can be calculated using the formula:

$$V_{line}(f) = \frac{Z_L(f)}{Z_L(f) + Z_s(f)} V_{generator}(f)$$

When the impedances involved are resistive this formula is very easy to evaluate (as it is the basic ohms law resistor divider equation). However many circuits involve reactive components (capacitors or inductors) which can make the magnitude and phase characteristics of the impedances vary with frequency. Each impedance value must then be represented using complex numbers which makes the above formula more difficult to evaluate. The SignalLevel class provides two static functions which calculate these generator levels and terminated signal levels for any source and load impedance. The two methods are:

- **CalculateGeneratorLevel** – calculates the signal generator level required to produce a desired (sinusoidal) signal level on the line at a particular frequency.

- **CalculateTerminatedLevel** – calculates the signal level which is present on the line given the level and frequency of the signal generator, the generator output impedance, and the termination impedance.

Each of these functions will accept Nothing/null for the termination impedance argument to specify "no termination".

Examples:

```
// This calculates the generator level required to
// produce a line level of -10 dBV @ 440 Hz
// when the generator has an output impedance
// compliant with TBR-21 (complex) and is
// teriminated with 600 ohms
Gen = SignalLevel.CalculateGeneratorLevel(
            SignalLevel.IndBV(-10),      // desired line level
            Frequency.InHz(440),         // signal frequency
            Impedance.TBR_21,            // output impedance
            Impedance.Resistive_600);    // termination impedance


// continued on next page...
```

```
// This calcualtes the line voltage which result
// if a generator produces -8 dBm @ 2 kHz with
// a 604 ohm output impedance and is terminated
// with a German ZR complex impedance
Line = SignalLevel.CalculateTerminatedLineLevel(
           SignalLevel.IndBm(-8), // generator level
           Frequency.InkHz(2),    // generator frequency
           Resistance.InOhms(604),// output impedance
           Impedance.German_ZR);  // termination impedance


// This demonstrates how an instrument's generator impedance
// setting can be used to calculate generator levels
Frequency Freq = Frequency.InHz(400);
Gen = SignalLevel.CalculateGeneratorLevel(
                SignalLevel.IndBV(-10),
                Freq,
                _7280.TelInt.ACImpedance,// output impedance
                Impedance.Resistive_600);

// Now we generate the specified level
_7280.ToneA.Generate(new Tone(Gen, Freq));
```

13.1.2. Frequency

Measurements and specifications of Frequency are managed by the Frequency class which represents frequency in Hertz (Hz). To assist in readability frequencies can be specified using two static functions:

- **InHz** – which creates a frequency descriptor specified in Hertz

- **InkHz** – which creates a frequency descriptor specified in kilohertz

The numeric value of a frequency can be accessed through the following read-only properties:

- **Value** – Returns the frequency in Hertz

- **ValueInkHz** – Returns the numeric frequency in kilohertz.

The Frequency class also defines a standard set of operators that make manipulations and sweeps very easy to implement and understand.

Examples:

```
Frequency Min = Frequency.InHz(100);
Frequency Max = Frequency.InkHz(20);
for (Frequency F = Min; F <= Max; F += Frequency.InHz(100))
{
    // Sweep from 100 Hz to 20 kHz in 100 Hz steps

}
```

13.1.3. DC Current

Measurements and specifications of DC currents are managed by the DCCurrent class. DC current descriptors can be created using the following static functions:

- **InAmps** – which creates a current descriptor specified in Amps (A)

- **InMilliamps** – which creates a current descriptor specified in milliamps (mA)

The value of a DC current can be accessed through the following read-only properties:

- **Value** – returns the current measured in amps (A)

- **ValueInMilliamps** – returns the current value expressed in milliamps (mA).

13.1.4. DC Voltage

Measurements and specifications of DC voltages are managed by the DCVoltage class. Voltage descriptors can be specified using the static function:

- **InVolts** – creates a voltage descriptor specified in Volts

The value of a DC voltage can be accessed through the following read-only property:

- **Value** – returns the voltage expressed in Volts

13.1.5. Resistance

Measurements and specifications of resistance are managed by the Resistance class which represents resistance in Ohms. Resistance descriptors can be created using the following static functions:

- **InOhms** – creates a resistance descriptor specified in ohms (Ω)

- **InKilohms** – creates a resistance descriptor specified in killohms (kΩ)

- **InMegohms** – creates a resistance descriptor specified in megohms (MΩ)

The numeric value of a resistance can be accessed through the following read-only properties:

- **Value** – returns the resistance expressed in ohms (Ω)

- **ValueInKilohms –** returns the resistance value expressed in kilohms (kΩ)

- **ValueInMegohms –** returns the resistance value expressed in megohms (MΩ)

13.1.6. Time Intervals and Durations

Relative measures of time are described by the TimeInterval class. Time interval descriptors can be created using the following static functions:

- **InSeconds** – creates a time interval descriptor specified in seconds (s)

- **InMilliseconds** – creates a time interval descriptor specified in milliseconds (ms)

The numeric value of a time interval can be accessed through the following read-only properties:

- **Value** – returns the time interval expressed in seconds

- **ValueInMilliseconds –** returns the time interval expressed in milliseconds

13.1.7. Decibels, Gain, and Unit-less Values

Some measurements and specifications such as gains, twists, and ratios are specified as unit-less quantities. These values are managed by the UnitlessQuantity class and can be created through one of the following static functions.

- **InAbsolute** – creates a unit-less value expressed as a number

- **IndB** – creates a relative unit-less value expressed in decibels (relative to 1)

Each unit-less quantity can be converted back and forth between representations using the functions:

- **ToAbsolute** – returns the unit-less value expressed as an absolute number

- **TodB** – returns the unit-less value expressed in dB (relative to 1)

The numerical value of the unit-less value can be returned by the read-only property

- **Value** – returns the unit-less value expressed in the current representation

Examples:

```
UnitlessQuantity Gain = UnitlessQuantity.IndB(-20); // -20 dB
Double Mult = Gain.ToAbsolute().Value;      // returns 0.1


// Adjust signal level using gain in decibels
Level = SignalLevel.InVrms(0.045) * UnitlessQuantity.IndB(20);
```

13.2. Impedances

Electrical impedance is a measure of opposition to alternating or direct current within a circuit; which depends on the type and configuration of components within any given circuit. Impedance characteristics are described within the aiDevices framework using the following classes and interfaces:

- **IImpedance** – this interface is implemented by any class which models a component or measure from which impedance characteristics can be calculated. This interface defines the GetImpedance function which will return the impedance (as a complex number) of the described network at a particular frequency.

- **Impedance** – this is a façade class that declares many static members and helper functions. This class is also used as a base class for the following classes.

- **ResistiveImpedance** – describes a purely resistive impedance characteristic which can be modeled with a simple resistor.

- **ComplexImpedance** – describes a complex impedance response which is defined by a series/parallel RC circuit (R_s +R_p||C_p) which is very commonly found in telephony impedance specifications.

The Impedance base class exposes several static variable declarations for common telephone line impedance values which are commonly installed in instruments.

- **Impedance.Resistive_600** – describes 600 Ω resistive impedance.

- **Impedance.Resistive_900** – describes a 900 Ω resistive impedance.

- **Impedance.TBR_21** – describes the complex ETSI TBR-21 network (270 Ω + [750Ω || 150 nF])

- **Impedance.German_ZR** – which represents the complex German-ZR impedance network (220 Ω + [820Ω || 115 nF])

> Terminated signal level calculations can be performed automatically based on these impedance descriptors through the SignalLevel static functions described in section 13.1.1.

Example:

```
' Sets the telephone line AC impedance to 600 Ω
_7280.TelInt.ACImpedance = Impedance.Resistive_600

' Sets the telephone interface AC impedance to German-ZR
_5620.TelInt.ACImpedance = Impedance.German_ZR

' Sets the telephone line AC impedance to 500 Ω
_5620.TelInt.ACImpedance = ResistiveImpedance.InOhms(500)
```

13.3. Filters and Signal Filtering

Most instruments supported by the aiDevices framework contain several filter banks which can be configured to make band-limited measurements, perform THD+N measurements, or generate band limits or shaped signals. Each filter bank typically supports a variety of filters, some of which have programmable corner frequencies (low pass, high pass etc) and some which have fixed response characteristics (DMTF, CMessage, etc).

Filters definitions are managed by means of descriptor classes that uniquely describe a particular filter configuration. Each filter descriptor class implements the IFilter interface and can be

- Created by a custom application and then passed to a device object which will configure the corresponding filter bank with the specified filter definition.

- Returned by a device object to an application which can then determine the filter configuration by casting the type of object returned.

While the potential number of filter definitions is limitless, most Advent Instruments products implement a very specific set of common filters. The Filter class exposes a façade of static functions which makes constructing these standard filters descriptors simple and easy to read.

- **ButterworthLowPass** – returns a 4th order Butterworth low pass filter descriptor with a programmable corner frequency

- **ButterworthHighPass** – returns a 4th order Butterworth high pass filter descriptor with a programmable corner frequency

- **ButterworthHighAndLowPass** – returns a descriptor with a combination of 4th order Butterworth low and high pass filters each with programmable corner filters.

- **Notch** – returns a single Butterworth notch filter descriptor with a programmable center frequency definition.

- **DualNotch** – returns a descriptor containing a combination of two Butterworth notch filters each with programmable center frequency

- **BandPass** – returns a standard 4th order Butterworth band pass filter descriptor with a programmable center frequency.

- **DTMFRow** – returns a descriptor that specified the filter which extracts only the low group (row) tones in a DTMF signal

- **DTMFColumn** – returns a descriptor that specified the filter which extracts only the high group (column) tones in a DTMF signal

- **AWeighting** – returns a standard A-weighting filter descriptor

- **CMessage** – returns a standard C-Message weighting filter descriptor

- **O41Psosphometric** – returns an ITU O.41 filter descriptor

Examples:

```vb
' Configures a butterworth lowpass (fc=1Khz) before the meter measurements
_7280.Meter.MeasurementFilter = Filter.ButterworthLowPass(Frequency.InkHz(1))

' Configures notch filters to remove 350 Hz and 440 Hz
 7280.Meter.NotchFilterBank = Filter.DualNotch(Frequency.InHz(350),
                                              Frequency.InHz(440))

' Apply the O.41 filter before measurements
 5620.Meter.MeasurementFilter = Filter.O41Psophometric
```

Not all instruments support all filter types or configurations! Be sure to check the instrument capabilities before assigning filter definitions. If an unsupported filter type is detected the associated device objects will generally throw a NotSupportedException

Filter descriptions can be very heterogeneous in nature and may not share any common base elements due to differences in implementation or specification. **All filter descriptors implement the IFilter interface and may not necessarily inherit from the Filter class.**

Each possible filter descriptor is sub-classed into categories based on response and function as illustrated by the class diagram shown in Figure 7.
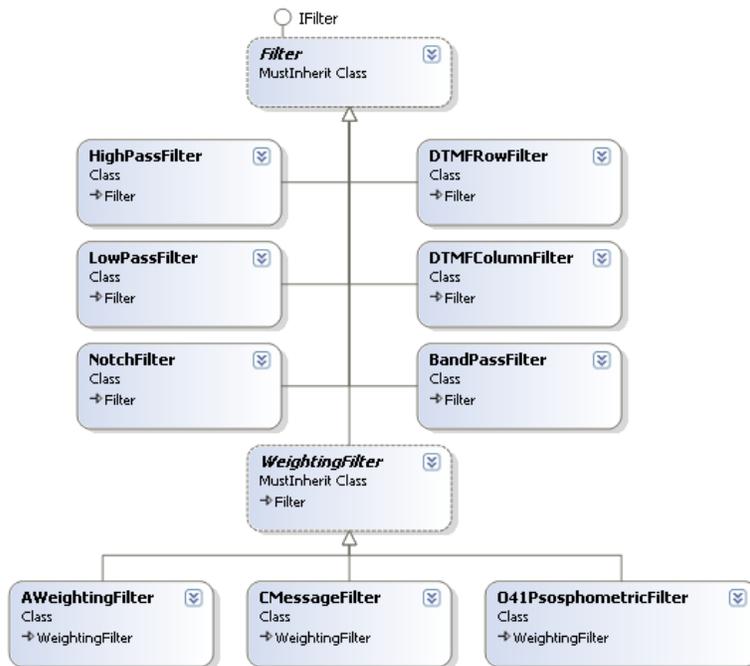
Figure 7 Filter class diagram

Applications which require more detailed information on the filter configuration can leverage the descriptor's inheritance structure and cast to determine the filter type as illustrated in the following example.

Examples:

```vbnet
Private Sub DetermineFilterType(ByVal Filter As IFilter)
    ' Check if it's a low pass filter
      Dim LPF = TryCast(Filter, LowPassFilter)
      If LPF IsNot Nothing Then
            Debug.Print("Low pass with corner=" & _
                  LPF.CornerFrequency)
      End If
      ' check if it's a notch filter
      Dim Notch = TryCast(Filter, NotchFilter)
      If Notch IsNot Nothing Then
            Debug.Print("Notch filter with center=" &
            Notch.CenterFrequency)
      End If
End Sub
```

13.4. Telephone Line State

Much of the signaling in analog telephony is achieved through changes in telephone line state. The general "state" of the telephone line can be determined by examining the line voltage and loop current passing through a telephone interface In general a telephone line can be viewed as being in one of four general states depending on the perspective of the viewer:

Disconnected • Very low voltage • Extremely low current This state is generally only experienced by terminal equipment when they have been disconnected from the central office	**In Use** • Low voltage • Low current This state is generally only experienced by Terminal Equipment (TE) when another parallel TE is already off hook.
Off hook • Low line voltage • High loop current This state occurs when the receiver is lifted on a phone which then draws loop current from the central office.	**On Hook** • High Line Voltage • Very low loop current This state occurs when the central office is connected to the telephone line but no phones are off hook.

Figure 8 Telephone Line States

Within the aiDevices framework these four states are represented using the Telephone_Line_State enumeration which has a value corresponding to each state shown in Figure 8. This type may be used by device classes, notifications, and signal definitions to describe these general line states.

13.5. Telephone Line Polarity

Another signaling method used within analog telephony is the "line reversal" which can be characterized as a polarity reversal of the line voltage as measured between the two conductors (tip and ring). The polarity of a telephone line is specified within aiDevices using an enumeration named Telephone_Line_Polarity which has two members:

- **Normal** – which indicates the voltage is in the "normal" polarity as defined by the relevant instrument

- **Reversed** – which indicates the voltage is reversed in relation to the instrument's normal polarity definition.

Each instrument supported by aiDevices defines a particular polarity as "Normal". Typically this polarity will result in negative line voltage measurements from the telephone interface since the "normal" DC voltage feed is traditionally specified as -48 Volts. Please note:
- Changes in line polarity are always specified relative to the instrument specific "normal" polarity
- Many telephone cables are wired without regard to polarity of the tip and ring conductors and may result unexpected telephone line polarities.

14. Signal Descriptor Classes

Within the aiDevices framework all phenomena which can be programmatically generated or detected by an instrument are collectively labeled as "signals". In fact a large number of the classes within the assembly are devoted to specifying these signals so the resulting descriptors can be used as a means of communication between applications and the device objects.

The above signal definition is quite broad and can encompass a large variety of definitions which differ widely with regard to their characteristics, meaning, and even transmission medium (for example DTMF vs. OSI). The aiDevices framework uses a variety of different language features and design techniques to make dealing with these signals simple, abstract, and extensible. The following sections detail the design and usage of each signal supported within the aiDevices framework.

14.1. Interfaces and Signal Categorization

From a cursory examination of the "signal" definition above it is clear that

- Many different types of phenomena can be labeled as "signals" but may vary greatly in meaning, representation, and medium and will thus have very little in common with each other.

- When considering a particular type of signal many of the specifications required to generate the signal are also required to adequately report the detection of the same signal. For example levels and frequencies are required to specify the generation and detection of a DTMF signal.

- Some signals are considered "valid" with or without reference to timing. For example DTMF is "valid" so long as the correct frequencies and levels, whereas CAS/DTAS is only valid if the frequencies and levels are applied for a very particular duration.

Since different types of signals do not inherently share many characteristics, all classes that represent signals do not inherit from a base class but instead implement the **ISignal interface**. This interface:

- Labels the descriptor classes that represent signals

- Defines the SignalType property which identifies the type of signal.

- Allows applications to collect and pass signal descriptors abstractly using ISignal without reference to their concrete types.

It is exceedingly difficult to design a general interface or base class to group all signals by particular characteristics (like frequency, voltage, etc) however aiDevices uses two other interfaces to group signals by duration and semantics illustrated below in Figure 9.
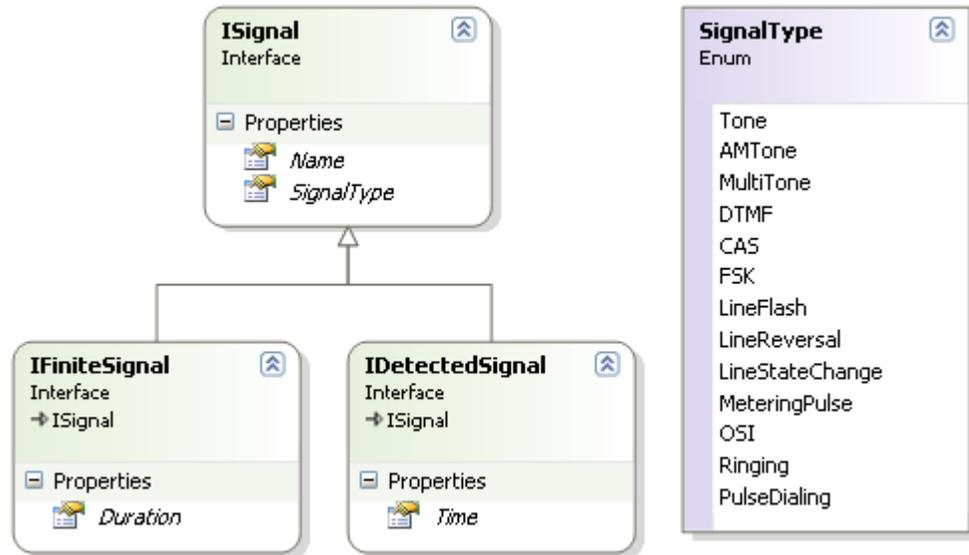
Figure 9 Signal Interface Inheritance Diagram

> All descriptor classes that represent a signal which occurs for a well defined and finite duration implement the **IFiniteSignal** interface.

This IFiniteSignal interface contains only one member which reports the duration of the signal. While this may seem trivial when examining a singular signal type, this information becomes essential when sequencing and sorting signal types.

> All descriptor classes that represent a signal which has been detected by an instrument implement the **IDetectedSignal** interface.

It is worthy of note that the ISignal interface does not contain a specification for a detection or transmission time. This is because the transmission time of signals are specified separately when generated so the same signal object can be used for multiple transmissions without manipulation. The IDetectedSignal interface exposes only a single member which reports the detection time of the signal.

14.2. Signal Descriptors and Inheritance Patterns

As noted in the previous section, it is very difficult to design a useful base class for all possible types of signals based on general characteristics. However, when considering a particular type of signal it essential to use base classes to ensure optimal code reuse, abstraction, and object compatibility.

In general, each *particular* type of signal can be specified using a set of classes which are all related through inheritance. The base class is usually named in a manner which indicates the type of signal it represents and is the most general and timing invariant. Each successive subclass then becomes successively more specific and named to convey the functional differences from its base class. As an example of this general inheritance pattern, consider the Ringing signal illustrated below.
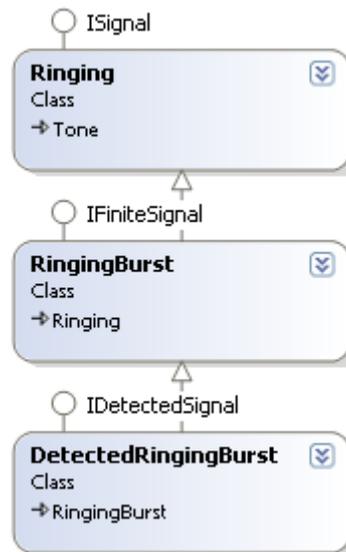
Figure 10 Example of Signal Inheritance Pattern

The following discussion will use the example shown above in Figure 10 to relate the inheritance design pattern commonly used with many different signals.

- The Ringing base class represents a ringing signal which can be applied on a telephone line to alert a customer to an incoming phone call. It contains properties for the signal level, frequency, wave shape, and DC offset to specify the basic characteristics of the voltage signal.

    o The Ringing base class implements the ISignal interface which labels it as representing a type of signal

    o Since a ringing signal is considered valid regardless of duration (which could even be infinite) this base class does not contain duration or timing information. In fact this class is so abstract it could conceivably be used to abstractly configure a generator or even report signal measurements.

- The RingingBurst class represents a ringing signal with a finite duration and hence implements the IFiniteSignal interface. Since this class inherits from the Ringing class it also contains all the same properties for defining a ringing signal. This class is slightly more concrete (in that it now only applies to finite duration ringing signals) however this class still does not contain any information regarding the particular instant in time to which this ringing burst occurs. By omitting this timing the same class may be used to specify multiple instances of the same signal.

- The DetectedRingingBurst class represents a singular instance of a ringing signal which has been detected by an instrument and thus implements the IDetectedSignal interface.

    o This class reports the particular instant in time when the start of this signal was detected.

    o This class may report extra measurement information over and above the Ringing and RingingBurst signal which pertains to this particular instance of the signal but not to ringing signals in general. (such as noise or distortion measurements)

14.3. Wave Shape

While not technically a signal, the Waveshape class describes the "shape" of periodic signals within the aiDevices framework. Each Waveshape object exposes the following members:

- **Name** – this property returns a descriptive name for this wave shape

- **CrestFactor** – this returns the ideal **theoretical** crest factor for the ideal wave shape. The crest factor is defined as the ratio of the peak signal value and the signal's Root-Mean Square (RMS) value. Deviations from this ideal theoretical crest factor indicate a distortion of some form (which cannot be determined from this measure alone)

Waveshape objects cannot be created directly by applications but rather the Waveshape class statically declares three natively supported shapes which illustrated in Figure 11.

Static Member	Theoretical Crest Factor	Illustration
Waveshape.Sinusoial	$\sqrt{2} \approx 1.414$	
Waveshape.Triangular	$\sqrt{3} \approx 1.732$	
Waveshape.Square	1	
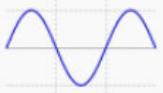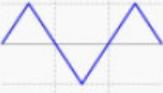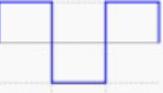
Figure 11 Native Wave Shapes

These Waveshape objects can then be used to define other higher level signals or configure signal generators.

Example:

```
// Make ringing sinusoidal
_7280.Ring.Shape = Waveshape.Sinusoiodal;

// change the tone to triangular shape
_7280.ToneA.Shape = Waveshape.Triangular;
```

14.3.1. CustomWaveShape

In certain cases applications may wish to define their own wave shape for use with signal generators. The CustomWaveShape class (derived from Waveshape) addresses this need by enabling users to create arbitrary shapes based on an array of floating point samples. This object will calculate the crest factor for these custom signals and ensure the sample format is appropriately scaled for use with signal generators. While the following example used a very small array it demonstrates how to use the Create method to construct such a custom wave shape.

Example:

```
// Create a custom wave shape definition
CustomWaveShape MyShape = CustomWaveShape.Create("Test Shape 1",
                            new double[]{1,2,-2,-1});
//Note: A "real" signal should have more sample points!
```

Not all device objects or instruments may support custom wave shapes
- Some simply may not support custom wave shapes
- Some may have a limitation on the number of points defined in the wave shape
Please check the device specific documentation for support information for your particular instrument.

14.4. Cadence

The general terms "cadence" and "signal pattern" are used interchangeably with the aiDevices framework to describe the timing structure of signals which may be generated by an instrument. These patterns are represented by the following descriptor classes:

- **Cadence** – describes an arbitrary on/off timing pattern which does not repeat.

- **RepeatingCadence** – describes a cadence which may be repeated a finite or infinite number of times.

Each Cadence is described by a sequence of durations that correspond to the lengths of time when the signal is active or inactive. The sequence alternates between:

- **On Time** – specifies the length of time when the signal is applied

- **Off Time** – specifies the length of time when the signal is not applied (between signals)

Each pair of "on" and "off" times forms an "interval" as illustrated in Figure 12.
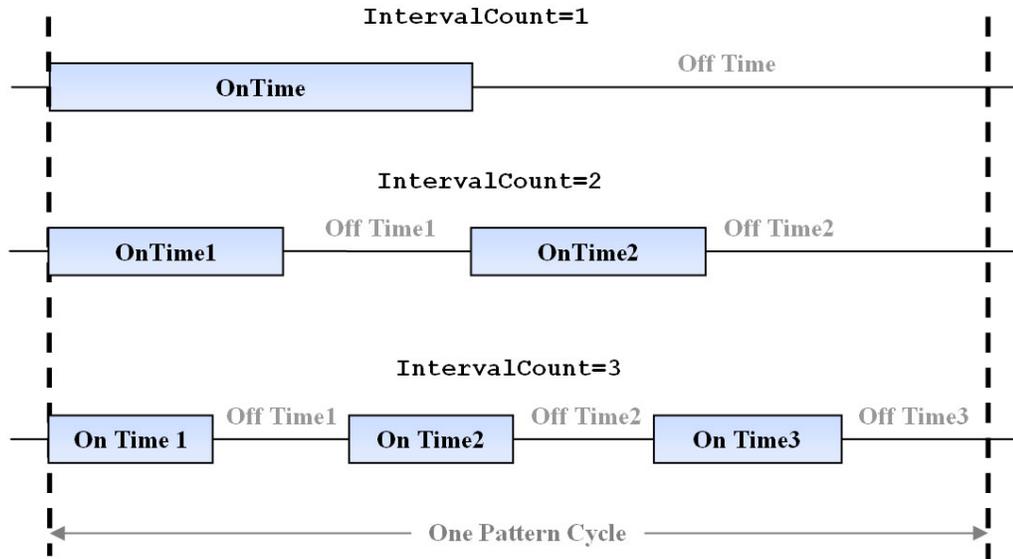
Figure 12 Simple Cadence Examples

As mentioned above, the Cadence class represents an arbitrary signaling pattern which is specified using a sequence of on and off times. The Cadence class exposes the following members:

- **IntervalCount** – returns the number of (on/off) intervals defined within the pattern

- **OnTime** – returns the "On Time" for an interval specified by index

- **OffTime** – returns the "Off Time" for an interval specified by index

- **Name** – returns a description of the pattern

Some signals are sequential in nature and specify patterns with zero off time between each signaling element. To specify such patterns the Cadence class defines the following static functions:

- **AdjacentTiming** – specifies a cadence using on-times and sets each off-time to zero.

The RepeatingCadence class is derived from Cadence and represents multiple repetitions of a pattern. This class defines the following members:

- **RepeatCount** – returns the number of times the pattern should be repeated

- **RepeatForever** – returns true if the pattern should be repeated indefinitely

- **IsActiveAfter** – returns true if the signal should be left "on" one the pattern completes.

All classes that represent a signal pattern or cadence implement the ISignalPattern interface.

Examples:

```vbnet
' 1 second on
P = New Cadence(TimeInterval.InSeconds(1))

' 2 seconds on, 4 seconds off
P = New Cadence(TimeInterval.InSeconds(2), _
                TimeInterval.InSeconds(4))


' 2 x 50 ms bursts separated by 100 ms
P = New Cadence("Custom Pattern",
                TimeInterval.InMilliseconds(50), _
                TimeInterval.InMilliseconds(100), _
                TimeInterval.InMilliseconds(50))

' 10 x (100ms on, 100 ms off) then signal is left on
P = New RepeatingCadence("Stutter Dial", 10, True, _
                    TimeInterval.InMilliseconds(100), _
                    TimeInterval.InMilliseconds(100))

' 2 seconds on, 4 seconds off repeated indefinitely
P = New RepeatingCadence("Ring forever", _
                    TimeInterval.InSeconds(2), _
                    TimeInterval.InSeconds(4))

' 50ms on, 100ms on, 150 ms on
P = Cadence.AdjacentTiming(TimeInterval.InMilliseconds(50), _
                           TimeInterval.InMilliseconds(100), _
                           TimeInterval.InMilliseconds(150))
```

14.5. Tones

Within the aiDevices framework a "Tone" is defined as:

- An AC signal of infinite or finite duration which is

  - o Periodic with a consistent fundamental frequency and shape

  - o Consistent in signal level from start to completion

The Tone class represents such a signal and can represent sinusoidal, square, triangular, and custom shaped signals of constant level and frequency. Often these tone structures are used as fundamental elements in more complex signal definitions (DTMF, CAS, etc) but may also be used directly to update and manage support objects like tone generators documented in section 16.1.1.

The tone class exposes the following members:

- **Level** – returns the signal level of the tone (see section 13.1.1)

- **Frequency** – returns the frequency of the AC signal (see section 13.1.2)

- **Shape** – returns a descriptor of the "shape" of the waveform (see section 14.3)

Example:

```vb
' 0 dBV sine wave at 440 Hz
Dim Tone1 = New Tone(SignalLevel.IndBV(0), _
                     Frequency.InHz(440))

' +3 dBm sine wave at 480 Hz
Dim Tone2 = New Tone(SignalLevel.IndBm(3), _
                     Frequency.InHz(480), _
                     Waveshape.Sinusoiodal)



' 0.1 Vrms square wave at 1 kHz
Dim Tone3 = New Tone(SignalLevel.InVrms(0.1), _
                     Frequency.InkHz(1), _
                     Waveshape.Square)
```

14.5.1. Amplitude Modulated Tone

In addition to "pure" tones, the aiDevices framework supports the specification of amplitude modulated tones using the AMTone class. This class inherits from the Tone class (which defines the carrier signal) and exposes the following additional members:

- **ModulationFrequency** – returns the frequency of the modulating signal.

- **ModulationDepth** – returns the depth of the modulating signal in percent from 0 to 100.

- **ModulationShape** – returns a descriptor of the "shape" of the modulating signal (see section 14.3).

Example:

```vb
' 10 kHz sine wave with 200 Hz square wave modulation
 Dim AM1 = New AMTone(CarrierLevel:=SignalLevel.IndBV(0), _
                      CarrierFrequency:=Frequency.InkHz(5), _
                      CarrierShape:=Waveshape.Sinusoiodal, _
                      ModulationDepth:=50, _
                      ModulationFrequency:=Frequency.InHz(200), _
                      ModulationShape:=Waveshape.Square)

' 400 hz sine wave with 75% 20 Hz sinusoidal modulation
 Dim AM2 As New AMTone(CarrierLevel:=SignalLevel.InVrms(1), _
                       CarrierFrequency:=Frequency.InHz(400), _
                       ModulationDepth:=75, _
                       ModulationFrequency:=Frequency.InHz(20))
```

14.6. Multi-Tone Signals

One classification of signal which occurs frequently in telephony, and is supported by most Advent Instruments products, is referred to as a 'Multi-Tone Signal' which is defined as:

- An alternating signal of arbitrary or finite duration consisting of

  - The summation of one or more periodic signals (tones) each having constant signal level, fundamental frequency, and wave shape; where

  - Each tone is simultaneously applied.

Within the realm of telephony signals this definition can apply to

- DTMF Signals (see section 14.9)

- CAS/DTAS Signals (see section 14.10)

- Metering Pulses (see section 14.11)

- MF Signaling

- Call Progress Tones, Special Information Tones, etc.

The MultiToneSignal class exposing the following members:

- **Name** – this returns a descriptive name for this signal.

- **Tones** – this property exposes a collection of the Tone objects used to specify the signal. These tones are sorted in order of ascending frequency.

The MultiToneSignal class is used as a base class for other specific signal classes but can also be used to directly create custom signal definitions. MultiToneSignal object can be constructed in several different manners by specifying

- An array of Tone objects

- A total level and a parameter array of one or more Frequency objects specifying the frequencies of each tone in the object (the wave shape is assumed to be sinusoidal)

Example:

```
' Create a signal with a total level of -3 dBV containing 4 tones
Dim ReceiverOffHook = New  MultiToneSignal.(SignalLevel.IndBV(-3), _
                                            Frequency.InHz(1400), _
                                            Frequency.InHz(2060), _
                                            Frequency.InHz(2450), _
                                            Frequency.InHz(2600))

' define three separate tones
Dim Tone1 = New Tone(SignalLevel.IndBV(0), Frequency.InkHz(1.2))
Dim Tone2 = New Tone(SignalLevel.IndBV(-3), Frequency.InkHz(2.2))
Dim Tone3 = New Tone(SignalLevel.IndBV(-6), Frequency.InHz(100))

' Create a multi-tone signal using the new operator and Tone objects
Dim T = New MultiToneSignal(Tone1, Tone2, Tone3)
```

14.7. Multi-Tone Sequence

The MultiToneSequence allows developers to define more complicated signals defined using a sequence of arbitrary multi-tone signals with specific timing. Such signals might include:

- **DTMF Dialing** – dialing sequences are defined as a sequence of DTMF digits. Note: each successive digit may be specified completely independently of the other digits.

- **Special Information Tones** – many telephone networks generate a very special sequence of tones (with particular frequencies and cadence) when a call could not be completed for a variety of reasons.

- **Payphone Recognition Tones** – which are a typically sequence of 1 or 2 tones which are generated when a pay telephone recognizes payment.

The MultiToneSequence class exposes the following members:

- **Tones** – returns a list of the multi-tone signals specified in the sequence in order of transmission (see section 14.6).

- **Cadence** – returns the descriptor which defines the timing of the signals (see section 14.4)

- **Name** – returns a descriptive name for the signal pattern

The sequences can be created using constructors or through static Create functions.

Examples:

```vb
Dim Level = SignalLevel.IndBV(-10)
Dim OnTime1 = TimeInterval.InMilliseconds(330)
        Dim OnTime2 = TimeInterval.InMilliseconds(274)
        Dim OnTime3 = TimeInterval.InMilliseconds(200)

 ' 3 sequential ascending tones with equal timing
 Dim S = MultiToneSequence.Create("Special Information Tone)", _
          Cadence.AdjacentTiming(OnTime1, OnTime1, OnTime1), _
          New Tone(Level, Frequency.InHz(985)), _
          New Tone(Level, Frequency.InHz(1428)), _
          New Tone(Level, Frequency.InHz(1776)))

Dim Tone1 = New MultiToneSignal(Level, _
                                Frequency.InHz(1100), _
                                Frequency.InHz(1750))

Dim Tone2 = New MultiToneSignal(Level, _
                                Frequency.InHz(750), _
                                Frequency.InHz(1450))

' 2 separate dual tone signals 200 ms each separated by 274 ms
Dim S = New MultiToneSequence("Payphone Recognition", _
                New Cadence(OnTime3, OnTime2, OnTime3), _
                Tone1, Tone2)

' Creates a multi-tone sequuence containing DTMF dialing
 Dim Dialing = DTMF.CreateDTMFDialing("5551234", _
                          SignalLevel.IndBm(-5), _
                          Durations:=OnTime1, _
                          InterDigit:=OnTime2)
```

14.8. Dual-Tone Signals

A dual-tone signal is a specialization of a multi-tone signal that applies to signals with only two simultaneously applied tones. This specialization directly applies to:

- DTMF signals (see section 14.9)

- CAS/DTAS signals (see section 14.10)

- Many call progress tones (which vary depending on region)

The DualToneSignal class inherits from MultiToneSignal and adds the following members:

- **HighTone** – this returns the tone definition within the signal with the higher frequency

- **LowTone** – this returns the tone definition within the signal with the lower frequency

- **Twist** – this returns the difference in level between the two tones expressed as a ratio (high frequency level / low frequency level).

Examples:

```vb
Dim Tone1 = New Tone(SignalLevel.IndBV(0), _
                     Frequency.InHz(440))
Dim Tone2 = New Tone(SignalLevel.IndBV(-3), _
                      Frequency.InHz(480))

' Each tone can ne completly specified individually
Dim RingBack = New DualToneSignal(Tone1, Tone2)

' Signal with total level of -12 dBV and two
' sinusoidal tones with equal levels
Dim DialTone = New DualToneSignal(Frequency.InkHz(350), _
                                  Frequency.InkHz(440), _
                                  SignalLevel.IndBV(-12))

' Signal with total level of -6 dBV and two
' sinusoidal tones with 3 dB twist
 Dim Busy = New DualToneSignal(Frequency.InkHz(480), _
                               Frequency.InkHz(620), _
                               SignalLevel.IndBV(-6), _
                               UnitlessQuantity.IndB(3))
```

14.9. Dual Tone Multiple Frequency Signals

Dual Tone Multiple Frequency (DTMF) signals are a specialization of a dual-tone signal which is commonly used in telephony networks as a form of in-band signaling. DTMF is used as the signaling method for touch tone dialing, an acknowledgement signal in certain Caller ID sequences, and even as a method of Caller ID delivery in certain regions.

Each DTMF signal consists of a combination of two tones selected from a 4x4 matrix which relates them to a particular key on a touch tone phone. Each key then corresponds to a row and column frequency as shown in Figure 13.

		Column Frequency			
		1209 Hz	1336 Hz	1477 Hz	1633 Hz
Row Frequency	697 Hz	1	2	3	A
	770 Hz	4	5	6	B
	852 Hz	7	8	9	C
	941 Hz	*	0	#	D

Figure 13DTMF Signal Matrix

The four keys in the rightmost column are historically used for network signals not related to dialing.

For the purposes of this document:
- The dual tone frequency specification above will be called a "DTMF signal"
- A particular finite duration instance of a DTMF may further be referred to as a "DTMF digit".

DTMF signals are specified by many different standard bodies which can vary slightly in exact specification; however the following general characteristics apply:

- Each DTMF signal must contain two sinusoidal tones each of which differ from the nominal frequencies in Figure 13 by a small percentage

- The difference in signal level between the two tones must not exceed a particular ratio; often specified as twist and expressed in decibels (dB).

- Each DTMF digit must have a minimum duration but beyond this requirement may be ambiguous in duration (callers may press and hold a touch tone key as long as they like during which DTMF will be generated).

To address all the different factors required to represent the DTMF signal template, represent DTMF signals, and specific detected signals aiDevices defines three separate classes as illustrated below:
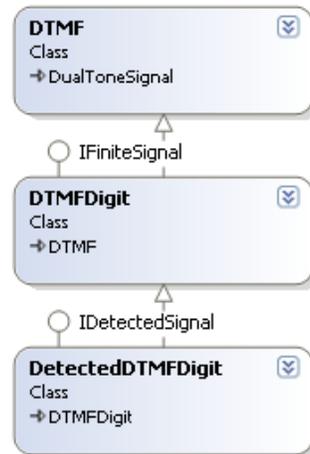
Figure 14 DTMF Class Diagram

In general:

- The DTMF class describes any dual tone signal which conforms to the DTMF frequency specification

- The DTMFDigit class describes a DTMF signal of finite duration without regard to absolute timing or usage.

- The DetectedDTMFDigit class represents a specific instance of a DTMF digit which has been detected and reported by an instrument.

14.9.1. DTMF

The DTMF class abstracts the DTMF frequency specification shown in Figure 13 but does not specify timing, duration, or instance specific values which are handled by derived classes. The DTMF class inherits from DualToneSignal and defines the following additional members:

- **Key** – returns the label in the DTMF matrix which corresponds to the signal frequencies with the signal (see Figure 13)

- **RowTone** – returns the Tone object which contains the row frequency

- **ColumnTone** –returns the Tone object which contains the column frequency

The DTMF class also exposes several static members which expose some shortcuts for dealing with the row/column frequency and key mapping:

- **GetKey** – this static function accepts a set of frequencies and returns the closest matching DTMF key.

- **GetRowFrequency** – this accepts a DTMF key character and returns the corresponding nominal row frequency

- **GetColumnFrequency** – this accepts a DTMF key character and returns the corresponding nominal column frequency

- **Keys** – this exposes an array of characters containing all valid DTMF key values.

- **IsDTMFKey** – function which returns true if the argument passed is a valid DTMF key character.

Examples:

```vb
' Creates a DTMF '3' signal with equal row and column levels
Dim D As New DTMF("3"c, SignalLevel.IndBV(-8))

' Creates a DTMF '3' signal with 3 dB twist
Dim D As New DTMF("D"c, SignalLevel.IndBV(-8), _
                        UnitlessQuantity.IndB(3))


' static functions (returns values commented )
DTMF.GetRowFrequency("#"c)        ' returns 941 Hz
DTMF.GetColumnFrequency("3"c)     ' returns 1477 Hz
DTMF.IsDTMFKey("3"c)              ' returns True
DTMF.IsDTMFKey("P"c)              ' returns False
DTMF.GetKey(Frequency.InHz(697), _
            Frequency.InHz(1209))    ' returns '1'
```

14.9.2. DTMF Digit

The DTMFDigit class derives from the DTMF class and abstracts DTMF signals with a finite duration The DTMFDigit class exposes the following members in addition to those exposed by the DTMF class:

- **Duration** – specifies the duration of the DTMF digit

Complete DTMF dialing sequences can be created using the MultiToneSequence class defined in section 14.7 or through the static DTMF.CreateDTMFDialing functions.

Example:

```csharp
// Creates a DTMF 'D' for 45 ms
DTMFDigit Ack = new DTMFDigit('D',
                SignalLevel.IndBV(-10),
                TimeInterval.InMilliseconds(45));

TimeInterval X = Ack.Duration;  // 45 ms
```

14.9.3. Detected DTMF Digit

The DetectedDTMFDigit class is the most specific of the DTMF signal classes and represents a singular occurrence of a DTMF digit that was detected by an instrument at a particular time. DetectedDTMFDigit objects are created exclusively by device objects when DTMF is detected and reported through notifications described in section 10.1.3. The DetectedDTMFDigit class inherits from the DTMFDigit class and exposes the following additional members:

- **Time** – reports the instant in time corresponding to the start of the detected DTMF digit

In future releases this class may report more specific information regarding DTMF signals including any measurements or analysis which may have been performed on the detected signal.

14.10. CAS/DTAS Signals

Many telephony standards specify a dual-tone signal which is used to alert equipment of subsequent incoming Caller ID messages. Most standard bodies refer to this signal as either:

- Customer Premises Equipment Alerting Signal (CAS); or
- Dual Tone Alerting Signal (DTAS)

The aiDevices framework and this document refer to this alerting signal as 'CAS' but assumes that names CAS and DTAS may be used interchangeably and refer to the same signal type.

While the details of the specification vary slightly between each standard, in general the signal is defined as:

- An AC signal which consists of two simultaneously applied sinusoids; where
- The sinusoids have nominal frequencies of 2130 Hz and 2750 Hz; and
- The signal is applied for a specific period of time; Usually 75ms or 100ms.

To address all the different factors required to represent the CAS signals, aiDevices defines two classes as illustrated in Figure 15. Note: since CAS includes a fixed duration requirement there are no CAS signal objects that do not implement IFiniteSignal.
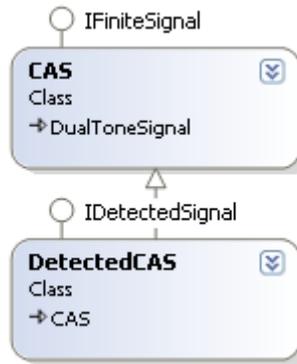
Figure 15 CAS class diagram

14.10.1. CAS

As the name suggests the CAS class represents a CAS signal defined above. Since CAS is composed of two sinusoids it inherits from the DualToneSignal class (see section 14.8) and exposes the following members:

- **Duration** – returns the duration of the CAS signal.

In addition, the CAS class also declares the following static members:

- **NominalHighCASFrequency** – which returns 2750 Hz

- **NominalLowCASFrequency** –which returns 2130 Hz

Example:

```
// Creates a CAS signal using twist
CAS Alert = new CAS(SignalLevel.IndBV(-20),          // -20 dBV total
                UnitlessQuantity.IndB(-1),           // -1dB Twist
                TimeInterval.InMilliseconds(75));    // 75 ms


// Creates a CAS with non-nominal frequencies and durations
CAS Alert2 = new CAS(CAS.NominalLowCASFrequency * 1.01, // 2130 Hz + 1%
             CAS.NominalHighCASFrequency * 0.99,         // 2750 Hz - 1%
             SignalLevel.IndBV(-20),
             UnitlessQuantity.IndB(-1),
             TimeInterval.InMilliseconds(75) * 1.02 );  // 75 ms + 2 %
```

14.10.2. Detected CAS

The DetectedCAS class represents a singular occurrence of a CAS signal that was detected by an instrument at a particular time. DetectedCAS objects are exclusively created by device classes when CAS is detected and reported through notifications described in section 10.1.3. The DetectedCAS class inherits from the CAS class and exposes the following additional members:

- **Time** – reports the instant in time corresponding to the start of the detected CAS signal.

In future releases this class may report more specific information regarding CAS signals including any measurements or analysis which may have been performed on the detected signal.

14.11. Metering Pulse Signals

Metering pulses are signals typically sent by telephone exchanges to telephones to inform the customer of the relative expense of a phone call in progress. Usually each pulse represents a particular incremental cost and more expensive calls will result in more pulses sent per minute. Metering pulse definitions vary depending on region and equipment manufacturer however generally a metering pulse can be defined as

- An AC signal which consists of a single sinusoid; where

- The sinusoid has a consistent frequency (usually 12 kHz or 16 kHz); and

- Is applied for a finite period of time (which can vary but is generally longer than 45 ms)

To address all the different factors required to represent metering pulse signals, aiDevices defines three classes as illustrated in Figure 16.
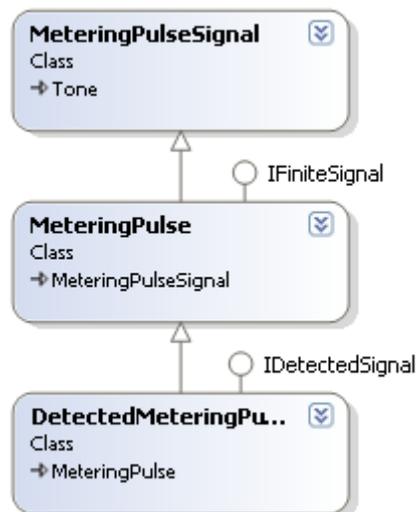
Figure 16 Metering Pulse Class Diagram

14.11.1. MeteringPulseSignal Class

The MeteringPulseSignal class describes the metering pulse specification described in 14.11. This class does not however specify timing, duration, or instance specific values which are handled by derived classes. The MeteringPulseSignal class inherits from Tone and exposes the following addition members:

- **Name** – returns a description of the signal

14.11.2. MeteringPulse Class

The MeteringPulse class derives from the MeteringPulseSignal class and represents a particular metering pulse with a particular finite duration. At this level of abstraction this class can represent any metering pulse regardless of usage (transmitted, detected, etc). The MeteringPulse class exposes the following members in addition to those exposed by the DTMF class:

- **Duration** – specifies the duration of the DTMF digit

Examples:

```
' defines a metering pulse as 0.2 Vrms at 12 kHz for 200 ms
Dim MP As New MeteringPulse(SignalLevel.InVrms(0.2), _
                            Frequency.InkHz(12), _
                            TimeInterval.InMilliseconds(200))
```

14.11.3. DetectedMeteringPulse

The DetectedMeteringPulse class represents a singular occurrence of a metering pulse that was detected by an instrument at a particular time. DetectedMeteringPulse objects are exclusively created by device classes when a metering pulse is detected and reported through notifications described in section 10.1.3. The DetectedMeteringPulse class inherits from the MeteringPulse class and exposes the following additional members:

- **Time** – reports the instant in time corresponding to the start of the signal.

In future releases this class may report more specific information regarding metering pulse signals including any measurements or analysis which may have been performed on the detected signal.

14.12. Ringing Signals

Ringing signals are typically very high voltage AC signals which are generated by a Central Office (CO) and are detected by TEs (phones) to cause them to make a sound (or other indication) which alerts the customer to an incoming phone call. While ringing signals vary from region to region, it can be defined as:

- A high voltage periodic AC signal; where

- The signal has a stable fundamental frequency (typically between 10 Hz and 100 Hz); and

- Is applied for a finite period of time; usually with a particular cadence.

To address all the different factors required to represent ringing signals, aiDevices defines three classes as illustrated in Figure 17.
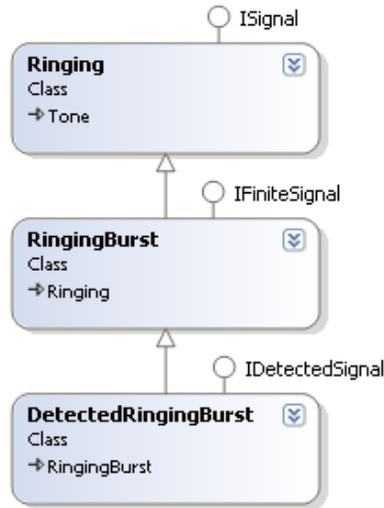

Figure 17 Ringing Class Diagram

14.12.1. Ringing Class

The Ringing class describes a ringing signal without specifying timing, duration, or instance specific values which are handled by derived classes. The Ringing class inherits from the Tone class and exposes the following addition members:

- **Name** – returns a description of the signal

- **DC** – specifies the DC offset of the ringing signal

14.12.2. RingingBurst Class

The RingingBurst class derives from the Ringing class and represents a particular ringing signal with a particular finite duration. At this level of abstraction this class can represent any ringing signal regardless of usage (transmitted, detected, etc). The RingingBurst class exposes the following members in addition to those exposed by the DTMF class:

- **Duration** – specifies the duration of the ringing signal

Examples:

```
Dim R = New RingingBurst(SignalLevel.InVrms(80), _
                         Frequency.InHz(22), _
                         DCVoltage.InVolts(48), _
                         TimeInterval.InSeconds(1))
```

14.12.3. DetectedRingingBurst Class

The DetectedRingingBurst class represents a singular occurrence of a ringing burst that was detected by an instrument at a particular time. DetectedRingingBurst objects are exclusively created by device classes when a metering pulse is detected and reported through notifications described in section 10.1.3. The DetectedRingingBurst class inherits from the RingingBurst class and exposes the following additional members:

- **Time** – reports the instant in time corresponding to the start of the signal.

> In future releases this class may report more specific information regarding metering pulse signals including any measurements or analysis which may have been performed on the detected signal.

14.13. Telephone Line State Changes

Much of the basic signaling in analog telephony is achieved through changes in telephone line state defined by line voltage and loop current (see section 13.4). These changes in telephone line states are described by several different classes illustrated below.
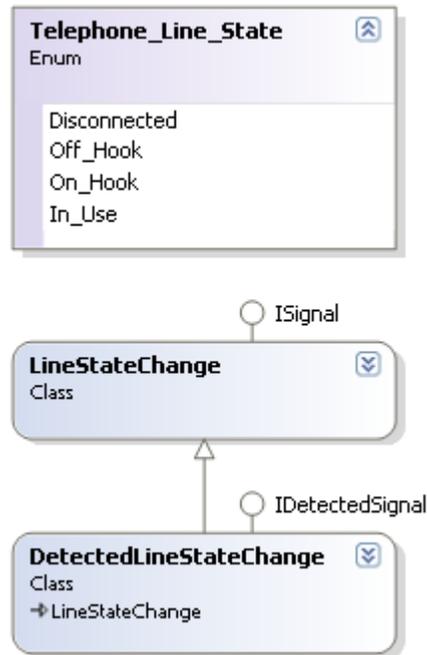
Figure 18 LineStateChange Class Diagram

14.13.1. LineStateChange Class

The LineStateChange class describes a change to a particular telephone line state as outlined in section 13.4. The LineStateChange class exposes the following members:

- **Name** – returns a description of this signal
- **State** – returns the new telephone line state

14.13.2. DetectedLineStateChange Class

As the name suggests, the DetectedLineStateChange class describes a change in telephone line state as detected by an instrument. Typically objects of this type are created by device objects when a change in line state is detected and reported through notifications described in section 10.1.3. The DetectedLineStateChange class inherits from the LineStateChange class and exposes the following additional members:

- **Time** – reports the instant in time corresponding to the change in line state.

In future releases this class may report more specific information regarding the detected change in telephone line state including
- The prior line state (if known)
- Measurements or analysis which may apply to the change in state

14.14. Telephone Line Reversals

Another signaling method within analog telephony is a line reversal which can be characterized as a reversal in polarity of the line voltage as measured between the two signal conductors (see section 13.5). Line reversals are described by several different classes illustrated below.
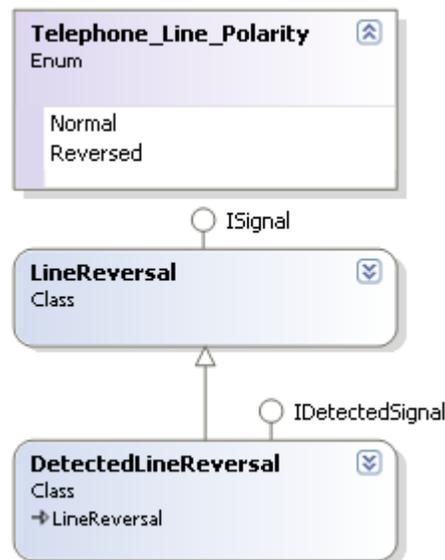
Figure 19 Line Reversal Class Diagram

Each instrument supported by aiDevices defines a particular polarity as "Normal". Typically this polarity will result in negative line voltage measurements from the telephone interface since the "normal" DC voltage feed is traditionally specified as -48 Volts. Please note:
- Changes in line polarity are specified relative to the instrument specific "normal" polarity
- Many telephone cables are wired such that the tip and ring conductors may be reversed resulting unexpected telephone line polarities.

14.14.1. LineReversal Class

The LineReversal class describes a change to a particular telephone line polarity as outlined in section 13.5. The LineReversal class exposes the following members:

- **Name** – returns a description of this signal

- **Polarity** – returns the new telephone line polarity (see section 13.5)

14.14.2. DetectedLineReversal Class

As the name suggests, the DetectedLineReversal class reports a change in telephone line polarity as detected by an instrument. Typically objects of this type are created by device objects when a reversal is detected and reported through notifications described in section 10.1.3. The DetectedLineReversal class inherits from the LineReversal class and exposes the following additional members:

- **Time** – reports the instant in time corresponding to the change in line reversal.

> In future releases this class may report more specific information regarding the line reversal

14.15. Line Flash Signals

Within analog telephony a line flash (also known as hook flash) is a signaling method used by Terminal Equipment (TE) to signal a central office by quickly hanging up and then picking up again. Historically this signal has been used to activate features such as call waiting or three way calling. Regardless of the usage, the line flash signal can be generally defined as

- A transition in telephone line state

  - Starting 'off hook'; then

  - Briefly 'on hook'; then

  - Ending 'off hook'

- Where the duration of the on-hook period can vary but generally does not exceed a second

Some of the instruments supported within aiDevices have the capability to bridge the telephone line to monitor and report signals. With this in mind the aiDevices framework also allows an additional definition of a line flash as:

- A transition in telephone line state

  - Starting 'in use'; then

  - Briefly 'on hook'; then

  - Ending 'in-use'

14.15.1. LineFlash Class

The LineFlash class describes a line flash signal defined in section 14.15 at a very high level without regard to particular line states. This class defines the following members:

- **Duration** – this returns the duration on the line flash.

14.15.2. DetectedLineFlash

The DetectedLineFlash class describes a particular line flash signal as detected by an instrument. DetectedLineFlash objects are exclusively created by device classes when a line flash is detected and are reported through notifications described in section 10.1.3. The DetectedLineFlash class inherits from the LineFlash class and exposes the following additional members:

- **Time** – reports the instant in time corresponding to the start of the line flash

- **LeadingEdge** – returns the DetectedLineStateChange object (see section 14.13.2) which corresponds to the beginning of the line flash signal

- **TrailingEdge** – returns the DetectedLineStateChange object that corresponds to the end of the line flash signal

In future releases this class may report more specific information regarding this detected signal possibly including measurements.

14.16. Open Switching Interval (OSI) Signals

Within analog telephony an Open Switching Interval (OSI) is a signaling method which can be used by a central office to signal a phone or other terminal equipment. Depending on the region and context this signal can be used to:

- Signal the terminal equipment to release the line (hang up)

- Signal the terminal equipment that Caller ID is about to be delivered.

Regardless of the usage the OSI signal can be generally defined as

- A temporary removal of DC feed which causes a brief transition into the 'disconnected' state; where

- The duration of the disconnected period can vary but generally does not typically exceed one second

14.16.1. OSI Class

The OSI class describes an open switching interval defined in section 14.16 at a very high level without regard to particular line states. This class defines the following members:

- **Duration** – this returns the duration of the signal

- **Name** – this returns a description of the signal

This class can be used:

- To define an OSI signal to be generated by a supporting instrument

- To report OSI information without regard to the specific changes in line state

14.16.2. DetectedOSI Class

The DetectedOSI class describes an OSI signal as detected by an instrument. DetectedOSI objects are exclusively created by device classes when an OSI is detected and is reported through notifications described in section 10.1.3. The DetectedOSI class inherits from the OSI class and exposes the following additional members:

- **Time** – reports the instant in time corresponding to the start of the OSI

- **LeadingEdge** – returns the DetectedLineStateChange object (see section 14.13.2) which corresponds to the beginning of the signal

- **TrailingEdge** – returns the DetectedLineStateChange object that corresponds to the end of the signal

In future releases this class may report more specific information regarding this detected signal possibly including measurements.

14.17. Pulse Dialing Signals

Before the advent of touch tone dialing, analog telephones used pulse dialing as a method to signal the central office with the desired telephone number to call. Each digit was signaled by repeatedly going on and off hook in rapid succession where the digit is specified by the number of resulting pulses. The aiDevices framework defines two classes which describe pulse dialing signals.

14.17.1. PulseDialingDigit Class

The PulseDialingDigit class represents a single pulse dialing digit without regards to timing or particular line state transitions. This class exposes the following members:

- **Digit** – this returns a character corresponding to the dialed digit represented by this signal ('0' to '9')

- **AverageMakeInterval** – this returns the average time interval between on-hook pulses. For the digit '1' this returns zero

- **AverageBreakInterval** – this returns the average duration of the on-hook pulses in this signal

- **PulsesPerSecond** – this returns the number of pulses per second expressed as a frequency

- **BreakPercentage** – this returns the ratio of the average on-hook interval to the average period of the signal.

- **PulseCount** – this returns the number of pulses within this digit

- **Duration** – this returns the duration of this signal

- **Name** – returns a description of this signal

14.17.2. DetectedPulseDialingDigit Class

The DetectedPulseDialingDigit class describes a particular instance of a pulse dialing digit that has been detected and reported by an instrument. This class inherits the PulseDialingDigit class and exposes the following additional members:

- **Edges** – this returns a list of the DetectedLineStateChange objects which comprise the detected pulse dialing digit. Applications can derive their own statistics from these signals if desired.

- **Time** – this returns the time corresponding to the first edge of the pulse dialing digit

14.18. Frequency Shift Keying (FSK) Signals

Many signaling methods (such as Caller ID) use the Frequency Shift Keying (FSK) modulation technique in order to convey digital information within the voice band of a telephone network. A simplified illustration is shown in Figure 20. An FSK signal is one in which

- A sinusoidal carrier signal is modulated to convey binary information by shifting between two discrete carrier frequencies; where

    o The frequency assigned to a logic '1' is referred to as the mark frequency

    o The frequency assigned to a logic '0' is referred to as space frequency

- Each bit is transmitted in a serial fashion and each will modulate the carrier signal for an equal time

- The baud rate describes the rate at which bits are modulated in the FSK signal (and is the inverse of the bit duration)
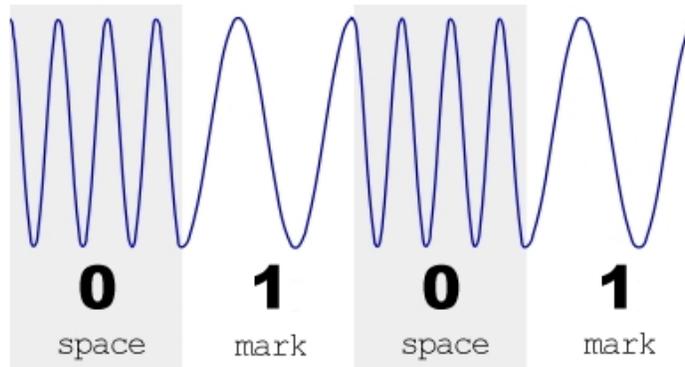
Figure 20 FSK Signal Example

When describing an FSK signal often it is convenient to separate the signal into logical layers which can be treated separately during synthesis and analysis. These layers are:

- **Data Layer** – this describes the "message" that is conveyed by the FSK signal which is independent of any formatting or encoding specification

    o General FSK data formats are documented in section 14.18.4

    o Caller ID Data formats are documented in section 15

- **Presentation Layer** – this layer is concerned with the encoding of the message contents into a bit pattern suitable for FSK modulation as well as any extra signaling required apart from the message contents.

    o Caller ID presentation layer is documented in section 15.1

- **Physical Layer** – this layer describes the modulation parameters required to modulate the FSK signal. This includes mark and space frequencies, signal level, baud rates, etc.

    o Physical signaling settings are documented in section 14.18.1.

    o Bit patterns used to modulate FSK are documented in section 14.18.2.

14.18.1. FSK Physical Settings

The FSKPhysicalSettings class is responsible for describing all the physical parameters required to modulate an FSK signal. This class does not deal with any details relating to the modulating signal but rather exposes the following members:

- **MarkFrequency** – specifies the frequency of the mark signal

- **SpaceFrequency** – specifies the frequency of the space signal

- **MarkLevel** – specifies the signal level of the mark carrier

- **SpaceLevel** – specifies the signal level of the space carrier

- **Twist** – reports the ratio of the mark and space level.

- **Baud** – specifies the rate at which bits will be modulated

- **MarkBitDuration / SpaceBitDuration** – these properties specify the duration of mark and space bits independently which can be used to simulate bit-skew. In an ideal signal these values should be identical and the inverse of the baud rate.

This class also exposes several static functions which help construct signal settings based on standard definitions:

- **GetMarkFrequency / GetSpaceFrequency** – returns the nominal signaling frequencies specified by a standard body which is passed by argument

- **GetBaud** – returns the nominal baud rate specified by a standard body which is passed by argument

- **GetDefaults** – these overloaded functions return nominal FSK signal settings specified by a standard body which is passed by argument

- **CalculateTwist** – this function assists in twist calculations for FSK signals (which differ from those for dual tone signals!)

Examples:
```
' Note: named parameters are shown for clarity but are not required!
' 1 Vrms FSK with 1200+2200 Hz carriers and 1200 baud
Dim S = New FSKPhysicalSettings(MarkFrequency:=Frequency.InHz(1200), _
                                SpaceFrequency:=Frequency.InHz(2200), _
                                CarrierLevel:=SignalLevel.InVrms(1),
                                Baud:=Frequency.InHz(1200))

' Fully specified FSK physical parameters
S = New FSKPhysicalSettings(MarkLevel:=SignalLevel.IndBm(-4), _
                       Spacelevel:=SignalLevel.IndBm(-3),
                       MarkFrequency:=Frequency.InHz(1203),
                       SpaceFrequency:=Frequency.InHz(2207),
                       MarkBitDuration:=TimeInterval.InMilliseconds(0.833),
                       SpaceBitDuration:=TimeInterval.InMilliseconds(0.831))

' Nominal FSK settings for ETSI Caller ID and -10dBV carrier level
S = FSKPhysicalSettings.GetDefaults(CallerIDStandardBody.ETSI,
                                    SignalLevel.IndBV(-10))

' Nominal FSK settings for TIA Caller ID, -10dBV average level, 3 dB Twist
S = FSKPhysicalSettings.GetDefaults(CallerIDStandardBody.TIA_Telecordia,
                                    AverageLevel:=SignalLevel.IndBV(-10), _
                                    Twist:=UnitlessQuantity.IndB(3))

' Returns nominal ETSI mark frequency plus 1 percent
FSKPhysicalSettings.GetMarkFrequency(CallerIDStandardBody.ETSI) * 1.01
' Returns nominal TIA baud rate less 2 percent
FSKPhysicalSettings.GetBaud(CallerIDStandardBody.TIA_Telecordia) * 0.98
```

14.18.2. Bit Patterns

An FSK signal can generally described as a sinusoidal carrier which switches between two distinct frequencies depending on the binary modulation signal; where '1' corresponds to the mark carrier frequency and the '0' corresponds to the space carrier frequency (see Figure 20). The modulating signal can then be described as a sequence of bits which sequentially modulate the carrier to produce the FSK signal. The BitPattern class represents this sequence of bits and is used internally to specify the FSK modulating signal. The BitPattern class exposes the following members:

- **BitCount** – returns the number of bits within the pattern

- **Clear** – this method removes all bits from within the pattern

- **AddBit** – this method appends a bit to the end of the pattern

- **SetBit** – this methods sets a bit value at a particular index in the pattern

- **GetBit** – this function returns a bit value at a specific index within the pattern

- **AddSetBits** – this method adds a sequences of '1' bits to the pattern

- **AddClearedBits** – this method adds a sequence of '0' bits to the pattern

- **AddAlternatingBits** – this method adds a sequence of alternating '1' and '0' bits to the pattern. Note: this alternating bit sequence is often used to generate "channel seizure" signals in Caller ID and other standards.

- **AddByte_LSBFirst** – this method adds **exactly 8 bits** as specified to the bit pattern starting with the least significant bit

- **AddByte_MSBFirst** – this method adds **exactly 8 bits** as specified to the bit pattern starting with the most significant bit

This class is useful for creating modulation patterns which are not natively supported by aiDevices however for specific data formats which are supported by aiDevices developers are urged to use the higher level classes defined in later sections.

> Unlike other descriptor classes the BitPattern class is mutable; which means the contents can be modified after an object is created.

14.18.3. Byte Patterns

Many signaling methods which transport binary data (such as Caller ID) encode messages using a sequence of 8-bit symbols (bytes). The BytePattern class represents just such a sequence of bytes which can be used to generate an FSK modulating signal. The BytePattern class exposes the following members:

- **ByteCount** – returns the number of bytes in the sequence

- **CheckSum** – this property specifies a checksum calculator object which will be automatically updated with all bytes added to the pattern (see section 14.19).

- **Clear** – this method clears all bytes from the pattern and resets the checksum calculator

- **Add** – this method adds a bytes, character, or string to the end of the byte pattern and updates the checksum calculator if specified

The BytePattern class also has support for calculating parity of character values. The Add method has an overload which can automatically calculate the parity of a character as it is added to a data pattern. In addition the BytePattern class exposes the following static functions:

- **CalculateParity** – returns the byte value of a character when the specified parity setting is applied

While this class is useful for defining arbitrary data patterns it is most useful as a base class for specific data formats such as FSKData and FSKCallerIDData.

> Unlike other descriptor classes the BytePattern class is mutable; which means the contents can be modified after an object is created.

14.18.4. FSK Data

The FSKData class represents a generic pattern of bytes which can be sent within an FSK transmission. Typically this class is used:

- To describe custom FSK message contents or report those which do not conform to formats supported by aiDevices

- As a base class for well known message formats like Caller ID

The FSKData class exposes the following members:

- **Name** – this returns a descriptive name of the message. Derived classes which implement specific formats overload this property to describe message contents.

- **Problems** – this returns a list of objects that describe any problems with the message format. Generally this only applies to derived classes which implement specific message formats.

14.18.5. FSK Transmissions

The FSKTransmission class represents a single FSK signal and contains all the settings and data necessary to transmit the signal. The FSKTransmission class exposes the following members:

- **PhysicalSettings** – specifies the physical parameters for the FSK signal (see section 14.18.1).

- **BitPattern** – specifies the bit pattern which will modulate the FSK signal (see section 14.18.2).

- **Duration** – returns the duration of the transmission based on the number of bits and baud rate.

- **Name** – returns a descriptive name for this signal

The FSKTransmission class may be used by applications to specify custom modulation patterns which may not be natively supported by aiDevices however typically applications will use sub-classes of the FSKTransmission classes which deal with specific FSK encodings such as Caller ID (see section 15).

```
Example:

// Create an FSK transmission with default physical settings
FSKTransmission F = new FSKTransmission(
          FSKPhysicalSettings.GetDefaults());

// You can modify the BitPattern after creation!
F.BitPattern.AddAlternatingBits(200, false);

//TODO: encode more bits here!
```

14.19. Checksum Calculations

Many data transmission formats (like Caller ID) include a fixed length number called a checksum which is calculated from the message contents and is used to detect errors in transmission. The aiDevices framework abstracts the calculation of these checksums using classes which derive from the CheckSumCalculator class. Instances of these checksum calculators can then be passed to data pattern objects which require checksum calculations such as BytePattern (section 14.18.3), FSKData (section 14.18.4), and FSKCallerIDData (section 15.3).

The abstract CheckSumCalculator class exposes the following members:

- **Clear** – clears the contents of the checksum calculator and resets back to initial state.

- **Add** – adds a byte to the checksum calculation.

- **Value** – this property returns the current result of the checksum calculation; which may be one or more bytes.

At present the aiDevices framework contains the following checksum calculator classes:

- **InvertedModulusChecksumCalculator** – this calculates the 2's compliment inverse of the 8-bit sum of the message data. This format is compatible with TIA and ETSI Caller ID messages discussed in section 15.3.

In future the aiDevices framework will support additional checksum formats such as the NTT CRC calculations.

15. Caller ID (FSK) Classes

Caller ID is a general term which refers to a caller identification service which is available in many phone networks throughout the world. These systems typically involve in-band signaling sent from the central office to a phone (or other termination equipment) when a new incoming call is established. Caller ID systems vary widely in terms of capabilities and technologies but typically most services will

- Deliver telephone number and/or name of the calling party.

- Deliver information regarding new voice mail messages.

- Deliver this caller information when a new incoming call begins with no prior call established or when an incoming call begins while an existing call is in progress; generally used in conjunction with call waiting features.

The signaling methods and common practices for Caller ID deliver vary widely depending on equipment manufacturer and region however a large number of Caller ID systems use Frequency Shift Keying (FSK) signaling as described in section 14.18. In addition to the message contents many Caller ID delivery schemes involve other signaling methods and sequences to notify terminal equipment of incoming Caller ID information. These other signals can include:

- **CAS** – this signal is typically sent in advance of off-hook Caller ID to notify the phone of incoming FSK so it can mute the receiver to avoid disturbing the user with the sudden and annoying sounds (see section 14.10).

- **Subscriber Alerting Signal (SAS)** – this signal is sent in advance of CAS to alert the customer (not the TE) of an incoming call which may be available through a call waiting service. The specifics of this signal are not universal and may include single tones or even voice recordings.

- **Line Reversal** – some systems may reverse the polarity of the telephone line in advance of Caller ID transmissions (see section 14.14).

- **Open Switching Interval (OSI)** – some systems will generate an OSI in advance of Caller ID transmissions (see section 14.16).

- **DTMF ACK** – some systems require acknowledgement of the CAS signal sent prior to Caller ID in which the phone (or TE) will send an acknowledgement back to the central office by means of a DTMF digit (see section 14.9).

Many Caller ID standards exist which are designed to ensure consistent interoperation of FSK based Caller ID transmissions throughout the world. The following sections contain many features which are designed based on the following standards

- Telecommunications Industry Association (TIA)

  o TIA-777-A

- European Telecommunication Standards Institute (ETSI)

  o ETS 300 778

15.1. Caller ID Transmission

Most Caller ID transmissions follow a common signaling and encoding scheme for generating the modulated FSK signal illustrated in Figure 21.
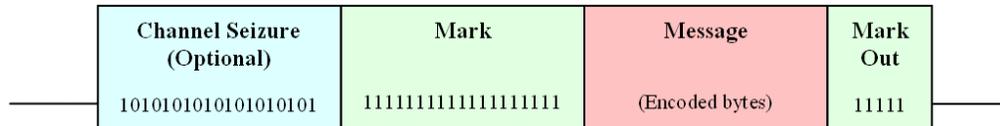
Figure 21FSK Caller ID Message Encoding Scheme

- **Channel Seizure (optional)** – many on-hook Caller ID transmissions start with a sequence of alternating mark and space bits which can be used as a qualification and initialize FSK demodulators. Channel seizure is not generally present in off-hook transmissions.

- **Mark Signal** – most Caller ID transmissions precede the message data with a continuous mark signal (string of '1' bits). Most receivers will require minimum mark signal duration as a qualification to decode the FSK message contents.

- **Message Bytes** – following the mark signal will be a sequence of bytes which comprise the contents of the Caller ID message such that each byte is encoded as follows:

  - **Start Bit** – each byte is preceded by a "start" bit consisting of a single space bit.

  - **Byte Value** – each byte value is encoded LSB first immediately following the start bit

  - **Stop Bit(s)** – following each byte is one or more mark bits referred to as "stop" bits. The duration of the stop bits is allowed to vary from byte to byte.

- **Mark Out** – usually the message bytes are followed by a short duration of mark signal before the carrier is terminated. This signal is required by some receivers to properly decode the final byte of the message.

The **FSKCallerIDTransmission** class represents an FSK transmission which adheres to the signaling and encoded scheme above. Note however that this class specifies only the physical and presentation layers of the FSK transmission (see section 14.18) and does not specify the message contents. The contents of the message are encoded separately via one of the following classes

- **FSKData** – see section 14.18.4 for generic message formatting.

- **SDMF**– see section 15.3.1 for the class documentation and section 15.4 for helper functions which create common messages based on high level parameters.

- **MDMF** – see section 15.3.2 for the class documentation and sections 15.4 and 15.5 for helper functions which create these messages based on high level parameters.

- **Custom Classes** – any class which inherits from FSKData and implements all of its features may be used to specify the contents of Caller ID messages.

The FSKCallerIDTransmission inherits from FSKTransmission and exposes the following additional members:

- **Message** – returns a class which specifies the byte pattern which is encoded as the message contents and is used to update the modulating bit pattern before transmission. Typically applications will specify a high level message format object however they may specify the data pattern using any subclass of BytePattern.

- **ChannelSeizurebits** – this specifies the number of alternating bits which will be inserted at the start of the FSK transmission.

- **MarkBits** – this specifies the duration of the mark interval before the message contents in bits.

- **StopBits** – this specifies the number of stop-bits which are inserted after each byte encoded in the message

- **MarkOutBits** – this specifies the number of mark bits which are inserted after the message contents.

- **RegenerateBitPattern** – this method will regenerate the bit pattern (exposed by the FSKTransmission) based on the current settings and data message contents specified within the FSKCallerIDTransmission object.

Unlike many other signaling objects defined within the aiDevices framework the FSKCallerIDTransmission object is mutable; which means that the object contents can be modified after the object is created.

The FSKCallerIDTransmission **automatically regenerates the bit pattern** contained within the internal BitPattern object whenever:
- Any of the FSKCallerIDTransmission parameters are modified
- The contents of the data pattern specified in the Message parameter are modified
- The RegenerateBitPattern method is called

However the **bit pattern is not automatically updated** when the contents of the BitPattern object are manipulated. This feature is essential in order to allow applications to intentionally create malformed bit patterns for compliance testing purposes.

Example:

```vbnet
' Creates SDMF 'Calling Number' message with the current date/time
' Note: notice the message and transmission are specified separately!
 Dim Msg = TIA.CallingNumberDeliveryMessage( CallerIDDateTime.Now, _
                                             "5551234")

' Creates an Caller-ID transmission which contains the above message
' with channel seizure, mark, stop bits, markout etc.
 Dim CID = New FSKCallerIDTransmission(Msg, _
                    ChannelSeizureBits:=80, _
                    MarkBits:=180, _
                    Stopbits:=1, _
                    MarkOutBits:=2, _
                    PhysicalSettings:=FSKPhysicalSettings.GetDefaults)

' Now we can manipulate the underlying bit pattern!
' Note: these changes will be overwritten if any higher level
' property is modified!
CID.BitPattern.SetBit(5, False)

' this change overwrites the above change to BitPattern!
 CID.MarkBits = 5
```

15.1.1. Custom Bit Encoding

By default the FSKCallerIDTransmission will automatically generate a bit pattern corresponding to the format illustrated in Figure 21. There may however be situations where applications may want to extend this functionality and encode the message contents in a custom fashion. Typical reasons for this may include:

- Stop bit elongation

- Intentional data impairments such as invalid checksums, bad stop bits, malformed channel seizure etc

- Customized or non-standard encodings

This situation is addressed by means of the BitPatternEncoder delegate declared within the FSKCallerIDTransmission class. Applications may override the default bit encoding by specifying a delegate method in the constructor which will be called whenever the bit pattern needs to be regenerated.

The FSKCallerIDTransmission class exposes the static GenerateDefaultBitPattern method which encodes the default bit pattern. This may be called within the custom delegate if applications simply want to modify the existing encoding.

The BitPatternEncoder delegate method must remain accessible for the entire lifespan of the FSKCallerIDTransmission object. Application developers are urged to use static (shared) methods when specifying this delegate to guarantee this behavior.

Example:

```csharp
// NOTE: The following 3 lines are a snippit and must be
// placed within a method!

FSKCallerIDData Msg =  TIA.CallSetupMessage(
                          CallerIDDateTime.Now(),
                           "John Doe", "5551324");
FSKPhysicalSettings Phys = FSKPhysicalSettings.GetDefaults();

// Creates a transmission with custom bit pattern encoder!
FSKCallerIDTransmission X = new FSKCallerIDTransmission(Msg,
                0, 100, 1, 2,
                Phys, ElongateStopBits); // <- Custom encoder



// This will be called whenver the bit pattern in the above
// transmission needs to be re-generated
static void ElongateStopBits(FSKCallerIDTransmission TX)
{
    BitPattern P = TX.BitPattern;
    P.Clear();
    // Channel Seizure
    P.AddAlternatingBits(100, false);
    // Mark time
    P.AddSetBits(TX.MarkBits);
    // Message contents
    for (int i = 0; i < TX.Message.ByteCount; i++)
    {
        // Start Bit
        P.AddClearedBits(1);
        // Byte Value
        P.AddByte_LSBFirst(TX.Message.GetByte(i));
        // Stop Bit
        P.AddSetBits(1);
        //Elongate every 8th stop bit by 10 bits!
        if ((i % 8) == 0) P.AddSetBits(10);
    }
    // Mark Out
    P.AddSetBits(TX.MarkOutBits);
}
```

15.2. Caller ID Date and Time

Many Caller ID messages deliver date and time information to inform the TE (phone) of the time when the incoming call occurred. Most of the Caller ID standards use the following encoding format for this date and time information:

- The date/time information is encoded using a string of 8 to 10 characters (depending on the parameter or meaning) where

- Each component of the time is represented with two digits and

- Each digit can range from '0' to '9'

The most common date/time format (which is used in TIA and ETSI Caller ID) is referred to within the aiDevices framework as the MMDDHHmm format where:

- The first two characters represent the month of the year and may range from "01" to "12"

- The next two character represent the day of the month and may range from "01" to "31"

- The next two characters represent the hour of the day (in 24 hour format) and may range from "00" to "23"

- The final two characters represent the minute within the hour and may range from "00 to "59"

A less common date/time format (used within some ETSI parameters) is referred to with the aiDevices framework as the internal format which is exactly like the MMDDHHmm format except:

- An additional two characters are appended to the end which indicate the seconds in the minute and range from "00" to "59"

The **CallerIDDateTime** class deals with the date/time issues and exposes the following members:

- **Month** – returns the month component of the date/time

- **Day** – returns the day component of the date/time

- **Hour** – returns the hour component of the date/time

- **Minute** – returns the minute component of the date/time

- **Second** –returns the second component of the date/time

- **ToString** – the ToString function is overloaded to return any of the standard date/time formats which can be used within a Caller ID message. By default it returns the date information in a typical format i.e. "Nov 3 3:30 pm"

- **GetProblems** – returns a list of problems which may be present with the date format

This class also exposes the following static functions:

- **Now** – this returns the current date/time information based on the host clock

Examples:

```vb
' get the current time
Dim DT = CallerIDDateTime.Now

' Create a specific day/time
DT = New CallerIDDateTime(Month:=12, _
                          Day:=11, _
                          Hour:=5, _
                          Minute:=12)

' Each return value is commented
DT.ToString()    ' Dec 11 5:12 AM
DT.ToString(CallerIDDateTime.Formats.MMDDHHmm)  ' 12110512
DT.ToString(CallerIDDateTime.Formats.MMDDHHmmss)  ' 1211051200
```

15.3. Caller ID Message Formats

Since most Caller ID messages can convey similar information regardless of format each of the classes which manage Caller ID messages derives from the abstract **FSKCallerIDData** class. This class derives from FSKData and exposes several members which apply specifically to Caller ID and can be used to extract and manipulate high level information regardless of format.

Unlike many other objects within the aiDevices framework **all Caller ID data format classes are mutable** which means that the data contents can be modified after the object is created. This feature is essential in order to allow applications to intentionally created malformed message contents for compliance testing purposes.

The FSKCallerIDData class exposes the following members:

- **IsFormatValid** – this returns false if the basic structure of the data prevents proper interpretation of the message contents, otherwise this returns true and the remaining properties can be considered valid. Typically this will only become false when the message contents are modified.

- **MessageType** – specifies a byte which determines the meaning and format of the data message.

- **MessageLength** – specifies the length of the message **as encoded within the message**.

- **Parity** – specifies the parity settings used to encode characters within the message

- **ContainsCallingNumber** – returns true if the message contains the calling number information

- **CallingNumber** – returns the calling number if contained within the underlying message. Otherwise this returns an empty string.

- **ContainsCallingName** – returns true if the message contains the name of the calling party

- **CallingName** – returns the calling name information if contained within the underlying message. Otherwise this returns an empty string.

- **ContainsDateTime** – returns true if the date/time information is contained within the underlying message.

- **DateTime** – returns the date and time information if contained within the underlying message. Otherwise this returns null.

All the well supported Caller ID message formats are implemented in classes which derived from FSKCallerIDData as shown in
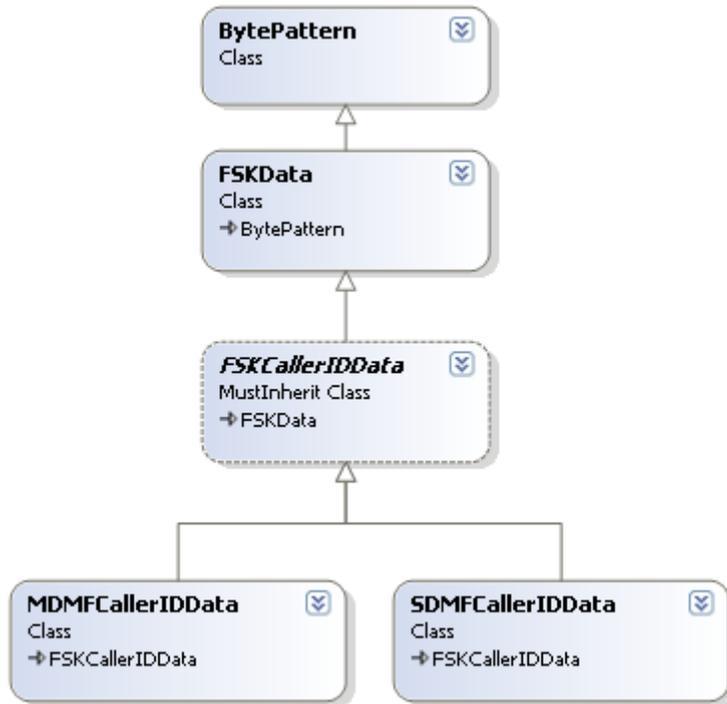
Figure 22FSK Caller ID Message Format Class Diagram

15.3.1. SDMF Message Format

One of the simplest Caller ID message formats (specified by the Telecommunications Industry Association in TIA-777-A) is the Single Data Message Format (SMDF) which is encoded as illustrated below.

Type Byte
Length Byte
Message Contents   **[1 to 255 bytes]**
Checksum Byte

Figure 23 SDMF Caller ID Format

The format of the contents is indicated by the message type byte. The two well defined formats are:

- **Calling Number Delivery (Type 0x4)** which contains:
    - Date/Time
    - Calling Number (or reason for absence if not specified)

- **Visual Message Waiting Indicator (Type 0x6)** which contains:
    - A flag which indicates if a new message is available

> This section is a brief summary of the SDMF format. Developers are urged to refer to TIA-777-A for a complete specification

The SDMFCallerIDData class is used to represent this SDMF message format. It inherits from FSKCallerIDData and exposes the following additional members:

- **Contents** – this specifies the contents of the SDMF message interpreted as a string.

While this class can be used to create any SDMF message the most common messages can be easily constructed using the method declared in section 15.4.

Examples:

```
' 555-1234 called on Dec 25 at 8:15
Dim SDMF = New SDMFCallerIDData(4, "122508155551234")
' Same as above
SDMF = TIA.CallingNumberDeliveryMessage( _
                New CallerIDDateTime(12, 25, 8, 15),
                "5551234")

' A private number called on Dec 25 at 8:15
SDMF = New SDMFCallerIDData(4, _
                    New CallerIDDateTime(12, 25, 8, 15), _
                    "P")
' An "Out of area" number called on Dec 25 at 8:15
SDMF = TIA.CallingNumberDeliveryMessage( _
                    New CallerIDDateTime(12, 25, 8, 15), _
                 TIA.ReasonForAbsence.Out_Of_Area)
```

15.3.2. MDMF Message Format

A popular Caller ID message format is Multiple Data Message Format (MMDF) which is specified by both the TIA and ETSI standard bodies. The basic structure of the encoding is illustrated below.

Message Type
Message Length
Parameter 1 Type
Parameter 1 Length
Parameter 1 Contents
(Parameter 1 Length Bytes)
Parameter 2 Type
Parameter 2 Length
Parameter 2 Contents
(Parameter 2Length Bytes)
…
Checksum Byte

Figure 24 MDMF Caller ID Format

The meaning of each MDMF message is usually conveyed by the message type byte and includes:

- **Call Setup (Type 0x80)**

- **Visual Message Waiting Indicator (Type 0x82)**

> This section is a brief summary of the MDMF format. Developers are urged to refer to one of the following standards for a complete specification.
> - TIA-777-A
> - ETS 300 778 or ETS 300 659

The MDMF class inherits from FSKCallerIDData and represents this MDMF format by exposing the following members:

- **ParameterCount** – returns the number of parameters in the message

- **GetParameter** – returns a parameter at a specific index in the message

- **InsertParameter** – inserts a parameter at a specific place in the message

- **RemoveParameter** –removes a parameter from the message

Each MDMF parameter is represented using the MDMFParameter class which exposes the following members:

- **ParameterType** –specifies the type value for the parameter

- **Name** – returns a descriptive name of the parameter which is based on the names from the TIA and ETSI specifications

- **Length** – returns the length of the parameter (not including the type and length bytes)

While this class can be used to create any MDMF message the most common messages can be easily constructed using the method declared in sections 15.4 and 15.5.

Example:

```
// Call Setup message
MDMFCallerIDData Msg = new MDMFCallerIDData(0x80, // Call Setup
            new MDMFParameter(1, "12250815"),   // Date/Time
            new MDMFParameter(2, "5551234"),    // Number
            new MDMFParameter(7, "John Doe"));  // Name

// Same as above using TIA helper method
Msg = TIA.CallSetupMessage(CallerIDDateTime.Now(),
                    "John Doe", "5551234");

// Scan through each parameter
for (int i = 0; i < Msg.ParameterCount; i++)
{
    Debug.Print(Msg.GetParameter(i).Name);
}

// We can insert parameters into the message
Msg.InsertParameter(0, new MDMFParameter(3, "18005551234"));

// parameters can be creating using byte array arguments
Msg.InsertParameter(0, new MDMFParameter(100,
                    new byte[]{1,2,3,4,5}));
```

15.4. TIA Messages

The Telecommunications Industry Association (TIA) defines many of the Caller ID formats and signaling specifications for North America. The aiDevices framework includes a facade class named TIA which contains helper definitions including:

- Construction functions for creating standard message types

- Enumerations for message types, parameter types, and standard values.

> The standard message type and parameters type values defined in TIA-777-A are specified by the **TIA.MessageType** and **TIA.ParameterType** enumerations with comparable names. For example:
>
> TIA.MessageType.Call_Setup (which has a value of 0x80)

This class contains mostly static functions which assist in the creation of MDMF and SDMF messages. These functions include:

- **CallingNumberDeliveryMessage** – this returns an SDMF message containing date/time and calling number information, or date/time and reason for number absence

- **SDMF_VMWI_Message** – this returns an SDMF Visual Message Waiting Indicator (VMWI) message.

- **CallSetupMessage** – returns a standard MDMF Call Setup Message

- **MDMF_VMWI_Message** – returns an MDMF Visual Message Waiting Indicator message

- **DateTimeParameter** – returns an MDMF Date/Time parameter with a specific date value

- **CallingNumberParameter** – returns an MDMF Calling Number parameter with the number specified

- **DialableDirectoryNumberParameter** – returns an MDMF DDN parameter

- **ReasonForAbsenceOfNumberParameter** – returns an MDMF parameter which contains the reason for the absence of the calling number

- **CallingNameParameter** – returns an MDMF parameter containing a calling name

- **CallQualifierParameter** – returns the MDMF parameter containing the Call Qualifier information

- **VMWI_Parameter** – returns the MDMF parameter containing the activate/deactivate VMWI information.

Examples:

```
MDMFCallerIDData msg;
FSKCallerIDData D;
CallerIDDateTime Time = CallerIDDateTime.Now();

// SDMF Calling Number Delivery
D = TIA.CallingNumberDeliveryMessage(Time, "5551234");

// SDMF Calling Number Delivery with number absent
D = TIA.CallingNumberDeliveryMessage(Time,
          TIA.ReasonForAbsence.Out_Of_Area);

// SDMF VMWI Activate message
D = TIA.SDMF_VMWI_Message(true);

//MDMF Call Setup message
D = TIA.CallSetupMessage(Time,
                    "Advent Instruments",
                    "6049444298");

//MDMF Call Setup message with absent number
D = TIA.CallSetupMessage(Time,
             "Someone",
             TIA.ReasonForAbsence.Private_Number);

//MDMF Call Setup message with both name and number absent
D = TIA.CallSetupMessage(Time,
             TIA.ReasonForAbsence.Out_Of_Area,
             TIA.ReasonForAbsence.Private_Number);

// MDMF Call Setup message with name absent
msg = TIA.CallSetupMessage(Time,
             TIA.ReasonForAbsence.Out_Of_Area,
              "5551234");

// Inserts an MDMF Dialiable Directory Number parameter
msg.InsertParameter(0,
     TIA.DialableDirectoryNumberParameter("18005551234"));
```

15.5. ETSI Messages

The European Telecommunications Standards Institute (ETSI) defines many of the Caller ID formats and signaling specifications for Europe. The aiDevices framework includes a facade class named ETSI which contains helper definitions including:

- Construction functions for creating standard message types

- Enumerations for message types, parameter types, and standard values.

The standard message type and parameters type values defined in ETSI 300 659 are specified by the **ETSI.MessageType** and **ETSI.ParameterType** enumerations with comparable names. For example: ETSI.ParameterType.Calling_Line_Identity (which has a value of 0x02)

This class contains mostly static functions which assist in the creation of MDMF messages. These functions include:

- **CallSetupMessage** – this returns an MDMF message containing date/time, calling name, and calling number. There are also several overloads which allow the user to specify reasons for absence of the name or number information.

- **MessageWaitingIndicatorMessage** – this returns an MDMF message containing the visual message waiting indication parameter.

- **CallTypeParameter** – returns the Call Type MDMF parameter

- **DateTimeParameter** – returns the Date/Time MDMF parameter containing a specified date/time

- **CallingLineIdentify** – returns the MDMF parameter containing the number of the calling party

- **CalledLineIdentityParameter** – returns the MDMF parameter containing the called party identification

- **ReasonForAbsenceOfCalledLine** – returns the MDMF parameter containing the reason for absence of the called line information

- **CallingPartyNameParameter** – returns the MDMF parameter containing the name of the calling party

- **ComplementaryDateTimParameter** –returns an MDMF parameter containing complementary date/time information which can be use to update clocks.

Example:

```
Dim M As MDMFCallerIDData
Dim Time = CallerIDDateTime.Now

' ETSI Call Setup Message
M = ETSI.CallSetupMessage(Time, "Jane Doe", "5551234")


' ETSI Message Waiting Indicator Message
M = ETSI.MessageWaitingIndicatorMessage(IndicatorOn:=True)

' You can use static functions to create MDMF parameters!
M = New MDMFCallerIDData(ETSI.MessageType.Call_Setup, _
                ETSI.CallingPartyNameParameter("A. Caller"), _
                ETSI.CallTypeParameter(ETSI.CallType.VoiceCall))
```

16. Device Support Classes

As mentioned in section 8.2, the majority of instrument features are managed by support classes which can be accessed through properties of a device object. Most of these support classes are reused on several instruments with similar features and are documented in the following sections.

16.1. Signal Generators

Each different type of signal generator within an instrument is managed and abstracted through a separate support class. Each of the signal generator classes will typically expose the following interface:

- **Generate** – usually these methods will accept one or more signal descriptors (see section 14) and will typically start the generation of the signal described.

- **IsActive** – this property will return true if the signal generator is currently active generating a signal.

- **IsBusy** - this property will return true if the signal generator is either waiting to generate a signal or is actually generating a signal.

- **StopGenerator** – this method will immediately stop any signal generation

Each following sub-section documents a particular type of device support object which manages a signal generator. Please be aware of the following important notes.

> The Generate methods may start or schedule the generation of a particular signal but **do not wait for the signal to complete!** This is due to the fact that many signals specifications require a very long time to complete or may continue indefinitely. If an application must wait until the completion of the signal being generated it should:
> - Use the *.Wait.Until(...)* features documented in section 16.3
> - Respond to Notifications documented in section 10

> All signal generator levels are specified as **open circuit levels** and will be accurate so long as the generator is not connected to any loads (terminations). To obtain the desired signal level when the generator is terminated you will need to adjust the generator level to compensate for the source and termination impedance as discussed in section 13.1.1.

Many signal generators support scheduling through a single TimeStamp argument of the Generate method which is usually named 'AtTime'. This single argument specifies the time (see section 9) when the specified signal should be started.

> Once the Generate method has been called to schedule a signal for transmission using a TimeStamp argument the signal generator will be 'busy' until the signal completes. During this process the generator will not be able to start another signal and may not be able to stop the generation of the scheduled signal.

> Applications must be aware of communication and processing delays when scheduling signals for transmissions. **If the specified transmission time has already passed signal generators will generate the specified signal immediately.**
>
> While delays can depend on various and unpredictable factors such as
> * The number of active USB connections
> * Baud rate (if COM port is used)
> * Speed of the host computer
> * Processing load on the application
>
> Applications should expect at least 10 to 20 milliseconds delay in sending commands to an instrument. For more complicated signals such as FSK these delays may be significantly longer. To avoid the effects of these delays, applications should:
> * Schedule the first transmission in a signaling sequence to occur a short time in the future (longer than the expected communication delay)
> * Use any advanced configuration option where available (i.e. FSK Generator offers a Configure method which will upload transmission information well in advance)

16.1.1. Tone Generator

One of the most fundamental types of signals which can be generated by most Advent Instruments products is the simple tone (see section 14.5). Each device object which supports simple tone generation will expose one or more ToneGenerator objects. Each ToneGenerator object exposes the following members:

* **Level** – specifies the signal level of the tone to generate

* **Frequency** – specifies the frequency of the tone to generate

* **Shape** – specifies the shape of the signal to generate (see section 14.3).

* **Phase** – specifies the phase of the tone being generated.

* **Generate** – these methods start the generation of the tone with either the current or specified settings.

* **Update** – these methods will update the tone generator with new level, frequency, and shape information.

* **StopGenerator** – this immediately stops the tone generator.

* **ResetToDefaults** – this resets the tone generator back to default settings

* **IsActive** – returns true if the tone generator is currently generating a tone

* **IsReserved** – returns true if the tone generator resource is reserved by another signal generator object. Note: When reserved all changes to this tone object will be ignored.

* **ReservedBy** – returns a string containing a description of the object which has reserved the tone generator.

Tone generators represent one of the most fundamental signal generators within most instruments. As such these generators are often reserved by higher level signal generators (FSK, DTMF, AM, etc). When these tone generators are reserved, accesses to their properties will be ignored as not to interfere with higher level signal generation

Tone generator objects do not generally allow for precise timing. To generate tones with much more accurate timing please see sections 16.1.2 and 16.1.5.

Example:

```
// configure Tone A with 1200 Hz sinusoid at -10 dBV
_5620.ToneA.Frequency = Frequency.InHz(1200);
_5620.ToneA.Level = SignalLevel.IndBV(-10);
_5620.ToneA.Shape = Waveshape.Sinusoiodal;

// start the tone with 90 degrees starting phase
_5620.ToneA.Generate(Phase.InDegrees(90));

Thread.Sleep(1000); // leave it active for approximately 1 second

// Update the generator with a Tone object
Tone NT = new Tone(SignalLevel.IndBm(-5), Frequency.InHz(900));
_5620.ToneA.Update(NT);

Thread.Sleep(1000); // wait again

// stop the generator!
_5620.ToneA.StopGenerator();
```

Defaults

The ToneGenerator object exposes a ResetToDefaults method which will reset only the one single tone generator's settings back to their default values. More specifically this method will:

- Immediately stop the tone generator (if active)

- Set the signal level to 0 Volts

- Set the generator frequency to 1 kHz

- Set the wave shape to sinusoidal

16.1.2. MF Generator

The MFGenerator class manages the generation of finite duration sequences of multi-tone signals. While the signals which can be generated may be complex the MFGenerator class defines a very simple interface:

- **Generate** – causes the MF generator to transmit the specified signal; either immediately or at a particular time

- **StopGenerator** – this immediately stops the MF generator

- **IsActive** – returns true if the MF generator is currently transmitting a signal.

The MF generator is capable of transmitting

- Multi-tone signals (or derived signal) documented in section 14.6.

- Multi-tone sequences documented in section 14.7.

Example:

```vb
' Note: real applications will need to wait for the generator to finish
With _7280.MFGenerator
     Dim Level = SignalLevel.IndBV(0)
     Dim Duration = TimeInterval.InMilliseconds(80)

      ' Generates a CAS
      .Generate(New CAS(Level, _
                TimeInterval.InMilliseconds(80)))

     ' Generate a single DTMF digit
       .Generate(New DTMFDigit("D"c, _
                       Level, _
                        Duration))

     ' Generated DTMF dialing
     Dim Dialing = DTMF.CreateDTMFDialing("12345", _
                                          Level, _
                                          Duration, _
                                          Duration)
     Duration = TimeInterval.InMilliseconds(333)

     ' 3 sequential tones with equal timing
      Dim SIT = MultiToneSequence.Create( _
                    "Special Information Tone)", _
                      Cadence.AdjacentTiming(Duration, _
                                              Duration, _
                                              Duration), _
                  New Tone(Level, Frequency.InHz(985)), _
                  New Tone(Level, Frequency.InHz(1428)), _
                  New Tone(Level, Frequency.InHz(1776)))

     ' Generate a sequence of 3 tones
     .Generate(SIT)
```

```
End With
```

16.1.3. FSK Generator

The FSKGenerator class manages the generation of FSK signals (see section 14.18) on supported instruments. This class exposes an interface which allows FSK to be generated based on:

- A modulating bit pattern (section 14.18.2) and physical settings (14.18.1); or

- A fully specified FSK transmission (section 14.18.5)

To this end the FSKGenerator class exposes the following members:

- **ResetToDefaults** – this resets the internal FSK generator settings back to default settings and clears any configured transmissions.

- **Configure** – this configures the generator with an FSK transmission so it may be transmitted at a later time with less latency

- **Generate** – these methods start the generation of an FSK signal either immediately or at a specific time

- **StopGenerator** – this stops any FSK transmissions in progress

In general when the FSK generator is started or stopped within an instrument the device object generates notifications which inform the application of the exact timing and status (see section 0).

Examples:

```
/* ---------------------------------------------
 * Example 1:  FSK Caller ID
 */
MDMFCallerIDData Msg;
FSKPhysicalSettings Phys;
FSKCallerIDTransmission TX;

// Create a Caller ID message
Msg = TIA.CallSetupMessage(CallerIDDateTime.Now(),
                           "John Doe", "5551234");
Phys = FSKPhysicalSettings.GetDefaults();

// Create an FSK transmission with mark, stopbits, and markout
TX = new FSKCallerIDTransmission(Msg,
                       0, 100, 1, 2,
                               Phys);
// Start the FSK generator!
_5620.FSKGenerator.Generate(TX);

/* ---------------------------------------------
 * Example 2:  FSK based on bit pattern
 */
```

```
BitPattern BP = new BitPattern();
BP.AddAlternatingBits(100, true);    // 101010101...
BP.AddSetBits(100);                  // 111111111...

// Generate FSK modulated with a bit pattern
_5620.FSKGenerator.Generate(BP,Phys);
```

16.1.4. AM Generator

The AMGenerator class manages the generation of Amplitude Modulated (AM) signals on supported instruments. The public interface is quite simplistic and includes:

- **Generate** – these methods start the generation of an amplitude modulated tone specified using an AMTone object (see section 14.5.1). This method will reserve two ToneGenerator objects available within the instrument.

- **StopGenerator** – this immediately stops the generation of an AM tone and releases any tone generators which were reserved within the instrument.

- **Update** – this updates the AM generator signal settings when an AM signal is being generated.

- **IsActive** – returns true when the AM generator is actively generating an amplitude modulated signal.

- **IsReserved** – returns true if the generator is reserved by another signal generator object. Note: When reserved all changes to this object will be ignored.

- **ReservedBy** – returns a string containing a description of the object which has reserved the generator.

To generate AM signals with more precise timing, developers should use the Pattern Generator documented in section 16.1.5.

The AM generator can be reserved by higher level signal generator objects. When these tone generators are reserved methods and property modifications will be ignored as not to interfere with higher level signal generation.

Examples:

```
// define an AM signal with peak level of 1 Vrms
AMTone AM = new AMTone(SignalLevel.InVrms(1),
                 Frequency.InkHz(1),      // 1 kHz
                 Waveshape.Sinusoiodal,   // sine carrier
                 50,                      // 50% modulation
                 Frequency.InHz(50),      // 50 Hz modulating
                 Waveshape.Triangular);   // triangle shape

// Start the AM signal with 90 degrees staring carrier phase
_5620.AMGenerator.Generate(AM,
```

```
                            Phase.InDegrees(90),   // carrier phase
                            Phase.InDegrees(0));   // modulation phase

Thread.Sleep(1000);

 // stop the AM signal
 _5620.AMGenerator.StopGenerator();
```

16.1.5. Pattern Generator

Many signaling methods often require basic signals to be patterned with very specific timing and cadence. Some examples within telephony are call-progress tones (such as dial tone, ring-back, and busy signals). The PatternGenerator manages the generation of signal patterns by reserving and commanding lower-level signal generators exposed by the device object in order go generate particular patterns.

When applications need to generate a particular signal pattern they will generally:

1. Specify a multi-tone compatible signal descriptor (see section 14.6)

2. Specify a cadence (see section 14.4)

The PatternGenerator class exposes the following members

- **Generate** – these methods start the pattern generator with the specified signal and pattern.

- **StopGenerator** – this immediately stops the generation of any active pattern and stops all signal generators used in the pattern

- **StopPattern** – this immediately stops the generation of any active signal pattern and can optionally leave the signal generators active

- **IsActive** – returns true while a signal pattern is being generated

Example:
```
' North American dial tone signal
Dim Dial = New DualToneSignal(Frequency.InHz(350), _
                              Frequency.InHz(440), _
                              SignalLevel.IndBV(-10))
' Stutter Dial Tone pattern
Dim Stutter = New RepeatingCadence("Stutter Dial", _
                                   10, True, _
                      TimeInterval.InMilliseconds(100), _
                      TimeInterval.InMilliseconds(100))
' Start the "dial" signal with the "stutter" pattern
_5620.PatternGenerator.Generate(Dial, Stutter)

' Wait up to 10 seconds for the pattern generator to finish
_5620.Wait.Until(ActionType.Pattern_Generator_Finished, _
                              TimeInterval.InSeconds(10))
```

16.1.6. Noise Generator

Many applications require the simulation of noise of various different types. The
NoiseGenerator class specifically manages the generation of white noise on supported
instruments. The NoiseGenerator class exposes a very simple set of members:

- **Generate** – this will activate the noise generator (immediately or at a particular
  time) with a specific noise level. The bandwidth of the white noise may vary
  from device to device. See the device specific documentation for details.

- **StopGenerator** – this method will stop the noise generator immediately, or at a
  particular time specified.

- **Level** – this property specifies the current noise level setting within the
  generator. This value can be adjusted while the noise generator is active.

Examples:

```csharp
//**********************************************
// Simple Example without scheduling

// Start the white noise generator
_7280.NoiseGenerator.Generate(SignalLevel.InVrms(0.2));

Thread.Sleep(1000); // wait a bit

// Stop the white noise
_7280.NoiseGenerator.StopGenerator();

///*********************************************
// Example generating FSK with synchronous noise
FSKCallerIDData Msg;
FSKCallerIDTransmission TX;

// Create Caller ID message
Msg = TIA.CallSetupMessage(CallerIDDateTime.Now(),
                           "John Smith", "5551234");
// Create an FSK transmission containing this message
TX = new FSKCallerIDTransmission(Msg, 0, 100, 1, 3,
                FSKPhysicalSettings.GetDefaults());

// Calculate a time T in the future to send FSK
TimeStamp T = _7280.Time.MostRecent() + TimeInterval.InSeconds(1);

// Schedule the FSK to start at time T
_7280.FSKGenerator.Generate(TX, T);

// Schedule the noise to start a bit before FSK
// (remember the FSK hasn't happened yet!)
_7280.NoiseGenerator.Generate(SignalLevel.IndBV(-35),
                T - TimeInterval.InMilliseconds(15));

// Schedule the noise to stop a bit after FSK
_7280.NoiseGenerator.StopGenerator(
            T + TX.Duration + TimeInterval.InMilliseconds(25));

// NOTE: By this point these signals are scheduled but
// probably haven't started yet!
// We should wait for the sequence to finish!
_7280.Wait.Until(ActionType.Noise_Generator_Stopped,
                           TimeInterval.InSeconds(4));
```

16.1.7. Echo Generator

Echo is a common impairment found in many networks which can usually be attributed to impedance mismatches at some place in the transmission medium. The EchoGenerator class manages the simulation of these echoes within supported instruments. Such echo generators can be modeled as a multi-tap delay and gain network as illustrated below in Figure 25.
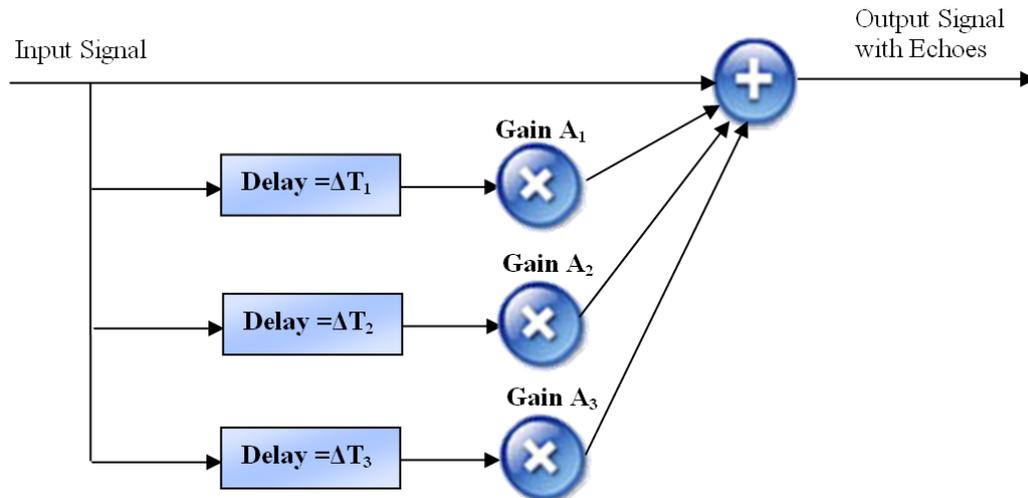
Figure 25Echo Generator Signal Diagram

Each tap in the echo generator consists of a delay ΔT_n and a gain A_n and are represented using the Echo class which has the corresponding two members. The input signal is delayed and scaled by each of the delay/gain networks. The outputs of the tap are then summed with the input signal to product the combined output signal. Each delay and gain can be adjusted independently to simulate a range of echo conditions.

- Typically each delay can be adjusted from 0 ms up to a maximum delay imposed by the instrument (typically 25 ms however please refer to the particular instrument's documentation)

- Each gain can be adjusted to

    o Attenuate by specifying absolute value less than one

    o Amplify by specifying absolute values greater than one

    o Invert by specifying negative gain values

The EchoGenerator class exposes the following members:

- **TapCount** – this returns the total number of echo taps which are available in the echo generator. Note: this may vary depending on the capability and version of the associated instrument.

- **GetEcho** – returns the echo definition at a particular tap in the echo generator

- **SetEcho** – sets the echo definition at a particular tap in the echo generator

- **Update** – loads the echo generator with an array of echo objects

- **Generate** – activates the echo generator

- **StopGenerator** – de-activates the echo generator

Example:

```
' 5 millisecond echo with -10 dB attenuation
Dim Echo1 = New Echo(TimeInterval.InMilliseconds(5), _
                     UnitlessQuantity.IndB(-10))

' 12.3 millisecond echo which inverts and halves the signal level
Dim Echo2 = New Echo(TimeInterval.InMilliseconds(12.3), _
                     UnitlessQuantity.InAbsolute(-0.5))

' assign each of the echo taps
_7280.EchoGenerator.SetEcho(0, Echo1)
_7280.EchoGenerator.SetEcho(1, Echo2)
_7280.EchoGenerator.SetEcho(3, Nothing) ' disable echo 3

' start the echo generator
_7280.EchoGenerator.Generate()

' TODO: generate signals here!

' stop the echo generator
_7280.EchoGenerator.StopGenerator()
```

16.2. Signal Detectors

16.2.1. Line State Detector

Most instruments with a telephone interface are capable of detecting telephony signals based on changes in telephone line state (see section 14.13). These signals include:

- **Line Flash** (see section 14.15)

- **Open Switching Interval** (see section 14.16)

- **Pulse Dialing** (see section 14.17)

As shown in Figure 26, the LineStateDetector class monitors notifications sent by the device object, detects patterns which correspond to valid telephony signals, and constructs and delivers matching notifications to the device object and any listening application.
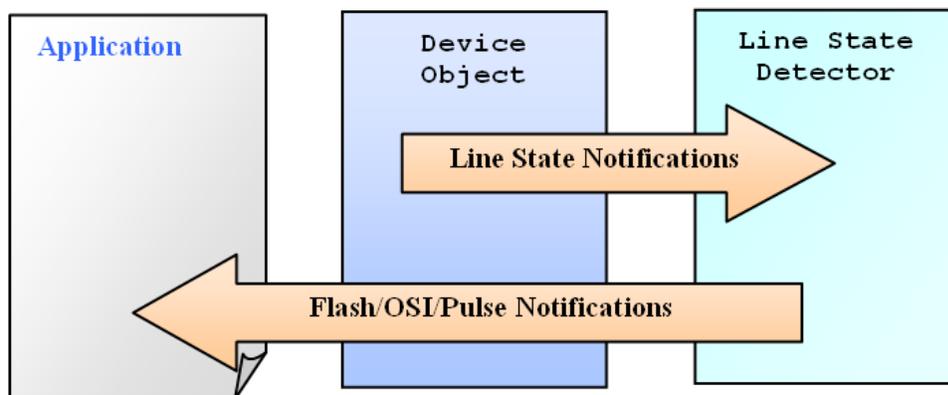
Figure 26 Line State Detector

Detection of these signals is managed by the LineStateDetector class which is usually accessible through the LineState property of the device object. This class exposes the following members:

- **DetectLineFlash** – enables detection of line flash signals

- **FlashDurationMinimum /FlashDurationMaximum** – these properties specify the range of valid flash durations which will be detected.

- **DetectOSI** – enables detection of OSI signals

- **OSIDurationMinimum / OSIDurationMaximum** – these properties specify the range of valid OSI durations which will be detected

- **DetectPulseDialing** – enables detection of pulse dialing digits

- **PulseBreakMinimum / PulseBreakMaximum** – specifies the range of "break" times will be detected as pulse dialing (see section 14.17).

- **PulseMakeMinimum / PulseMakeMaximum** – specifies the range of "make" times which will be detected as pulse dialing (see section 14.17).

> Instrument specific device classes often expose a sub-class of the LineStateDetector class which will allow applications to configure voltage and current thresholds used to determine the telephone line state.

16.2.2. DTMF Detector

Many of the instruments supported by aiDevices are capable of detecting DTMF signals as documented in section 14.9. This detector can usually be configured by means of a DTMFDetector object which can be accessed using a device object property with the same name. A simplified illustration of a DTMF detector is show in Figure 27.
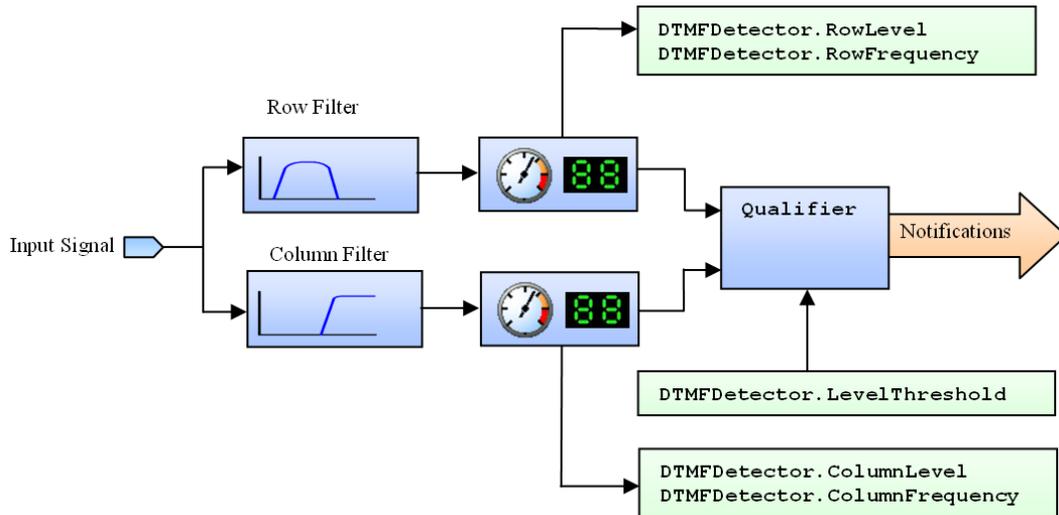
Figure 27DTMF Detector Structure

The DTMF detector can be configured using the following properties:

- **LevelThreshold** – specifies the minimum signal level required for both the row and column tones for DTMF to be properly detected.

The DTMF detector also reports several measurements using the following properties:

- **RowLevel / RowFrequency** – returns RMS level and frequency measurements taken after the DTMF row filter.

- **ColumnLevel / ColumnFrequency** – returns RMS level and frequency measurements taken after the DTMF column filter.

16.2.3. FSK Detector

Many of the instruments supported by aiDevices are capable of detecting and decoding FSK signals as documented in section14.18. This detector can usually be configured by

means of an FSKDetector object which can be accessed using a device object property with the same name.

The FSK detector can be configured using the following properties:

- **LevelThreshold** – specifies the minimum signal level required for both the mark and space signals for FSK to be properly detected.

> In future version this class may expose more advanced filtering options and detector configurations

16.3. Wait Manager

Events of importance are generally reported to applications from device objects by means of notification objects (see section 10) which provide a simple yet flexible structure for event driven and simulation applications. However this system alone can be cumbersome when simply waiting for particular events to occur in top-down style programs. In order to simplify these conditional wait conditions, device objects expose a WaitManager object (usually named Wait) which enables applications to wait for a particular notification before continuing execution. These objects:

- Simplify application structure by hiding the notification details

- Make code simpler to read and maintain

- Are completely thread safe and can be used without risk of race conditions

Most members of the WaitManager class are named "Until" and follow the same general pattern:

- Generally the first parameter of Until specifies the condition to wait for

- The last argument is named OrTimeout and specifies the maximum amount of time to wait for the specified condition to occur.

At present there are four overloads of the "Until" method which will:

- Wait for an ActionNotification with the specified ActionType and returns the time at which the action occurred (or null if it did not). This is very useful when waiting for signal generators to finish before continuing execution.

- Wait for a change in telephone line state to a specified value and returns the DetectedLineStateChange object.

- Wait for the detection of a specific signal type and returns the corresponding ISignal object

- Wait for a condition specified by the application by means of a delegate

Applications can still leverage the simple Wait.Until( ) programming structure to wait for custom conditions by specifying a delegate function of type **NotificationFilterDelegate**. This delegate function will:

- Accept a single Notification argument which should be tested to determine if it indicates the desired condition

- Return 'true' if the Notification argument indicates the condition which is being waited for.

This delegate can then be specified as an argument to one of the "Until" methods after which it will be called to test incoming notifications to determine if they specify the desired condition.

> Functions used as NotificationFilterDelegate arguments must remain accessible and valid during the entire duration of the wait manager Until function and **will be accessed on a thread other than the calling thread!** Applications developers are urged to
> - Declare such delegates as static functions
> - Ensure that only local variables are accessed within the scope of the function

Examples:

```csharp
DetectedLineStateChange LS;
Notification N;
TimeStamp T;
TimeInterval Timeout = TimeInterval.InSeconds(10);

//----------------------------------------------
// Wait for a change in line state to "In-Use"
//----------------------------------------------
LS = _5620.Wait.Until(Telephone_Line_State.In_Use,Timeout);
if (LS != null)
    Debug.Print("In Use detected " + LS.ToString());
else
    Debug.Print("Timeout!");

//----------------------------------------------
// Go off hook 1 second and wait for it to happen
//----------------------------------------------
_5620.TelInt.GoOffHook(_5620.Time.Now() +
                        TimeInterval.InSeconds(1));
T = _5620.Wait.Until(ActionType.Telephone_Interface_Went_OffHook,
                                    Timeout);
if (T != null)
    Debug.Print("Detected off hook at " + T.ToString());
else
    Debug.Print("Timeout!");

//--------------------------------------------------------
// Wait for DTMF '3' as specified by a custom delegate
//--------------------------------------------------------
N = _5620.Wait.Until(WaitForDTMF3, Timeout);
if (N != null)
    Debug.Print("Detected " + N.ToString());
else
    Debug.Print("Timeout!");


// Delegate function used to detect DTMF '3'
static bool WaitForDTMF3(Notification N)
{
    // Check if this is a signal notification
    SignalNotification S = N as SignalNotification;
```

```
        if (S == null) return false;

        // Check if this is DTMF
        DTMF D = S.Signal as DTMF;
        if (D == null) return false;

         // Check if this is the digit 3
         return (D.Key == '3');
}
```

16.4. Detected Signal List

All signals detected by an instrument are reported to applications by means of notification objects (see section 10). However this system alone can be cumbersome for applications which:

- Are written in a top-down structure, or

- Only require information about signals detected by an instrument, and

- Do not have stringent timing requirements or do not need to take action based on the detected signals

- Require signals to be reported in temporal order

The DetectedSignalList object provides a simple and flexible solution for just such applications (illustrated in Figure 28). Typically this list is accessible through the DetectedSignals property of the implementing device object and will:

- Automatically receive notifications from the device object and deal with all the corresponding threading issues

- Records all detected signals reported in SignalDetectedNotifications objects (see section 10.1.3) into a list. This list:

  - Is automatically sorted into ascending order based on the signal start time.

  - Can grow up to a maximum capacity (reported by the Capacity property) at which incoming signals are ignored.

- Allows the calling applications to access the contents of the list in a thread safe manner.
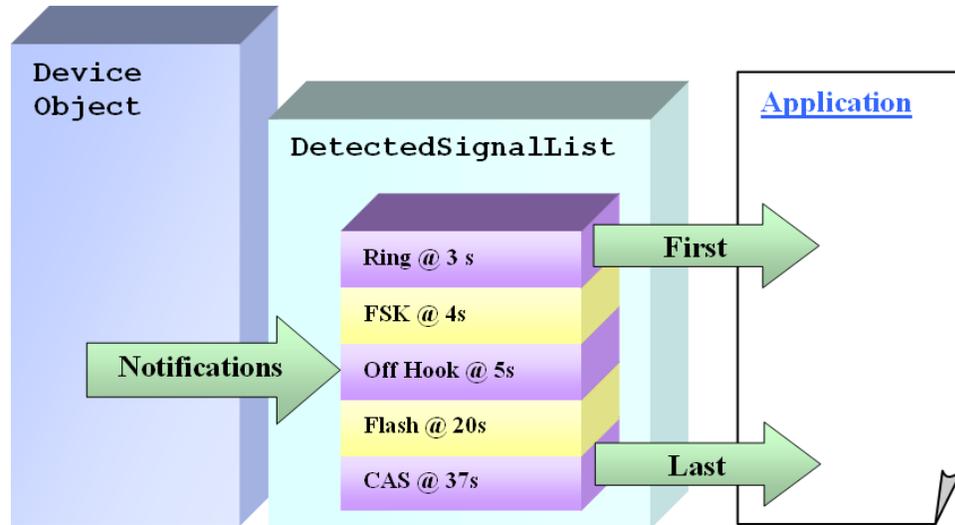
Figure 28Detected Signal List

The DetectedSignalList class exposes the following members:

- **Count** – returns the number of detected signals recorded

- **Capacity** – returns the maximum number of signals which can be recorded by the list (after which the incoming signals are ignored)

- **Clear** – flushes all the signals from the list

- **Enable** – when true incoming signals will be recorded, when false incoming signals are ignored.

- **First** – returns the "oldest" signal in the list

- **Item** – returns the signal at the specified index in the list. An index of zero corresponds to the "oldest" signal.

- **Last** – returns the "newest" signal in the list

- **ExtractFirst** – removes and returns the "oldest" signal in the list.

- **Extract** – removes and returns a signal at the specified index in the list or the entire contents of the list.

The DetectedSignalList class is completely thread-safe and applications are free to access members without regard to threading conflicts. However developers should consider setting the enable property to false before indexing the contents of the DetectedSignalList through the Item methods to ensure that incoming signals don't accidently change the list contents during the traversal.

16.5. Time Manager

Each instrument is responsible for managing its own time-base following the system documented in section 9. Each device object then exposes a TimeManager object which gives applications access to the instrument time information. The TimeManager has the following public members:

- **Epoch** – this returns the time stamp corresponding to the time when the communications were established with the instrument. In general all timing from the device is referenced from this time.

- **MostRecent** – this returns the most recently polled timestamp returned from the device. This does not incur timing delays and is only guaranteed to be within half a second of the current instrument time.

- **Now** – this queries the connected instrument and returns a time stamp containing the current device time. **Note: this time value will be subject to slight inaccuracies due to communications and processing delays.**

16.6. Telephone Interfaces

A complete analog telephony circuit generally consists of

- One Central Office (CO) telephone interface (also called FXS) usually provided by a telephone service provider. This interface provides the DC voltage and loop current required for all other devices attached to the circuit.

- A telephone conductor containing at least two conductors (called tip and ring)

- One or more Terminal Equipment (TE) telephone interfaces (also called FXO or CPE) connected in parallel across the tip-ring conductors. TEs are usually phones, fax machines, modems, etc.
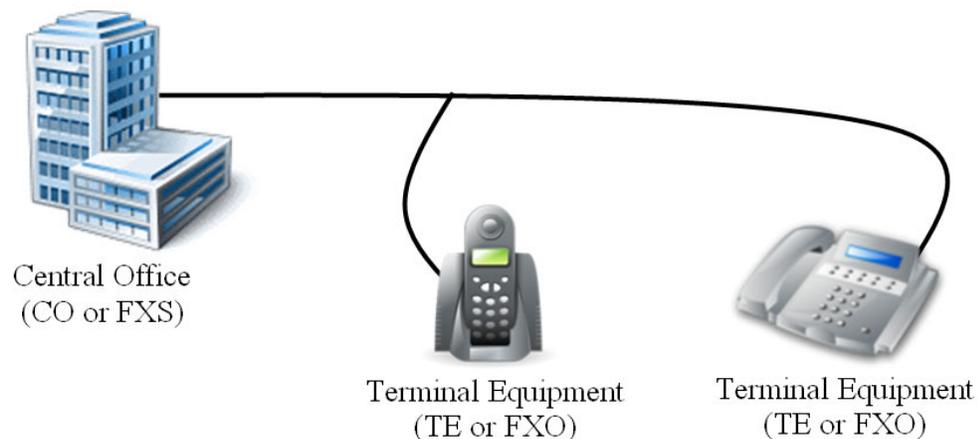

Figure 29 Simplified Telephone Interface Arrangements

Many Advent Instruments products contain one of these telephone interface circuits, and while the details of each of these interfaces vary depending on implementation, each telephone interface has certain fundamental abilities based solely on its role in the telephone circuit. The aiDevices framework defines three interfaces which are implemented by support classes that manage a telephone interface circuit.

The **ITelephoneInterface** interface is implemented by all support classes which manage a telephone interface circuit regardless of its arrangement in the circuit. This interface defines:

- **ACImpedance** – specifies the AC impedance characteristics which will be presented by the telephone interface to the telephone line.

- **Balance** – specifies the AC impedance characteristic of the telephone network as seen by the instrument. This is used in the instrument's signal hybrid and affects the trans-hybrid loss.

- **IsOffHook** – returns true if the telephone line is in the off hook state from the telephone interface's perspective. From the central office's perspective this means that a connected device is drawing significant loop current. From the terminal equipment's perspective this means that the hook-switch is in the off-hook state (receiver is lifted).

- **LineState** – returns the telephone line state (see section 13.4) from the perspective of the telephone interface. This state information can report a disconnected telephone line or a telephone line which is "in use".

The **IFXSTelephoneInterface** interface is implemented by classes which manage central office (or FXS) circuitry. This interface inherits from ITelephoneInterface and defines the very basic properties and behaviors which are required by its role in the telephone circuit:

- **SourceVoltage** – this specifies the on-hook DC feed voltage which will be presented on the telephone line.

- **SourceCurrent** – this specifies the regulated off hook loop current which will be available when a TE goes off hook. Note: some telephone interfaces also support a voltage source mode

- **Disconnect** – this method will disconnect the telephone interface circuitry from the telephone line.

- **Connect** – this method will connect the telephone interface circuitry to the telephone line.

- **IsDisconnected** – returns true if the telephone interface circuitry is currently disconnected from the telephone line.

- **Reverse** – this method will reverse the polarity of the telephone interface (and thus the voltage of the DC feed voltage)

The **IFXOTelephoneInterface** interface is implemented by all classes which manage Terminal Equipment (TE) telephone interface circuitry. This interface inherits from ITelephoneInterface and defines the very basic properties and behaviors which are required by its role in the telephone circuit:

- **GoOffHook** –this method will place the hook switch into the off hook state (the analogy would be lifting the receiver on a phone)

- **GoOnHook** – this method will place the hook switch into the on-hook state (the analogy would be hanging up a phone).

Please see the instrument specific documentation details for each particular telephone interface class.

16.7. Recording and Downloading

Many Advent Instruments products are capable of making recordings of AC and DC information and downloading it to the host computer. Each class which is responsible for such recordings inherit from the RecordingManager class which exposes the following interface:

- **SupportedSampleRates** –returns an array of Frequency objects which describe the sample rates which may be used for recordings within the device.

- **SupportedFormats** – returns an array of descriptors which describe the resolution, scale, and description of the samples which may be recorded by the device.

- **RecordingDepth** – returns the maximum possible samples which can be recorded by the instrument.

- **StartRecording** – this will start a finite duration recording with the maximum possible duration and default settings. Each particular recording class may implement several overloads which may offer more options.

- **StartRecordingIndefinitely** – this will start a recording which will continue indefinitely (overwriting the oldest samples) until stopped. Each particular recording class may implement several overloads which may offer more options.

- **StopRecording** – this will immediately stop any active recordings within the instrument.

- **IsRecording** – returns true if samples are being recorded within the instrument

- **IsRecordingIndefinitely** – returns true if samples are being recording indefinitely until stopped.

- **SampleRate** – sample rate currently being used for recordings.

- **RecordedSamples** – returns the number of samples which have been recorded

- **RecordedTime** – returns the duration of the recording in seconds.

- **StartDownload** – this starts the download of samples from the device to the host computer. The download process is generally performed in the background on a worker thread to a SampleWriter object which determines the destination for the samples. An asynchronous callback method may be specified which can be used to notify the application when the download completes.

- **IsDownloading** – returns true if a download is in progress.

Each particular instrument may have different requirements or restrictions on recording and downloading. Please see the instrument specific documentation in later sections for details.

The destination for downloaded samples must be specified using SampleWriter objects which handle all the details required to convert the stream of samples into the required destination format and all threading issues. Each class which is capable of receiving a stream of downloaded samples is derived from the SampleWriter class which defines the following members:

- **IsFinished** – this returns true if the download of samples is complete
- **Close** – this method closes the destination for samples (such as a file) and terminates the download of samples into the destination.

The following sub-sections discuss particular SampleWriter classes which are available within the aiDevices framework

Future releases will contain SampleWriter classes which will support additional download formats. If you require a particular download format please contact technical support.

16.7.1. Downloading to .WAV Files

The aiDevices framework defines the WaveSampleWriter class which manages the details of writing download samples to a .wav file in PCM format. When multiple sources of data are downloaded (i.e. voltage and current), this class will write the concurrent samples in to separate channels (i.e. left and right). Each sample will be scaled into a 16 bit integer and scaled such that the 16 bit signed integer matches the full range of the corresponding sample format.

Example:

```
With _7280.ACRecording

      ' records 1000 samples
      .StartRecording(.SupportedSampleRates(0), 1000)

      ' wait a while for recording to complete
      While .IsRecording
            Thread.Sleep(100)
      End While

      ' initiates a download of the samples to a wave file
      ' the method DownloadDone will be called when complete
      .StartDownload(New WaveSampleWriter("C:\test.wav"), _
                     AddressOf DownloadDone)
End With

'This method is called asycrhonously when downloading is
'complete
Private Sub DownloadDone(ByVal Writer As SampleWriter)
      ' TODO: possibly process the samples?
End Sub
```

17. AI-5620 TE Simulator

One of the instruments supported by aiDevices is the AI-5620 Terminal Equipment Simulator which

- Simulates Terminal Equipment (TE) with programmable telephone interface characteristics

- Detects, analyzes, generates, and records telephony signals (such as DTMF, FSK, Caller ID, Metering Pulses, etc)

- Tests the functionality and compliance of central office (FXS) equipment

For a more detailed product information and specifications please refer to the "AI-5620 User Guide" which is available at www.adventinstruments.com.

The features of the AI-5620 instrument are accessed through the AI5620_TE_Simulator class within the aiDevices framework. Unless otherwise noted, all documentation within the following subsections refers specifically to the AI5620_TE_Simulator class.

17.1. Establishing Communications

Communications with the AI-5620 instrument can be established using one of the three static Connect methods as documented in section 8.4. Each of these functions will behave as follows:

- If an instrument is found which is supported by the class and communications are established successfully, then an instance of the device object will be created and returned. This object can then be used to control the connected instrument.

- If no supported instruments are found then null is returned.

- If an instrument is found but communications are not established correctly or the instrument is not supported by the device class then the function **will raise an Exception** which must be handled by the calling application.

When communications are established with an AI-5620 instrument through any of the Connect methods available:
- The device object **does not modify instrument settings** but rather **synchronizes** with the current state of the AI-5620 which may be left in a particular state by another application. This behavior is vital in situations when connections must be established and terminated with the AI-5620 without disturbing the telephone line state, signal routing, or signal generators.
- If your application requires a particular instrument configuration it must call ResetToDefaults after communications are established or adjust each device setting to the desired state.
- Certain features (such as recording) may not be able to be completely synchronized when communications are established and **may** reset such features to default settings.

Examples:

```csharp
AI5620_TE_Simulator Dev = null;
try {
    // connect to any available AI-5620
    Dev = AI5620_TE_Simulator.Connect();
    if (Dev==null)
    {
        // No AI-5620 instruments available!
    } else {
        if (Dev.Exceptions.Count !=0)
        {
            // AI-5620 is connected but has
            // reported an problem
        }
    }
}catch(Exception ex)
{
    // AI-5620 may be present but could not connect!
}
```

17.2. Terminating Communications

Once an application has finished using an AI5620_TE_Simulator object it must always call one of the Close methods before setting the object variable to null! Please refer to section 8.6 for detailed information regarding these Close methods.

When communications are terminated with an AI-5620 instrument through a Close method:
- All automated behavior (such as pattern generation, scheduling, etc) will stop immediately.
- All static device settings (such as line impedances, hook switch state, signal routing) and simple signal generators (tone generators, echo, etc) will remain in their current states. This behavior may be desirable if the instrument is configured within a test fixture.
- If your application requires a particular instrument state after communications are terminated then it should call ResetToDefaults (or ensure that each instrument setting is property configured) **before** calling Close

17.3. Resetting to Default Settings

In many applications it is desirable to reset the instrument settings to their defaults to return the device to a known operating condition. The AI5620_TE_Simulator class implements a ResetToDefaultSettings method as documented in section 8.7. In general this will:

- Stop signal generators and reset all signal generator settings to nominal defaults

- Reset all detector settings to nominal defaults

- Reset all telephone interface settings (see section 17.6.1).

- Reset all digital outputs to "Output Low" and disable all special functions

- Reset all signal routing, measurement settings, and filters to defaults

- Reset protection mechanisms within the instrument.

This ResetToDefaultSettings **does not initiate a hardware reset** of the associated instrument but rather reconfigures the instruments with "safe" default values.

17.4. Determining Instrument Capabilities

The particular capabilities of the instrument's hardware and firmware in combination with the supporting device classes are reported through the Capabilities property of the AI5620_TE_Simualtor class. The AI-5620 specific capabilities are reported using the properties documented in the following sections.

> The capabilities of the tone generator, FSK generator, noise generator, echo generator, and metering pulse detector are reported using the standard interfaces documented in section 8.3.

17.4.1. Telephone Interface Capabilities

The capabilities of the telephone interface are reported through the following properties

- **ACResistanceMinimum**
  ACResistanceMaximum – reports the range of acceptable range of resistance values which can be used to program the telephone interface ACImpedance property.

- **OffHookDCResistanceMinimum**
  OffHookDCResistanceMaximum – reports the acceptable range of resistance values which can be assigned to the telephone interface OffHookDCResistance property.

- **OnHookDCResistanceMaximum** – reports the maximum possible on hook DC resistance.

- **OnHookProgrammableDCResistanceMaximum**
  OnHookProgrammableDCResistanceMaximum – reports the supported range of programmable on hook DC resistances which can be assigned to the telephone interface OnHookDCResistance property.

- **AvailableRingingLoads** – returns an array containing the supported ringing load values which can be assigned to the telephone interface RingerLoad property.

17.4.2. Metering Pulse Detector Capabilities

The capabilities of the metering pulse detector are reported through the following properties

- **MeteringPulseFrequencyMaximum**
  MeteringPulseFrequencyMinimum – reports the range of detectable metering pulse frequencies.

- **MeteringPulseFrequencyToleranceMaximum** – reports the maximum frequency tolerance which can be specified for metering pulse detection

- **MeteringPulseTemplatesMaximum** – reports the maximum number of metering pulse templates which can be simultaneously detected.

17.5. Signal Routing and Processing

The AI-5620 instrument can be configured to route a selection of internal signals throughout the instrument and the connectors on the front and rear panels of the instrument in order to meet the requirements of most applications.

The signal routing capabilities of the AI-5620 can be accessed using a SignalManager object which is accessed through the Signals property of the AI5620_TE_Simulator class. The following sections describe the signal routing capabilities within the instrument and highlight the corresponding programming interface; starting with the signal generators shown in Figure 30 below.
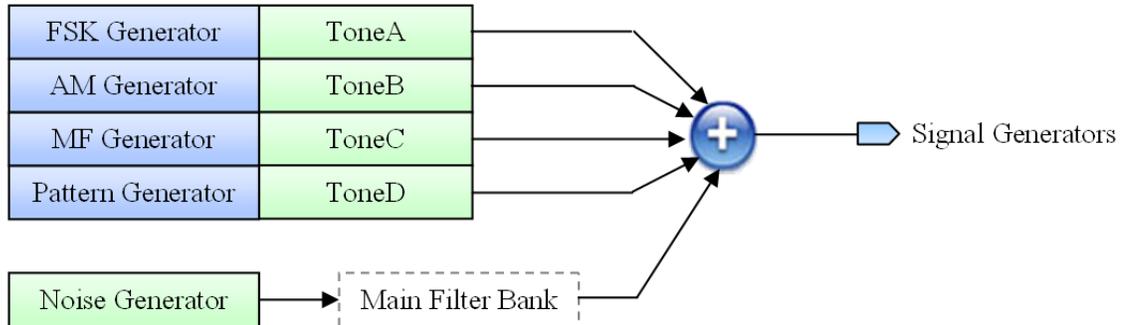
Figure 30 AI-5620 Signal Generator Routing Diagram

The outputs of each of the fundamental signal generators are summed together to produce the internal signal labeled "Signal Generators" as illustrated in Figure 30. This signal can then be routed to many signal blocks through the instrument including the telephone interface. A heavily simplified diagram of this telephone interface signal routing is shown in Figure 31 below.
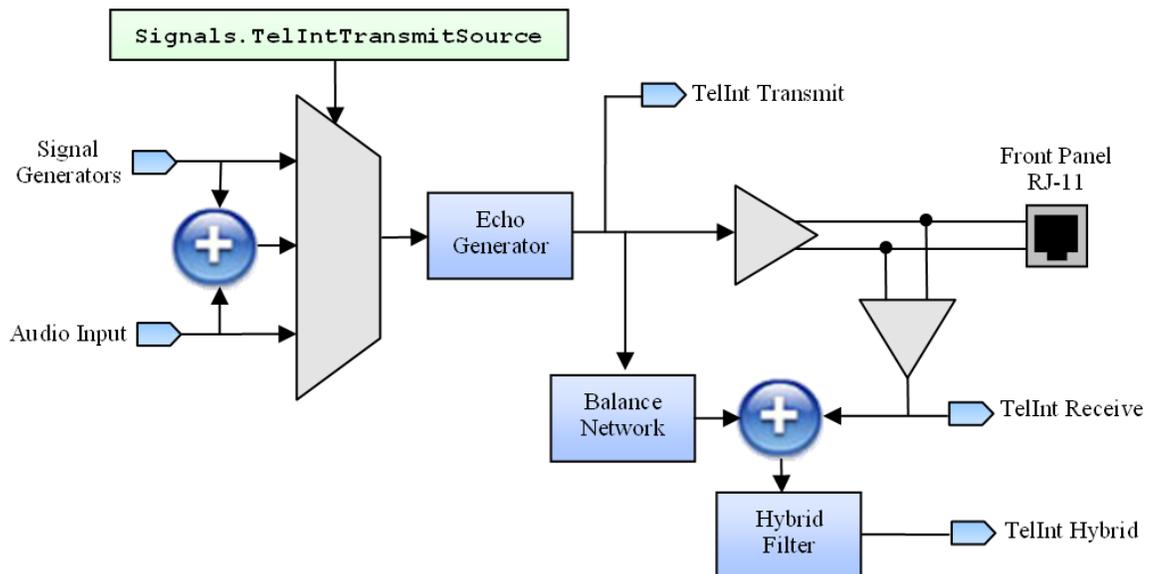
Figure 31Simplified AI-5620 Telephone Interface Routing Diagram

Three different AC signals can be selected to be transmitted on the telephone line using the Signals.TelIntTransmitSource property:

- Signal generator outputs (Figure 30)

- Audio input (Figure 32)

- The summation of both these signals

- No signal (silence)

This signal is then routed to the echo generator (see section 0) whose output is "TelInt Transmit". This signal is then transmitted onto the tip-ring interface if the AC termination is applied (see section 18.6).

The AC voltage signal measured from the telephone interface is yet another internal signal which is labeled "TelInt Receive" and is typically routed to the meter and signal detectors. The AI-5620 also contains a signal hybrid (which cancels transmitted signals from the received signals) which results in another signal labeled "TelInt Hybrid". Applications will tend to use this signal when attempting to isolate signals from other devices on the telephone line from those sent by the AI-5620.

The AI-5620 is also equipped with two BNC connectors on the rear panel which can be used as a signal source or sink for AC signals within the instrument. The signal routing for these connectors is illustrated in Figure 32.
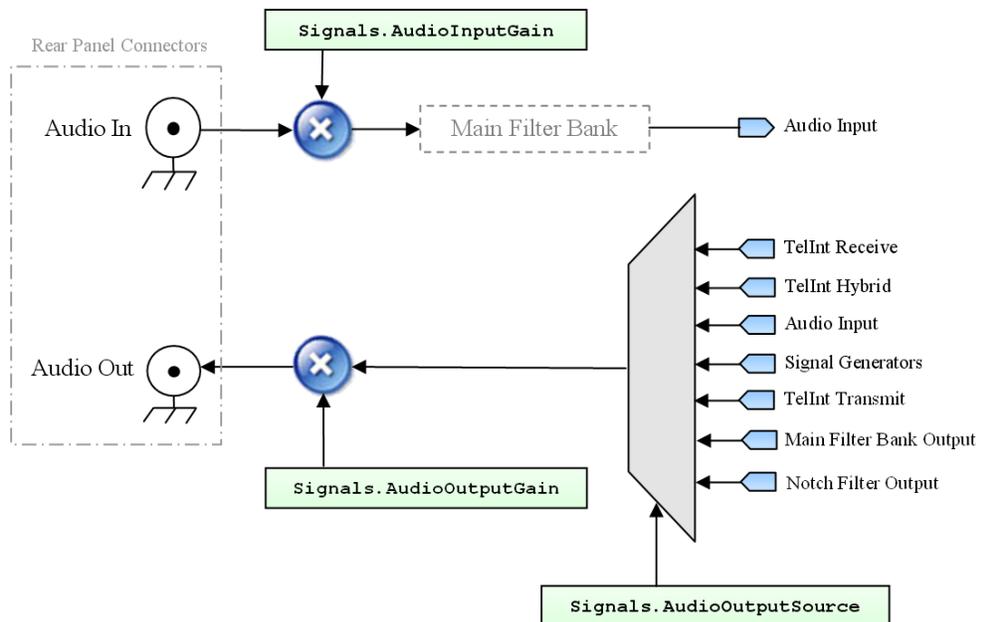
Figure 32 AI-5620 Audio Input/Output Routing Diagram

Audio signals from within the instrument can be routed to the "Audio Out" connector using the following Signals properties:

- **AudioOutputSource** – this selects the signal which is routed to the audio output connector

- **AudioOutputGain** – this selects the gain which is applied to the signal before it is applied the audio output. This can be used to correct for losses or inversions in external equipment or to mute the output (by setting zero gain).

The signals routed to the meter and internal signal detectors can be selected using the following Signal properties.

- **MeterSource** – this selects the internal signal which is routed to the meter for level and frequency analysis as illustrated in Figure 33. Please see section 17.7 for more information on the meter properties and signal configurations.

- **DetectorSource** – this property selects the signal which is routed to each of the AC signal detectors within the instrument (see section 17.10).

The AI-5620 is also equipped with a programmable filter bank which can be inserted into one of several logical signal processing positions within the instrument. This "Main Filter Bank" is configured using the following Signals properties:

- **MainFilterBank** – this property can be used to configure the main filter bank with a particular filter definition (see section 13.3).

- **MainFilterPosition** – this property selects the position of the main filter bank within the signal processing chain. The available settings are:

  - **Before Meter** – this places the main filter bank before the meter measurements (see section 17.7)

  - **Filtering Audio Input** – this settings configures the main filter bank to filter the signals applied to the audio input connector (as shown in Figure 32)

  - **Filtering Noise Generator** – this setting configures the main filter bank to filter the noise generator output before it is summed with the rest of the signal generator outputs (as shown in Figure 30). This enables applications to "shape" or band-limit the noise generator output.

17.5.1. Reset to Defaults

The SignalManager object exposes a ResetToDefaults method which will reset only the signal routing settings back to their default values. More specifically this method will:

- Set the audio output gain to zero and set the source to "none"

- Set the audio input gain to zero

- Clear the main filter bank (no filtering) and places the filter before the meter

- Both the meter source and the detector source are configured to receive the "TelInt Receive" signal (see Figure 31)

- The signal generators are selected as the telephone interface transmit source.

17.6. Telephone Interface

The telephone interface circuitry (FXO) of the AI-5620 is accessible through a single RJ-11 connector on the front panel of the instrument. This interface can be connected to a central office or other FXS circuit which will provide the DC feed voltage and loop current required for the telephony circuit. The AI5620_TE_Simulator class allows applications to control the features of this telephone interface circuitry through the TelInt property of the device object. The TelephoneInterface class exposes the following members:

- **ACImpedance** – specifies the AC impedance presented on the telephone interface when the AC termination is applied on the telephone (generally when off hook). This impedance may be specified using:

    o An Impedance object specified in section 13.2.

    o One of the values returned by the FixedImpedancesAvailable list

- **ACTerminationBandwidth** – specifies the AC bandwidth available when the AC termination is applied on the telephone line. When the wideband setting is specified, the low frequency corner is reduced which allows for more accurate low frequency transmissions.

- **ACTerminationMode** – specifies when the AC termination will be applied to the telephone line (which is required to transmit AC signals). The available settings are:

    o **Disconnected** – the AC termination will never be applied to the telephone line

    o **Connected_OfffHook (default)** – the AC termination will be applied only when off hook. This setting most accurately reflects the operation of a normal TE.

    o **Connected_OnHook** – the AC termination will be applied to the telephone only when on hook.

    o **Connected** – the AC termination is always applied to the telephone line

- **Balance** –specifies the AC impedance of the telephone network as seen by the instrument. This is used in the instrument's signal hybrid and affects the trans-hybrid loss.

- **Connect** – this method will cause the telephone interface to be internally connected; either immediately or at a specified time.

- **Disconnect** – this method will cause the telephone interface to be internally disconnected; either immediately or at a specified time.

- **FastSettle** – when set to true the time required for DC voltages and currents to settle when going off-hook is significantly shortened. (Typically from about 20ms to about 2ms).

- **FixedImpedancesAvailable** – returns a list of the fixed impedances installed within the instrument. If optional complex impedances are installed they will also appear in this list.

- **FixedTermination** – selects the fixed termination (load) which can be applied to the telephone interface. The available settings are:

    o **None** – no fixed terminations are applied

    o **Resistive_600R** – a 600 Ω resistor is connected between tip and ring

    o **Short_Circuit** – the tip and ring conductors are shorted together.

- **Generate** – this method will cause a line flash to be generated; either immediately or at a specified time.

- **GoOffHook** – this method will cause the telephone interface hook switch to be taken off hook; either immediately or at a specified time.

- **GoOnHook** – this method will cause the telephone interface hook switch to be taken on hook; either immediately or at a specified time.

- **HighGainMuteTX** – when this property is set to true, the telephone interface transmitter will be muted and the measurement circuit gain will be increased which allows the instrument to measure significantly lower levels signals.

- **IsDisconnected** –returns true if the telephone interface circuitry is currently disconnected internally.

- **IsOffHook** – returns true if the telephone interface hook switch is off hook.

- **LineState** – returns the current detected line state (see section 13.4).

- **OffHookDCResistance** – specifies the DC load resistance presented by the instrument when off hook.

- **OnHookDCResistance** – specifies the DC load resistance presented by the instrument when on hook.

- **SetMaximumOnHookDCResistance** – this will configure the telephone interface with the maximum possible on hook dc resistance

- **RingerLoad** – specifies the ringing load (in REN) which will be applied to the telephone interface when on hook.

- **ResetToDefaults** – this method will reset the telephone interface to default settings (see section 17.6.1).

Example:
```vbnet
Imports Advent.aiDevices.AI5620_TE_Simulator

'Assumes: Dim  5620 As AI5620 TE Simulator

With  5620.TelInt
       ' Use TBR-21 AC impedance when off hook only!
       .ACImpedance = Impedance.TBR_21
      .ACTerminationBandwidth =TerminationBandwidth.Normal
      .ACTerminationMode =ACTerminationSetting.Connected OffHook

      ' make DC transients settle faster when going off hook
      .FastSettle = True

       ' set resistance when off hook
       .OffHookDCResistance = Resistance.InOhms(200)

       ' set maximum possible on-hook DC resistance!
       .OnHookDCResistance =
             _5620.Capabilities.OnHookDCResistanceMaximum

      ' assume source has 600 ohm impedance and configure
      ' the signal hybrid accordingly
      .Balance = Impedance.Resistive 600

      ' connect a 1 REN ringing load when on hook
      .RingerLoad = 1

      ' Take the telephone interface off hook
      .GoOffHook()
      Thread.Sleep(2000)  ' wait a while

      ' generate a 500 ms line flash
     .Generate(New LineFlash(TimeInterval.InMilliseconds(500)))
      Thread.Sleep(2000)  ' wait a while

      ' go back on hook
      .GoOnHook()
End With
```

17.6.1. Reset to Defaults

The TelephoneInterface object exposes a ResetToDefaults method which will reset only the telephone interface settings back to their default "safe" values. More specifically this method will:

- Go on hook with maximum possible on-hook DC resistance

- Re-connect the telephone interface to the front panel (if not already connected)

- Remove all ringing loads

- Change the AC termination mode such that the line is AC terminated when off-hook and with normal bandwidth

- Configure the AC impedance and balance setting to 600 Ω

- Remove all fixed terminations

- Un-mute the transmitter

- Disable fast setting

17.7. Meter and Measurements

The AI-5620 instrument contains a signal meter which can be configured to perform filtering, level measurement, and frequency measurements on many signal sources available within the device. These features are implemented by the MeterManager class and are accessible through the Meter property of the AI560_TE_Simulator class. The AC signal routing within the meter is illustrated below in Figure 33.
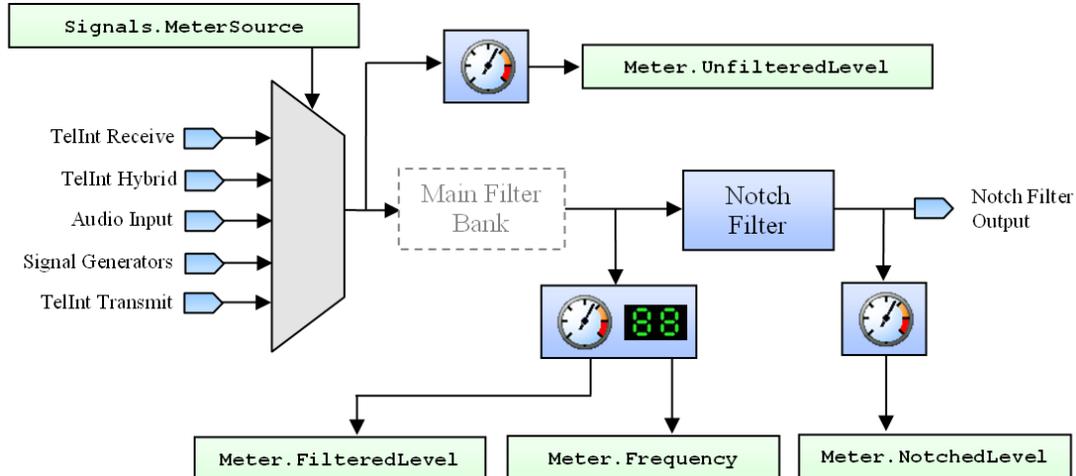

Figure 33 AI-5620 Meter Signal Routing Diagram

The AC signal routed to the meter is selected using the Signals.MeterSource property which is described in section 17.5.

The MeterManger class exposes the following members which manage AC signal measurements.

- **MeasurementFilter** – this configures the main filter bank and places the filter before the meter measurements (see section 17.5).

- **UnfilteredLevel** – returns the current RMS level measurement of the input signal before any filtering is applied.

- **FilteredLevel** – returns the current RMS level measurement after the main filter bank.

- **Frequency** – returns a frequency measurement taken after the main filter bank.

- **NotchFilterBank** – configures the notch filters within the meter with up to two independent notch filters.

- **NotchedLevel** – returns the current RMS level measurement taken after both the main filter bank and notch filter bank.

- **ACMeasurementSpeed** – this property configures the settling time and low frequency accuracy of each of the above level measurements. The allowed values are:

  - **Very Fast** $\Rightarrow$ 25 ms settling time ($\pm$ 0.1 dB accuracy $f \geq 500$ Hz)

  - **Medium** $\Rightarrow$ 100 ms settling time ($\pm$ 0.1 dB accuracy $f \geq 100$ Hz)

  - **Slow** $\Rightarrow$ 400 ms settling time ($\pm$ 0.1 dB accuracy $f \geq 30$ Hz)

  - **Very Slow** $\Rightarrow$ 1.1 second settling time ($\pm$ 0.1 dB accuracy $f \geq 10$ Hz)

> All AC signal level measurements are affected the ACMeasurementSpeed which configures the settling time and low frequency accuracy of the level meters.

The MeterManger class also exposes the following members which manage DC-coupled signal measurements from the telephone interface:

- **LineVoltage** – returns the DC line voltage measured in the telephone interface.

- **LoopCurrent** – returns the DC loop current measured within the telephone interface.

- **CommonModeVoltage** – returns the common mode voltage measured from the tip-ring conductors with respect to earth ground.

- **DCMeasurementSpeed** – this property specifies the averaging applied to the DC-coupled measurements above. This averaging will affect the accuracy of the measurements in the presence of low frequency AC signals. The allows values are:

  - **No Filtering** $\Rightarrow$ No averaging is applied

  - **Medium** $\Rightarrow$ 0.5 second settling time (≥ 40 dB rejection $f \geq 100$ Hz)

  - **Slow** $\Rightarrow$ 2 second settling time (≥ 40 dB rejection $f \geq 30$ Hz)

  - **Very Slow** $\Rightarrow$ 5.5 second settling time (≥ 40 dB rejection $f \geq 10$ Hz)

The measurement manager also contains the following general members

- **GetMeasurements** – this method returns a set of nearly simultaneous measurements from the meter.

17.8. Instrument Status

Once communications are established with an AI-5620, the instrument's firmware and the device class cooperate to periodically send important status information to the device's StatusManager object (in the background) where it is accessible to the parent application without incurring communications delays. This status information is accessible through the Status property of the device object.

> All properties of the StatusManager object (accessed through the Status property) can be read **without incurring communication delays or any associated processing**. This is very helpful for updating status displays without interfering with other tasks.

> All information reported through the Status property is **automatically updated in the background from a worker thread** and is thread-safe. The application must be careful to note:
> - The reported information may be updated at any time with respect to the execution of the application.
> - The status information is updated at a fixed rate and may take as long as 400 milliseconds between refreshes (although typically it is faster)

The accessible status values are:

- **LineState** – the most recently reported telephone line state (see section 13.4)

- **UnfilteredLevel** – the most recent unfiltered level measurement from the meter (see section 17.7)

- **FilteredLevel** – the most recent filtered level measurement from the meter (see section 17.7)

- **MeterFrequency** – the most recent frequency measurement from the meter (see section 17.7)

- **LineVoltage** – the most recent line voltage measurement from the meter (see section 17.7)

- **LoopCurrent** – the most recent loop current measurement from the meter (see section 17.7)

- **IsVoltageProtectionActive** – indicates if the over-voltage protection mechanism has been engaged within the AI-5620 instrument (see section 17.14)

- **IsPowerProtectionActive** – indicates if the over-power protection mechanism has been engaged within the AI-5620 instrument (see section 17.14)

- **IsTransmitterClipping** – returns true if the signal being generated on the telephone interface is clipping due to very high signal level.

- **IsReceiverClipping** – returns true if the measurement circuitry within the telephone interface is clipping due to very high signal levels.

17.9. Signal Generation

17.9.1. Tone Generators

Each AI-5620 is equipped with four tone generators whose features are accessible through the ToneA, ToneB, ToneC, and ToneD properties.

Each of these properties returns a ToneGenerator object (see section 16.1.1) which can be used to generate simple tones or can be reserved by higher level signal generators to produce more complicated signaling.

17.9.2. Pattern Generator

The AI5620_TE_Simualtor object supports the generation of multi-tone patterns through the PatternGenerator class which is returned by the property with the matching name (see section 17.9.2).

17.9.3. AM Generator

The AI-5620 is equipped with an AM signal generator which is accessed through the AMGenerator property (see section 16.1.4).

17.9.4. Echo Generator

The AI-5620 is equipped with an echo generator for simulating telephone network impairments which is accessible through the EchoGenerator property (see section 0).

17.9.5. FSK Generator

The AI-5620 is equipped with an FSK generator which can be accessed through the FSKGenerator property (see section 16.1.3).

17.9.6. MF Generator

The AI-5620 is equipped with an MF generator for generating dual tone signals which can be accessed through the MFGenerator property (see section 16.1.2).

17.9.7. Noise Generator

The AI-5620 is equipped with a white noise generator which can be accessed through the NoiseGenerator property (see section 16.1.6).

> The main filter bank can be configured to filter the output of the noise generator in order to band-limit the resultant noise signal (see section 17.5).

17.9.8. Pulse Dialing Generator

The AI-5620's pulse dialing capabilities are exposed through the PulseDialingGenerator class which can be accessed through a property of the device object bearing the same name. This generator is very simplistic in its public interface and defines only the following members:

- **Generate** – this method will cause a pulse dialing digit to be generated; either immediately or at a specified time.

- **IsActive** – this returns true if a pulse dialing digit is currently being generated.

> The timing of the start and end of a pulse dialing digit is reported through signal notifications documented in section 10.

Example:

```
With _5620
    ' dial  '3' with default digit timing
    .PulseDialingGenerator.Generate(New PulseDialingDigit("3"c))

    ' wait up to 5 seconds for this digit to be generated
    .Wait.Until(ActionType.Pulse_Dialing_Generator_Stopped, _
                                TimeInterval.InSeconds(5))

End With
```

17.10. Signal Detection

17.10.1. Detected Signal List

An automatically updated list of detected signals is accessible through the DetectedSignal property of the AI5620_TE_Simuator device object (see section 16.4).

17.10.2. Line State Detector

The AI5620_TE_Simulator class detects line-state based signals using a sub-class of the LineStateDetector class (see section 16.2.1) which can be accessed through the LineState property. In addition to the features of the base class this derived class allows applications to configure the voltage and current threshold settings which are used to determine the telephone line state (see section 13.4).

The AI-5620 determines the telephone line state by comparing voltage and current measurements to a set of threshold settings as illustrated in Figure 34.
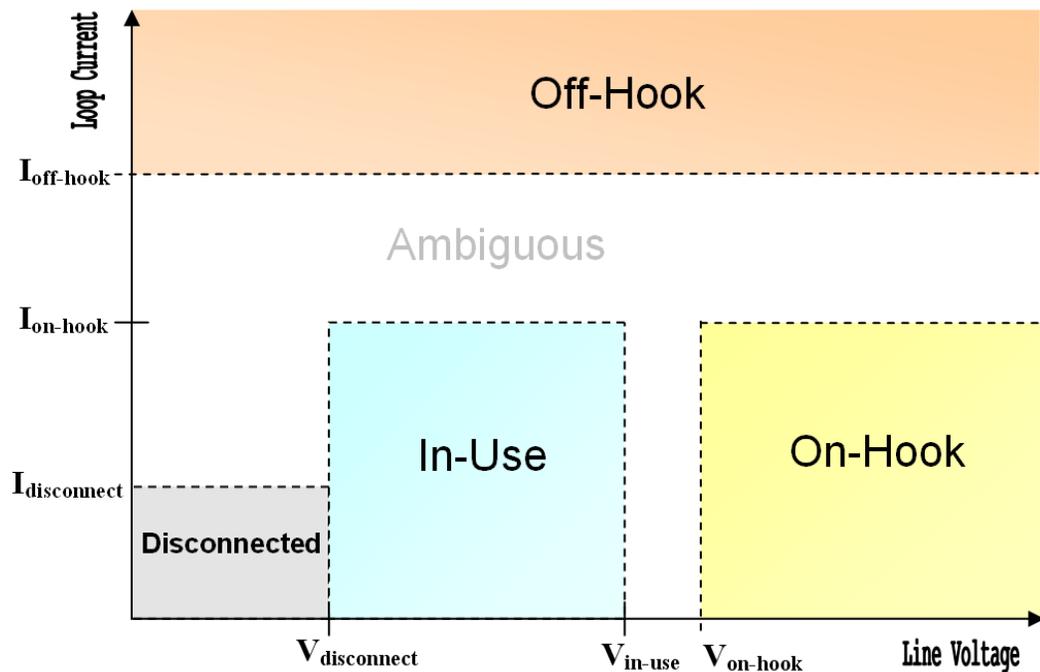
Figure 34 AI-5620 Telephone Line State Diagram

The AI-5620 continuously measures the line voltage and loop current present in the telephone interface circuitry and maps the measurements to one of the four possible telephone line states. When the instantaneous line state changes and persists for a short period of time the AI-5620 determines that the line state has changed and issues a notification. If the measurements do not fall into any of the well defined regions in Figure 34 the line state is considered "ambiguous" and it is assumed that no line state change has occurred. (Note: generally these "ambiguous" regions only happen in transition from one state to another).

The LineStateDetector class exposes the following threshold properties:

- **ThresholdDisconnectVoltage** ($V_{disconnect}$) – this specifies the voltage below which the telephone line may be considered disconnected (assuming the loop current is also below the disconnect threshold).

- **TheholdInUseVoltage** (V_{in-use}) – this specifies the voltage below which the telephone line may be considered to be "In Use" (assuming the line voltage is also above the disconnect threshold and the loop current is below the on-hook threshold)

- **ThresholdOnHookVoltage** ($V_{on-hook}$) – this specifies the voltage above which the telephone line may be considered to be "On Hook" (assuming the loop current is also below the on-hook threshold)

- **ThresholdDisconnectCurrent** ($I_{disconnect}$) – this specifies the loop current below which the telephone line may be considered to be disconnected (assuming the line voltage is also below the disconnect threshold).

- **ThresholdOnHookCurrent** ($I_{on-hook}$) – this specifies the loop current below which the telephone line may be considered to be "In Use" or "Off Hook" depending on line voltage.

- **ThresholdOffHookCurrent** ($I_{off-hook}$) – this specifies the loop current above which the telephone line may be considered to be off hook. (Assuming the telephone interface hook switch is off-hook or a fixed termination is applied).

Reset to Defaults

The LineStateDetector object exposes a ResetToDefaults method which will reset these threshold settings back to their default values. More specifically this method will assign the following default values:

Threshold	Default
$V_{disconnect}$	1 V
V_{in-use}	19 V
$V_{on-hook}$	21 V
$I_{disconnect}$	0.5 mA
$I_{on-hook}$	1 mA
$I_{off-hook}$	10 mA

17.10.3. DTMF Detector

The DTMF detector is accessible through the DTMFDetector property of the AI5620_TE_Simuator device object (see section 16.2.2).

17.10.4. FSK Detector

The FSK detector is accessible through the FSKDetector property of the AI5620_TE_Simuator device object. See section 16.2.3 for detailed class documentation.

17.10.5. Metering Pulse Detector

The AI-5620 contains a detector algorithm which can detect metering pulses (as defined in section 14.11) with up to three distinct frequency definitions. This feature is managed by the MeteringPulseDetector class and can be accessed through a property with the same name. This class exposes the following members:

- **FrequencyTollerance** – this specifies the maximum acceptable frequency deviation (in percent) from the template frequencies for valid metering pulses to be detected.

- **TemplateCount** – this returns the maximum number of metering pulse frequency definitions which can be simultaneously detected

- **TemplateFrequency** – this specifies the center frequency for a specific metering pulse template.

- **ResetToDefaults** – this resets the detectable templates back to 12 kHz and 16 kHz.

Example:

```
With _5620.MeteringPulseDetector

        ' detect 12 kHz metering pulses
        .TemplateFrequency(0) = Frequency.InkHz(12)

        ' also detect 14 kHz metering pulses
        .TemplateFrequency(1) = Frequency.InkHz(14)

        ' disable the third template
        .TemplateFrequency(3) = Nothing

        ' allow 1% frequency deviation
        .FrequencyTollerance = 1

End With
```

17.11. Instrument Time

Information regarding the AI-5620 device time base (discussed in section 9) is accessible through the Time property of the AI5620_TE_Simulator object. The returned object exposes the following members:

- **Epoch** – this returns the time stamp corresponding to the device time epoch (see section 9) which corresponds to the time when communications were first established.

- **MostRecent** – this property returns the time stamp information from the most recent status update from the instrument. This time value should be accurate to within 0.5 seconds.

- **Now** – this function actually polls the instrument and returns the current instrument time. The time stamp returned is accurate to within the communication delay (which varies but is normally in the order of 50 milliseconds)

17.12. Waiting

The AI5620_TE_Simulator class exposes a WaitManager object which assists applications in waiting for particular events of interest (see section 16.3).

17.13. Digital I/O

The AI-5620 TE Simulator is equipped with three digital outputs and two digital inputs which are available on the rear panel of the instrument. These digital inputs and outputs can be configured using the DigitalIO property of the AI5620_TE_Simulator object. The returned object exposes the following members:

- **GetDigitalInputs** – returns an array containing the states of each digital input

- **InputA** – returns the current logic state of digital input A. (0=false,1=true)

- **InputAMode** – this specifies the special purpose (if any) for digital input A. The available settings are:

  - **General Purpose Input** – no special purpose is assigned to the input. The logic value present on the input can be read using the InputA property.

  - **Hook Switch Control** – the digital input controls the hook switch within the AI-5620. Logic '1' corresponds to off-hook, logic '0' corresponds to on-hook.

- **InputB** – returns the current logic state of digital input B. (0=false,1=true)

- **OutputA** – specifies the output signal applied to digital output A. The available settings are:

  - **Output Low** – logic '0' will be applied to the digital output

  - **Output High** – logic '1' will be applied to the digital output

  - **FSK Decoder Output** – the demodulated bit pattern will be applied to the digital output.

- **OutputB** – specifies the output signal applied to digital output B. The available settings are:

  - **Output Low** – logic '0' will be applied to the digital output

  - **Output High** – logic '1' will be applied to the digital output

  - **Ring Detect** – the digital output will be set high when ringing is detected on the telephone interface.

- **OutputC** – specifies the output signal applied to digital output C. The available settings are:

  - **Output Low** – logic '0' will be applied to the digital output

  - **Output High** – logic '1' will be applied to the digital output

  - **Hook Switch Status**– the digital output will be set high ('1') when the telephone interface is taken off hook.

17.14. Instrument Protection

The firmware within the AI-5620 is designed to detect hazardous operating conditions and automatically take action to prevent permanent damage to the instrument. Typically these conditions include:

- Very high DC voltages applied to the telephone interface

- Very high currents flowing through the telephone interface

- Excessive power dissipation within the telephone interface circuitry

Depending on the nature of the hazardous condition, the firmware may take one of the following actions to protect the internal circuitry:

- Disconnect the entire telephone interface circuitry from the front panel jack. This disconnection may be temporary or until reset by the parent application.

- Disconnect the AC termination circuitry used to transmit AC signals on the telephone line.

- Disconnect the ringing loads from the telephone line.

> Whenever a protection mechanism is invoked within the AI-5620, the device object will deliver a ProtectionNotification object through the notification system documented in section 10

While the notification mechanism can inform a parent application that a protection mechanism has been engaged, the Protection property of the AI5620_TE_Simulator object exposes the following properties to determine the protection mechanisms which may be engaged on the instrument:

- **IsProtected** – this returns true when any protection mechanism is active within the AI-5620 instrument. For more specific information about the protection mechanism see the properties below.

- **IsTelIntDisconnectedStatically** – this will return true if the telephone interface circuitry has been statically disconnected. This disconnect state will remain in place until the parent resets the protection mechanism (see Reset method)

- **IsTelIntDisconnectedTemporarily** – this returns true if the telephone interface circuitry has been disconnected, but will be automatically reconnected by the firmware after a short delay.

- **AreRingingLoadsDisconnected** – this returns true if the ringing loads have been disconnected. Typically the firmware will automatically reconnect the ringing loads after a short period of time, or when the hazardous conditions have been removed.

- **IsACTerminationRemoved** – this returns true if the AC termination circuitry has been removed from the line to prevent damage. In this condition, the instrument will not be able to generate signals onto the telephone line. Generally the AC termination circuitry will be reconnected to the telephone line once the hazardous condition is removed.

Some hazardous operating conditions (extremely high voltages/ currents) will cause the instrument to disconnect the telephone interface statically. When this occurs, the application should **ensure that the hazardous conditions are removed** and then may reset the protection mechanism through the following method:

- **Reset** – this method will reset any static protection mechanism which may be engaged and reconnect the telephone interface.

The protection mechanisms built into Advent Instruments products are designed to **prevent permanent damage caused by operating conditions which are hazardous to the instrument's internal circuitry.** Such conditions can include:
- High voltages
- Excessive current draw
- Unbalanced current draw from a telephone interface
- Excessive power dissipation within the instrument

When a protection mechanism is engaged by an instrument: **ensure the hazardous condition is removed before resetting the protection mechanisms**! Repeated long term exposure to these hazardous conditions may still damage the instrument.

18. AI-7280 CO Simulator

One of the instruments supported by aiDevices is the AI-7280 Central Office Simulator which

- Simulates a central office (FXS) with programmable telephone interface characteristics

- Detects, analyzes, generates, and records telephony signals (such as DTMF, FSK, Caller ID, Line Flash, OSI, etc)

- Generates ringing and Caller ID signals

- Tests the functionality or compliance of Terminal Equipment (TE)

For a more detailed product information and specifications please refer to the "AI-7280 User Guide" which is available at www.adventinstruments.com.

The features of the AI-7280 instrument are accessed through the AI7280_CO_Simulator class within the aiDevices framework. Unless otherwise noted, all documentation within the following subsections refers specifically to the AI7280_CO_Simulator class.

18.1. Establishing Communications

Communications with the AI-7280 instrument can be established using one of the three static Connect methods as documented in section 8.4. Each of these functions will behave as follows:

- If an instrument is found which is supported by the class and communications are established successfully, then an instance of the device object will be created and returned. This object can then be used to control the connected instrument.

- If no supported instruments are found then null is returned.

- If an instrument is found but communications are not established correctly or the instrument is not supported by the device class then the function **will raise an Exception** which must be handled by the calling application.

When communications are established with an AI-7280 instrument through any of the Connect methods available:
- The device object **does not modify instrument settings** but rather **synchronizes** with the current state of the AI-7280 which may be left in a particular state by another application. This behavior is vital in situations when connections must be established and terminated with the AI-7280 without disturbing the telephone line state, signal routing, or signal generators.
- If your application requires a particular instrument configuration it must call ResetToDefaults after communications are established or adjust each device setting to the desired state.
- Certain features (such as recording) may not be able to be completely synchronized when communications are established and **may** reset such features to default settings.

Examples:

```csharp
AI7280_CO_Simulator Dev = null;
try {
      // connect to any available AI-7280
      Dev = AI7280_CO_Simulator.Connect();
      if (Dev==null)
      {
            // No instruments available!
      } else {
            if (Dev.Exceptions.Count !=0)
            {
                  // AI-7280 is connected but has
                  // reported an problem
            }
      }
}catch(Exception ex)
{
      // AI-7280 may be present but could not connect!
}
```

18.2. Terminating Communications

Once your application has finished using an AI7280_TE_Simulator object it must always call one of the Close methods before setting the object variable to null! Please refer to section 8.6 for detailed information regarding these Close methods.

> When communications are terminated with an AI-7280 instrument through a Close method:
> - All automated behavior (such as pattern generation, ringing patterns, scheduling, etc) will stop immediately.
> - All static device settings (such as line impedances, feed voltages, and signal routing) and simple signal generators (tone generators, echo, etc) will remain in their current states. This behavior may be desirable if the instrument is configured within a test fixture.
> - If your application requires a particular instrument state after communications are terminated then it should call ResetToDefaults (or ensure that each instrument setting is property configured) **before** calling Close

18.3. Resetting to Default Settings

In many applications it is desirable to reset the instrument settings to their default settings to return the device to a known operating condition. The AI7280_CO_Simulator class implements a ResetToDefaultSettings method as documented in section 8.7. In general this will:

- Stop signal generators and reset all signal generator settings to nominal defaults

- Reset all detector settings to nominal defaults

- Reset all telephone interface settings

- Reset all digital outputs to "Output Low" and disable all special functions

- Reset all signal routing, measurement settings, and filters to defaults

> This ResetToDefaultSettings **does not initiate a hardware reset** of the associated instrument but rather reconfigures the instruments with "safe" default values.

18.4. Determining Instrument Capabilities

The particular capabilities of the instrument's hardware and firmware in combination with the supporting device classes are reported through the Capabilities property of the AI7280_CO_Simualtor class. The AI-7280 specific capabilities are reported using the properties documented in the following sections.

> The capabilities of the tone generator, FSK generator, noise generator, echo generator, and ring generator are reported using the standard interfaces documented in section 8.3.

18.4.1. Telephone Interface Capabilities

The capabilities of the telephone interface are reported through the following properties

- **SourceVoltageMinimum**
  SourceVoltageMaximum – reports the range of supported range of DC feed voltages which can be used to program the telephone interface SourceVoltage property.

- **LoopCurrentMinimum**
  LoopCurrentMaximum – reports the supported range of loop currents which can be assigned to the telephone interface SourceCurrent property.

- **OffHookCurrentThresholdMinimum**
  OffHookCurrentThresholdMaximum – reports the support range of off hook current thresholds which can be programmed through the LineState.ThresholdOffHookCurrent property.

18.5. Signal Routing and Processing

The AI-7280 instrument can be configured to route a selection of internal signals throughout the instrument and the connectors on the front and rear panels of the instrument.

The signal routing capabilities of the AI-7280 can be accessed using a SignalManager object which is accessed through the Signals property of the AI7280_CO_Simulator class. The following sections describe the signal routing capabilities within the instrument and highlight the corresponding programming interface; starting with the signal generators shown in Figure 35.
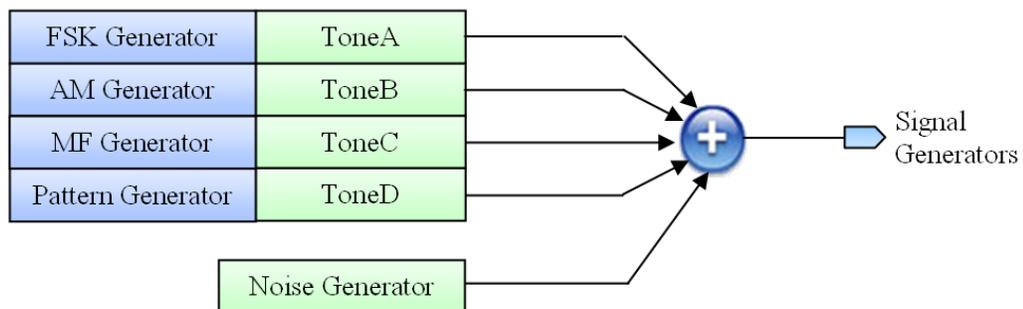

Figure 35 AI-7280 Signal Generator Routing Diagram

The outputs of each fundamental signal generator are summed together to produce the internal signal labeled "Signal Generators" as illustrated in Figure 35. This signal can then be routed to many signal blocks through the instrument including the telephone interface. A heavily simplified diagram of this telephone interface AC signal routing is shown below in Figure 36 (For DC related discussion see section 18.6).
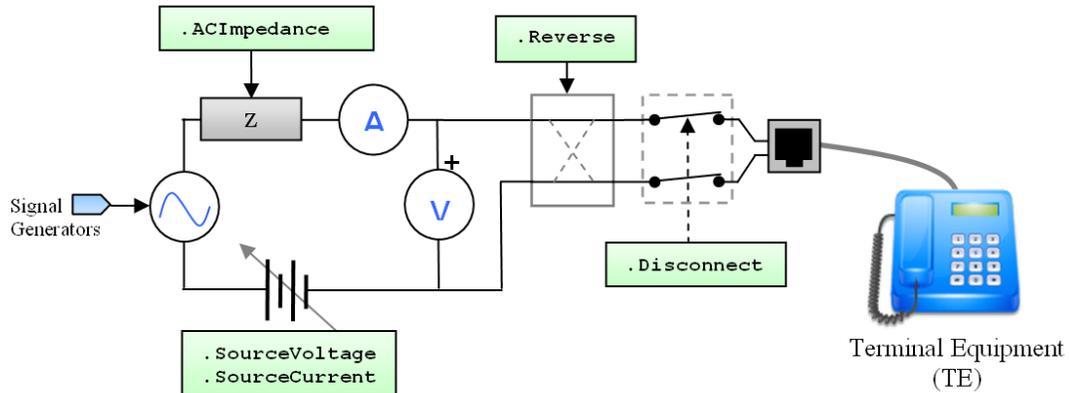

Figure 36 Simplified AI-7280 Telephone Interface AC Signal Routing

The output of the signal generators and the audio input from the rear panel can be combined and transmitted on the telephone line. The amplitude of these signals can be adjusted using the following method:

- **SetTelIntGains** – this method sets the gains applied to the signal generator output and audio input before the signals are summed and transmitted onto tip and ring.

This signal is then routed to the echo generator (see section 0) whose output is used as the internal signal labeled "TelInt Transmit" and is transmitted onto the tip-ring conductors.

The AC voltage signal measured from the telephone interface is yet another internal signal which is labeled "TelInt Receive" and is typically routed to the meter and signal detectors. The AI-7280 also contains a signal hybrid (which cancels transmitted signals from the received signals) which results in another signal labeled "TelInt Hybrid". Applications will tend to use this signal when attempting to isolate signals generated by other devices on the telephone line from those generated by the AI-7280.

The SignalManager also exposes the following methods:

- **AudioOutputSource** – selects the internal signal which is routed to the audio output connector on the rear panel

- **AudioOutputGain** – selects the gain factor which will be applied to signals which are routed to the audio output on the rear panel.

- **MeterSource** – selects the AC signal routed to the meter (see section 18.7)

- **AnalyzerSource** – selects the AC signal which is routed to the detectors within the instrument (see section 18.10)

- **ResetToDefaults** – this method will reset all the signal routing properties to defaults (see section 18.5.1)

18.5.1. Reset to Defaults

The SignalManager object exposes a ResetToDefaults method which will reset only the signal routing settings back to their default values. More specifically this method will:

- Set the audio output gain to zero and set the source to "none"

- Set the audio input gain to zero and the generator gain to 1

- Clear the main filter bank (no filtering) and places the filter before the meter

- Both the meter source and the detector source are configured to receive the "TelInt Receive" signal (see Figure 31)

18.6. Telephone Interface

The (FXS) telephone interface circuitry of the AI-7280 is accessible through a single RJ-11 connector on the front panel of the instrument. This telephone interface will supply DC feed voltage and loop current to one or more connected TEs. A heavily simplified version of the telephone interface is illustrated in Figure 37.
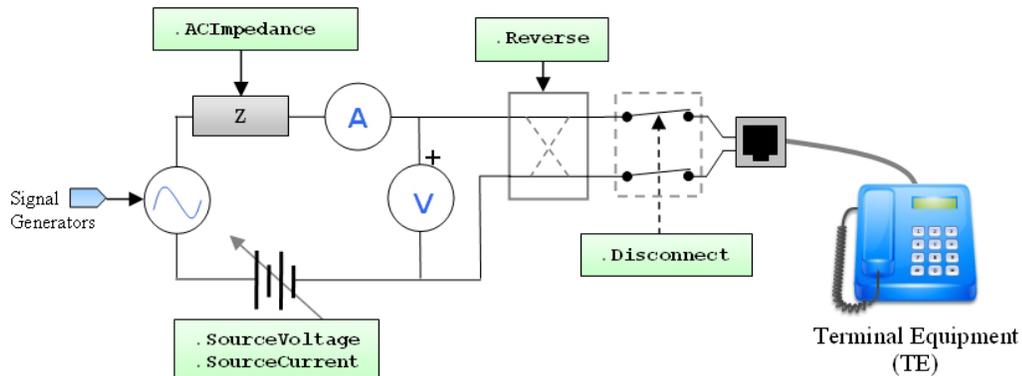
Figure 37 Simplified AI-7280 Telephone Interface

The AI7280_CO_Simulator class allows applications to assert full control over the features of this telephone interface circuitry through the TelInt property of the device object. This class exposes the following members:

- **SourceVoltage** – this specifies the DC feed voltage which will be applied at the telephone when on hook.

- **SourceCurrent** – this specifies the regulated DC current which will be sourced when a TE is off hook (and the applicable SourceMode is selected).

- **SourceMode** – this setting specifies the sourcing behavior of the telephone interface circuitry. The available settings are:

  o **Constant Current** – when this mode is selected the loop current will be regulated according to the SourceCurrent setting when a TE goes off hook.

  o **Constant Voltage** – when this setting is selected the AI-7280 will provide a constant voltage when a TE goes off hook and the loop current will be determined by this voltage setting and the total loop resistance including the fixed 400 Ω output resistance of the AI-7280.

- **OffHookCurrentThreshold** – this specifies the DC loop current above which the telephone line will be considered off hook and the telephone interface will switch to a current regulated source (if selected).

- **IsOffHook** – returns true if a TE is drawing sufficient loop current to put the telephone interface in the off-hook state. This current is determined by the OffHookCurrentThreshold property.

- **LineState** – returns the currently detected line state (see section 13.4).

- **ACImpedance** – this property specifies the AC impedance presented on the telephone interface. This impedance may be specified using an Impedance object specified in section 13.2 or one of the values returned by the FixedImpedancesAvailable list.

- **FixedImpedancesAvailable** – returns a list of the fixed impedances installed within the instrument. If optional impedances are installed they will appear in this list.

- **Balance** – this specifies the AC impedance of the telephone network as seen by the instrument. This is used in the instrument's signal hybrid and affects the trans-hybrid loss.

- **Connect** – this method will cause the telephone interface to be internally connected; either immediately or at a specified time.

- **Disconnect** – this method will cause the telephone interface to be internally disconnected; either immediately or at a specified time.

- **IsDisconnected** – this returns true if the telephone interface circuitry is currently disconnected internally.

- **Generate** – this method will cause an OSI to be generated; either immediately or at a specified time.

- **MeasurementPoint** – this selects the place within the telephone interface circuitry where the voltage measurements are taken. The available settings are:

  - **Inside Pair** – measurements are taken from the inside pair of telephone conductors on the RJ-11 connector (this is the same pair on which the DC feed is applied)

  - **Outside Pair** – measurements are taken from the outside pair of telephone conductors on the RJ-11 connector. This setting may be helpful when it is desirable to measure voltages other than those generated by the AI-7280 telephone interface.

- **MeasurementRange** – this selects the range of the AC signal measurements within the telephone interface. The available settings are:

  - **Normal Range** – when this setting is selected up to 5 Vrms may be measured without distortions in the measurements

  - **High Range** – when this setting is selected signals much larger than 5 Vrms may be measured however very low voltages will not be able to be measured accurately.

- **Polarity** – returns the line polarity of the telephone interface (see section 13.5)

- **Reverse** – this will reverse the polarity of the telephone interface circuitry with respect to the RJ-11 tip-ring conductors.

- **ResetToDefaults** – this method will reset the telephone interface to default settings (see section 17.6.1).

Example:

```vbnet
Imports Advent.aiDevices.AI7280_CO_Simulator

With _7280.TelInt

      ' Set the DC feed and off-hook current
      .SourceVoltage = DCVoltage.InVolts(48)
      .SourceCurrent = DCCurrent.InMilliAmps(26)
      .SourceMode =
AI7280_CO_Simulator.TelIntSourceMode.ConstantCurrent

      ' set TBR-21 output impedance
      .ACImpedance = Impedance.TBR_21
      .Balance = Impedance.Resistive_600

      ' connect the telephone interface to the RJ-11 connector
      .Connect()

      ' measure voltages on the inside pair
       .MeasurementPoint = TelIntMeasurementPoint.InsidePair
       .MeasurementRange = TelIntMeasurementRange.NormalRange

      ' reverse the line polarity
      .Reverse()

          ' go off hook at 10mA
          .OffHookCurrentThreshold = DCCurrent.InMilliAmps(10)

      'reverse the telephone line polarity
      .Reverse()

      ' generate an OSI
      .Generate(New OSI(TimeInterval.InMilliseconds(100)))

End With
```

18.7. Meters and Measurements

The AI-7280 instrument contains a signal meter which can be configured to perform filtering, level measurement, and frequency measurements on many signal sources available within the device. These features are implemented by the MeterManager class and are accessible through the Meter property of the AI7280_CO_Simulator class. The AC signal routing within the meter is illustrated below in Figure 33.
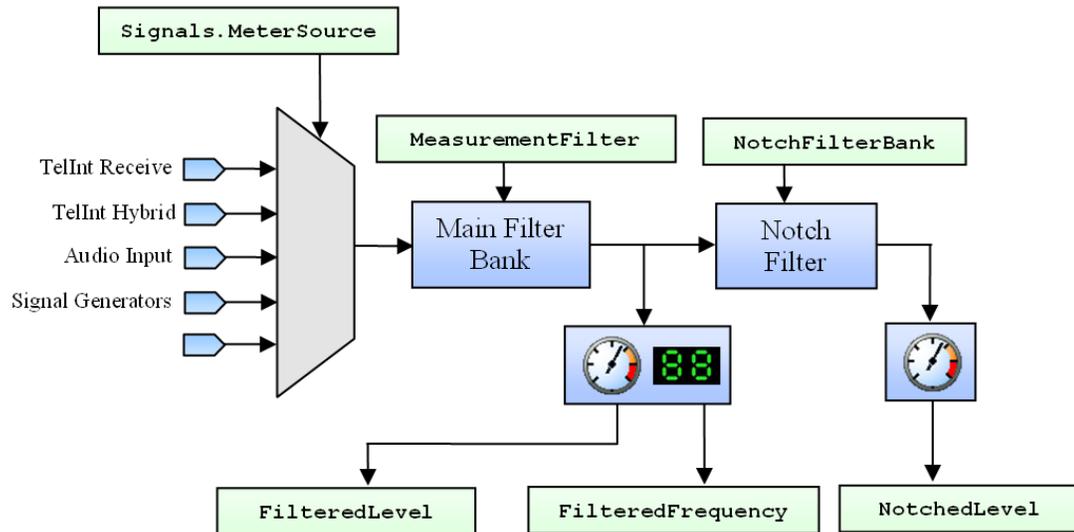

Figure 38 AI-7280 Meter Signal Routing Diagram

The AC signal routed to the meter is selected using the Signals.MeterSource property which is described in section 17.5. The MeterManger class exposes the following members which manage AC signal measurements.

- **MeasurementFilter** – this loads the main filter bank and places the filter before the meter measurements (see section 13.3).

- **UnfilteredLevel** – returns the current RMS level measurement of the input signal before any filtering is applied.

- **FilteredLevel** – returns the current RMS level measurement after the main filter bank.

- **FilteredFrequency** – returns a frequency measurement taken after the main filter bank.

- **NotchFilterBank** – configures the notch filters within the meter with up to two independent notch filters (see section 13.3).

- **NotchedLevel** – returns the current RMS level measurement taken after both the main filter bank and notch filter bank.

- **ACMeasurementSpeed** – this property configures the settling time and low frequency accuracy of each of the above level measurements. The allowed values are:

  - o **Very Fast** $\Rightarrow$ 25 ms settling time ($\pm$ 0.1 dB accuracy $f \geq 500$ Hz)

  - o **Medium** $\Rightarrow$ 100 ms settling time ($\pm$ 0.1 dB accuracy $f \geq 100$ Hz)

  - o **Slow** $\Rightarrow$ 400 ms settling time ($\pm$ 0.1 dB accuracy $f \geq 30$ Hz)

  - o **Very Slow** $\Rightarrow$ 1.1 second settling time ($\pm$ 0.1 dB accuracy $f \geq 10$ Hz)

All AC signal level measurements are affected the ACMeasurementSpeed which configures the settling time and low frequency accuracy of the level meters.

The MeterManger class also exposes the following members which manage DC-coupled signal measurements from the telephone interface:

- **LineVoltage** – returns the DC line voltage measured in the telephone interface.

- **LoopCurrent** – returns the DC loop current measured within the telephone interface.

- **UnbalancedCurrent** – returns the difference in loop current measurements between the tip and ring conductors.

- **DCMeasurementSpeed** – this property specifies the averaging applied to the DC-coupled measurements above. This averaging will affect the accuracy of the measurements in the presence of low frequency AC signals. The allows values are:

    o **No Filtering** $\Rightarrow$ No averaging is applied

    o **Medium** $\Rightarrow$ 0.5 second settling time ($\geq$ 40 dB rejection f $\geq$ 100 Hz)

    o **Slow** $\Rightarrow$ 2 second settling time ($\geq$ 40 dB rejection f $\geq$ 30 Hz)

    o **Very Slow** $\Rightarrow$ 5.5 second settling time ($\geq$ 40 dB rejection f $\geq$ 10 Hz)

The measurement manager also contains the following general members

- **GetMeasurements** – these methods returns a set of nearly simultaneous measurements from the meter

18.8. Instrument Status

Once communications are established with an AI-7280, the instrument's firmware and the device class cooperate to periodically send important status information to the device's StatusManager object (in the background) where it is accessible to the parent application without incurring communications delays. This status information is accessible through the Status property of the device object.

All properties of the StatusManager object (accessed through the Status property) can be read without incurring communication delays or any associated processing.

All information reported through the Status property is **automatically updated in the background from a worker thread** and is thread-safe. The application must be careful to note:
- The reported information may be updated at any time with respect to the execution of the application.
- The status information is updated at a fixed rate and may take as long as 400 milliseconds between refreshes (although typically it is faster)

The accessible status properties are:

- **LineState** – the most recently reported telephone line state (see section 13.4)

- **IsOffHook** – returns true if the most recent update indicates the telephone line was in the off-hook state.

- **IsRinging** – returns true if the most recent update indicated that the ringing generator was active.

- **MeterLevel** – the most recent filtered level measurement from the meter (see section 18.7)

- **MeterFrequency** – the most recent frequency measurement from the meter (see section 18.7)

- **LineVoltage** – the most recent line voltage measurement from the meter (see section 18.7)

- **LoopCurrent** – the most recent loop current measurement from the meter (see section 18.7)

18.9. Signal Generation

18.9.1. Tone Generators

Each AI-7280 is equipped with four tone generators whose features are accessible through the ToneA, ToneB, ToneC, and ToneD properties. Each of these properties returns a ToneGenerator object which can be used to generate simple tones or can be reserved by higher level signal generators to produce more complicated signaling (see section 16.1.1).

18.9.2. Pattern Generator

The AI7280_CO_Simualtor object supports the generation of multi-tone patterns through the PatternGenerator class which is returned by the property with the matching name (see section 17.9.2).

18.9.3. AM Generator

The AI-7280 is equipped with am AM generator which can be accessed through the AMGenerator property (see section 16.1.4).

18.9.4. Echo Generator

The AI-7280 is equipped with an echo generator for simulating telephone network impairments which is accessible through the EchoGenerator property (see section 0).

18.9.5. FSK Generator

The AI-7280 is equipped with an FSK generator which can be accessed through the FSKGenerator property (see section 16.1.3).

18.9.6. MF Generator

The AI-7280 is equipped with an MF generator for generating dual tone signals which can be accessed through the MFGenerator property (see section 16.1.2).

18.9.7. Noise Generator

The AI-7280 is equipped with a white noise generator which can be accessed through the NoiseGenerator property (see section 16.1.6).

> The main filter bank can be configured to filter the output of the noise generator in order to band-limit the resultant noise signal (see section 17.5).

18.9.8. Ringing Generator

The AI-7280 is equipped with a ringing generator which can generate ringing signals (see section 14.12) which are generated in the on-hook line state and used to cause TEs to alert a customer to an incoming call. The features of the ringing generator are accessible through the RingGenerator property of the AI-7280_CO_Simulator class. The RingGenerator class exposes the following members.

- **Generate** – these methods will begin generating a ringing signal (see section 14.12) either immediately or at a specified time. Ringing may also be started with a specific cadence (see section 14.4).

- **StopGenerator** – this immediately stops the ringing generator.

- **Level** – specifies/updates the signal level of the ringing signal

- **DC** – specifies the DC offset during ringing

- **Freq** – specifies/updates the frequency of the ringing signal

- **Shape** – specifies/updates the shape of the ringing signal (see section 14.3).

- **Update** – these methods will update the ringing generator with new level, DC, frequency, and shape information.

- **ResetToDefaults** – this resets the tone generator back to default settings

- **IsActive** – returns true if the ringing generator is currently active

> The ringing generator will stop automatically when an off hook condition is detected.

Example:

```vbnet
With _7280.RingGenerator

        ' Define ringing with 80 Vrms @ 22 Hz and 48V DC
        Dim R = New Ringing(SignalLevel.InVrms(80),
                            Frequency.InHz(22),
                            DCVoltage.InVolts(48))

        ' generate ringing with 2 seconds on, 4 seconds off
        ' continuing indefinitely
        Dim RP = New Cadence(TimeInterval.InSeconds(2),
                        TimeInterval.InSeconds(4))

        ' Generate ringing with a pattern
        .Generate(R, RP)

        Thread.Sleep(3000)   ' wait a bit

        ' Change the signal level to 60 Vrms
        .Level = SignalLevel.InVrms(60)

        ' Immediately stop the ringing generator
        .StopGenerator()
End With
```

18.9.9. Metering Pulse Generator

Metering pulses are signals sent by telephone exchanges to telephones to inform the customer of the expense of the phone call (see section 14.11). Usually each pulse represents a particular incremental cost and more expensive calls will result in more pulses sent per minute. The AI-7280 is equipped with a metering pulse generator which can be configured to transmit metering pulses at a very well controlled rate. The metering pulse generator is managed by the MeteringPulseGenerator and can be accessed through the property with the same name. This class exposes the following members:

- **Generate** – these methods will cause the generator to immediately begin generating metering pulse signals; either indefinitely until stopped or for a specified number of pulses.

- **StopGenerator** – this immediately stops the metering pulse signal generator.

- **IsActive** –returns true if the metering pulse generator is currently active

- **IsGeneratingIndefinitely** – returns true if the generator is configured to continue transmitting metering pulses indefinitely (until stopped)

- **Signal** - this property specifies the metering pulse signal which is generated periodically (see section 14.11). This can be modified while the generator is active.

- **Interval** – this specifies the time interval between consecutive metering pulses. This property can be updated while the generator is active

- **Count** – returns the total number of pulses remaining to be generated. If pulses are being generated indefinitely this returns a negative value.

- **ResetToDefaults** – this stops the metering pulse generator and resets all the settings back to defaults.

```
Example:

With _7280.MeteringPulseGenerator

      ' Define a metering pulse
      Dim MP As New MeteringPulse( _
                    SignalLevel.InVrms(0.3), _
                    Frequency.InkHz(12), _
                    TimeInterval.InMilliseconds(200))


      'Start generating 100 pulses separated by 10 seconds
      .Generate(MP, TimeInterval.InSeconds(10), 100)

      'Start generating pulses separated by 60 seconds
      ' indefinitely (until stopped)
      .Generate(MP, TimeInterval.InSeconds(60))

      ' stop the metering pulse generator
      .StopGenerator()

End With
```

18.10. Signal Detection

18.10.1. Detected Signal List

An automatically updated list of detected signals is accessible through the DetectedSignal property of the AI7280_CO_Simuator device object (see section 16.4).

18.10.2. Line State Detector

The AI7280_CO_Simulator class detects line-state based signals using a sub-class of the LineStateDetector class (see section 16.2.1) which can be accessed through the LineState property. In addition to the features of the base class this derived class allows applications to configure the current threshold setting which is used to determine the telephone line state (see section 13.4).

The LineStateDetector class exposes the following threshold properties:

- **ThresholdOffHookCurrent** ($I_{off\text{-}hook}$) – this specifies the loop current above which the telephone line may be considered to be off hook.

Reset to Defaults

The LineStateDetector object exposes a ResetToDefaults method which will reset these threshold settings back to their default values. More specifically this method will assign the following default values:

Threshold	Default
$I_{off\text{-}hook}$	10 mA

18.10.3. DTMF Detector

The DTMF detector is accessible through the DTMFDetector property of the
AI7280_CO_Simuator class (see section 16.2.2).

18.10.4. FSK Detector

The FSK detector is accessible through the FSKDetector property of the
AI7280_CO_Simuator class (see section 16.2.3).

18.11. Recording

The AI-7280 is capable of recording AC and DC samples from the telephone interface or
audio input connector and downloading the resulting samples to the host computer.
Applications may access these recording features through the ACRecording and
DCRecording properties of the AI7280_CO_Simulator class. Each of these objects is an
implementation of the RecordingManager documented in section 16.7.

Developers should be aware of the following limitations particular to the AI-7280.

> Multiple simultaneous downloads are not possible within the AI-7280. Recordings must
> be stopped before downloads can be started.

18.12. Instrument Time

Information regarding the AI-7280 device time base (discussed in section 9) is accessible
through the Time property of the AI7280_CO_Simulator object. The returned object
exposes the following members:

- **Epoch** – this returns the time stamp corresponding to the device time epoch (see
  section 9) which corresponds to the time when communications were first
  established.

- **MostRecent** – this property returns the time stamp information from the most
  recent status update from the instrument. This time value should be accurate to
  within 0.5 seconds.

- **Now** – this function actually polls the instrument and returns the current
  instrument time. The time stamp returned is accurate to within the
  communication delay (which varies but is normally in the order of 50
  milliseconds)

18.13. Waiting

The AI7280_CO_Simulator class exposes a WaitManager object which assists
applications in waiting for particular events of interest (see section 16.3).

18.14. Digital I/O

The AI-7280 is equipped with three digital outputs and two digital inputs which are available on the rear panel of the instrument. These digital inputs and outputs can be configured using the DigitalIO property of the AI7280_CO_Simulator object. The returned object exposes the following members:

- **GetDigitalInputs** – returns an array containing the states of each digital input

- **InputA** – returns the current logic state of digital input A. (0=false,1=true)

- **InputB** – returns the current logic state of digital input B. (0=false,1=true)

- **OutputA** – specifies the output signal applied to digital output A. The available settings are:

    o **Output Low** – logic '0' will be applied to the digital output

    o **Output High** – logic '1' will be applied to the digital output

    o **Hook Switch Status** – the digital output will be set high when the telephone interface is in the off hook state.

- **OutputB** – specifies the output signal applied to digital output B. The available settings are:

    o **Output Low** – logic '0' will be applied to the digital output

    o **Output High** – logic '1' will be applied to the digital output

    o **FSK Decoder Output** – the demodulated bit pattern will be applied to the digital output.

- **OutputC** – specifies the output signal applied to digital output C. The available settings are:

    o **Output_Low** – logic '0' will be applied to the digital output

    o **Output_High** – logic '1' will be applied to the digital output

- **ResetToDefaults** – this will cause all the digital outputs to be set to the "output low" state.

18.15. Protection Mechanisms

The firmware within the AI-7280 is designed to detect hazardous operating conditions and automatically take action to prevent permanent damage to the instrument. Typically it will detect only one condition:

- High unbalanced current flow from the telephone interface.

When this occurs the instrument will disconnect the entire telephone interface circuitry from the front panel jack for a short period of time and then reconnect. If the unbalanced condition still exists it will disconnect the telephone line and repeat as necessary.

> Whenever a protection mechanism is invoked within the AI-7280, the device object **will deliver a ProtectionNotification object documented in section 0** to any listening applications.

While the ProtectionNotification object informs the parent application that a protection mechanism has been engaged, the Protection property of the AI7280_TE_Simulator object exposes the following properties to determine the protection mechanisms which may be engaged on the instrument:

- **IsProtected** – this returns true when any protection mechanism is active within the AI-7280 instrument.

- **IsTelIntDisconnectedTemporarily** – this returns true if the telephone interface circuitry has been disconnected, but will be automatically reconnected by the firmware after a short delay.

At present there are no static protection mechanisms which can be reset by the application software. All mechanisms are automatically reset by the application.

19. Terminology and Definitions

Cadence	The on/off timing structure which can be specified to generate signals with particular timing.
Caller ID	A general term which refers to a caller identification service which is available in many phone networks which deliver information about the calling party to terminal equipment when a new incoming call is established.
CAS	A short dual tone signal which used to signal TEs of incoming Caller ID transmissions (see section 14.10).
Checksum	A fixed length number appended to a data transmissions which is calculated from the message contents and is used to detect errors in transmission (see section 14.19).
Complex Impedance	In telephony this generally refers to the impedance characteristic produced by a series RC circuit in the form $R_s + [R_p \| C_p]$ (see section 13.2).
CO	**(Central Office)** See also FXS.
CPE	**(Customer Premises Equipment)** See also FXO.
Decibel (dB)	A logarithmic unit of measurement which expressed the magnitude of a physical quantity relative to a specified or implied reference. $$dB = 20\log\left[\frac{Value_{measured}}{Value_{reference}}\right]$$ For example dBV measures voltage relative to 1 Volt RMS.
Descriptor	A class/object which is used to specify (describe) a real entity by collecting descriptive characteristics. For example a class might contain "red", "spherical", "3 cm diameter" and might be used as a descriptor for a ball.
Device Class	A class which communicates with and controls an Advent Instruments hardware product (see section 8).
Device Support Class	A class which implements a particular feature of an instrument and is accessed through a property of a device class.
Device Time	A timing system measured in seconds from the instant when communications were established with an instrument (see section 9).

DTMF	**(Dual Tone Multiple Frequency)** an in-band signaling technique used for touch tone dialing and Caller ID delivery (see section 14.9).
Epoch	The instant in time when communications were initially established with an instrument; corresponding to a device time of zero (see section 9).
ETSI	**(European Telecommunications Standards Institute)**
Flash	A signaling method used by Terminal Equipment (TE) to signal a central office by quickly hanging up and then picking up again. Historically this signal has been used to activate features such as call waiting or three-way calling (see section 14.15).
FSK	**(Frequency Shift Keying)** A signal modulation technique used to transmit binary data in which a sinusoidal carrier is shifted between two discrete frequencies (see section 14.18).
FXO	**(Foreign Exchange Office)** See also TE.
FXS	**(Foreign Exchange Station)** An analog telephone signaling interface which supplies power (DC feed, loop current), generates ringing signals, and can send and receive voice band signals with an FXO. See also CO,
Immutable Class	A class whose member values cannot be changed once the object is created. Immutable classes are often descriptors.
Impedance	Measure of opposition to alternating or direct current within a circuit (see section 13.2).
Instrument	Advent Instruments hardware external to the computer.
Mark Out	The number of "mark" bits appended to the end of an FSK transmission after the message contents.
MDMF	(**Multiple Data Message Format**) A format of Caller ID message data specified by the TIA and ETSI standard bodies.
Metering Pulse	A signal periodically sent by telephone exchanges to telephones to inform the customer of the relative expense of a phone call in progress (see section 14.11).
MF	**(Multi Frequency)** An in-band signaling technique used within telephony networks as a form of trunk signaling to route telephone calls. This technique was a precursor to DTMF signaling and also consisted of two simultaneously applied tones.
Notification	A descriptor object which is passed by device objects to notify any listening applications of important information with minimal latency. Such events include detected signals, internal exceptions, and completed actions.
OSI	**(Open Switching Interval)** is a signaling method which can be used by a central office to signal terminal equipment to either release the line (hang up) or that Caller ID is being delivered (see section 14.16).
POTS	(**Plain Old Telephone Service**) This term refers to an older style telephone that supports only the basic services.

Ringing	A high voltage periodic AC signal which is generated by a Central Office (CO) and is detected by TEs which cause them to make a sound (or other indication) which alerts the customer to an incoming phone call (see section 14.12)
RMS	(**Root Mean Square**)A statistical measure of the magnitude of a varying quantity calculated as: $$V_{rms} = \sqrt{\frac{1}{T_2 - T_1} \int_{T_1}^{T_2} [V(t)]^2 \, dt}$$ In electrical circuits the RMS voltage and current measures are easily related to power and are more commonly used to express the magnitude of time varying signals.
SDMF	(**Single Data Message Format**) A format of Caller ID message data specified by the TIA standard body.
Signal	Any phenomena which can be generated or detected by an instrument.
SMS	(**Short Message Service**) a communications service which delivers short text messages between compatible telephone handsets.
Static/Shared	A variable or method which is declared "statically" and whose scope applies to a particular class and not an instance of an object. (C# uses the term "static", VB.Net uses the term "shared")
TE	(**Terminal Equipment**) An analog telephone signaling interface (phone, fax, modem, etc) which can generate on and off hook signaling, accept ringing signals, and typically can send and receive voice band signals.
Telephone Interface	The circuitry which is used to interface a device to a telephone circuit. A telephone interface may be either an FXO or FXS.
TIA	(**Telecommunications Industry Association**) A North American standardization organization.
VMWI	(**Visual Message Waiting Indicator**) A Caller ID message typically sent to inform the customer of an unread voice mail message

20. Revision History

Nov 2009	Assembly = 1.0.0.0	File=1.0.0.0 (BETA)

First official beta release for customer review and acceptance testing.

Dec 18, 2009	Assembly =1.0.1.0	File=1.0.1.0 (BETA)

Second official beta release for customer review and acceptance testing.

Jan 1, 2010	Assembly =1.1.0.0	File=1.1.0.0

First official public release

Feb 19, 2010	Assembly =1.1.0.0	File=1.1.1.0

Fixes
- Resolved issue with AI-7280 Signal.OutputSource and Signal.AnalyzerSource which affected the incorrect firmware property
- Fixed bug in AI-7280 DC downloading which could produce resource conflict exceptions and possibly corrupted data
- Fixed AI-5620 Protection.Reset so it correctly re-connects the telephone interface when protection mechanism was engaged
- Fixed the AI-5620 ringing load property which could sometimes select the wrong load in the firmware

New Features
- Added the ArraySampleWriter class which supports the download of recorded samples into arrays

March 5, 2010	Assembly =1.1.2.0	File=1.1.2.0

Fixes
- Fixed bug in AI-7280 AC and DC downloading which produced erroneous resource conflict exceptions

New Features
- Added the IsBusy property to all signal generators which indicate if the generator is currently generating a signal or is scheduled to generate a signal in the future.

March 8, 2010	Assembly =1.1.3.0	File=1.1.3.0

Fixes
- Fixed bug in AI-7280 AC and DC recording manager in which very short duration single-shot recordings were missed.

May 13, 2010	Assembly =1.1.4.0	File=1.1.4.0

Fixes
- Improved the clarity of the exception message which occurs when the firmware version is out of date
- Removed accidentally public method from AI-7280 AC recording manager.

June 28, 2011	Assembly=1.1.5.0	File=1.1.5.0

Fixes
- Resolved issue for both AI-5620 and AI-7280 where DTMF digits would occasionally not get reported until a subsequent digit arrived.

July 4, 2011	Assembly=1.1.6.0	File=1.1.6.0

Fixes
- Resolved pulse dialing issue for both AI-5620 and AI-7280 where pulse dialing digits were not properly detected with fast (20pps+) pulse dialing.

July 6, 2011	Assembly=1.1.7.0	File=1.1.7.0

Fixes
- Resolved issue in AI-7280 support script which caused intermittent pulse dialing detection at fast rates (20pps) due to missed on/off hook signals

Nov 14, 2011	Assembly = 1.1.8.0	File = 1.1.8.0

New Features
- Added support for the AI-7280 Expanded DC Feed Option which allows the output voltage and current range to 105 V and 105 mA.

Nov 21, 2011	Assembly = 1.1.9.0	File = 1.1.9.0

Fixes
- Fixed bug in MF Generator manager (affecting AI-5620 and AI-7280) where digits were sometimes incorrectly mapped when uploaded to the firmware causing incorrect tone sequences to be generated.

Nov 22, 2011	Assembly = 1.1.10.0	File = 1.1.10.0

Fixes
- Fixed bug in DetectedSignalList which caused a DuplicateKeyException when certain signals with the same time stamp are added to the list.

Dec 20, 2011	Assembly = 1.1.11.0	File = 1.1.11.0

Fixes
- Fixed bug in MDMF Caller ID Decoding where it incorrectly reported the absence of Name information in TIA specified formats
- Fixed bug which occasionally truncated detected FSK messages by 1 byte.
- Fixed bug in AI-7280 metering pulse generator code which caused a transient signal at the start of the first pulse
- Corrected the AI-7280 TelInt.Linestatus so it returned the current telephone line state and not the one reported in the last status update.
- Fixed bug in AI-5620 pulse dialing scheduler which caused some digits to be started at the wrong time

New Features
- Incorporated new firmware scheduling features into the AI-7280 OSI, noise, and on/off hook scheduling to improve timing accuracy

Feb 27, 2012	Assembly = 1.1.12.0	File = 1.1.12.0

Fixes
- Improved latency on pulse dialing detection for both AI-7280 and AI-5620.

March 5, 2012	Assembly = 1.1.13.0	File = 1.1.13.0

Fixes
- Fixed parsing bug in device file system management code which would fail with certain regional settings causing an exception claiming a flash file item was invalid. This affected both AI-5620 and AI-7280

March 9, 2012	Assembly = 1.1.14.0	File = 1.1.14.0

Fixes
- Fixed bug in device file system management code which failed on invalid setup dates in the firmware version information with certain windows regional settings which caused device initialization to fail.

June 7, 2012	Assembly = 1.1.15.0	File = 1.1.15.0

New Features
- Added ability to disable FSK receiver during FSK transmissions to reduce application latencies.

June 14, 2012	Assembly = 1.1.16.0	File = 1.1.16.0

Fixes
- Improved the AI-7280 FSK data upload speed to reduce latencies in FSK data transmissions.

21. Technical Support

For technical support or general questions for this or any Advent Instruments product, please contact us in any of the following methods.

- **Email**:

Technical Questions:	techsupport@adventinstruments.com
Sales Inquires:	sales@adventinstruments.com

- **In North America**:

Tel:	(604) 944-4298
Fax:	(604) 944-7488
Mail:	Advent Instruments Inc.
	111 - 1515 Broadway St.
	Port Coquitlam, BC, V3C6M2
	Canada

- **In Asia**:

Tel:	(852) 8108-1338
Fax:	(852) 3909-2338
Mail:	Advent Instruments (Asia) Ltd.
	Unit 42, 18/F., Block D,
	Wah Lok Industrial Centre, Phase II,
	31 / 35 Shan Mei Street,
	Fotan, Shatin, New Territories, Hong Kong