Ham Cockpit Plugins

Developer's Guide

Table of Contents

Overview

Ham Cockpit is a universal, plugin-based software for Radio Amateurs. The functions available in the program depend entirely on the set of installed plugins. Some standard plugins come with the software, a wide variety of optional and third party plugins is (or will be) available on the Internet.

The standard plugins make Ham Cockpit an SDR client that interfaces with SDR radios, processes the I/Q and audio streams, and outputs processed audio to a sound card. A band scope and waterfall display plugins also come as standard.

Third party developers can create all kinds of plugins, both freeware and shareware, limited only by the imagination, including:

- plugins that replace the standard plugins, with more functions or better characteristics,
- plugins that add new signal processing functions, such as denoising, notch filters, diversity combining algorithms, demodulators for new modes, decoders for the digital modes, skimmers, and interfaces to the new types of SDR radios;
- all kinds of non-DSP plugins for QSO logging, cluster monitoring, award tracking, propagation prediction, contesting, SO2R operation, pileup management, etc.

By selecting the proper plugins, the end user can build a system optimized for his or her particular needs, and have all functions, previously spread across multiple programs, in one integrated environment.

MEF Framework

The plugin system of Ham Cockpit is based on the MEF library [1] that is part of the .NET Framework 4. MEF automatically discovers and loads the plugins, and makes connections between the interfaces exported and imported by the plugins and the host application.

To be found by Ham Cockpit, a plugin must be placed in the *Plugins* subdirectory of the program's installation directory, and must export the `IPlugin` interface described in the Implementing the IPlugin Interface section. The plugins may optionally export other interfaces defined by the host application, such as `IVisualPlugin`, if they have a visual representation, or `ISignalSource` if they input signals from an SDR radio.

In addition to the standard interfaces, the plugins may define their own interfaces for communication between the plugins. For example, a plugin that downloads spots from the DX clusters could define and export its own interface, e.g., `ISpotSource`, and a band map plugin could import this interface. Upon program startup, MEF would connect the export to the import, and the bandmap plugin would be able to receive cluster spots through this interface and plot them on the band map. All such connections are established by MEF automatically.

Plugin Assembly

A Ham Cockpit plugin is a .NET assembly containing one or more classes that implement and export the `IPlugin` interface. The assembly name must have the following structure:

`<author_id>.HamCockpitPlugins.<plugin_name>`

where `<author_id>` is a unique identifier of the plugin author. If you have a Ham callsign, use it as an author_id. For example, if UR5EMI created a Volume Control plugin, the assembly name would be

`UR5EMI.HamCockpitPlugins.VolumeControl`

This would prevent a name conflict with the standard Volume Control plugin in the assembly named

`VE3NEA.HamCockpitPlugins.VolumeControl`

Exporting Interfaces

`IPlugin` and all other standard interfaces are defined in the *VE3NEA.HamCockpit.PluginAPI* assembly that comes with the Ham Cockpit application. The plugin projects must import this assembly to gain access to the interface definitions. Right-click on *References* in your project, click on *Add Reference, Projects, Browse*, navigate to the installation folder of Ham Cockpit and select the assembly DLL.

Add to your project a reference to the MEF framework: right-click on *References*, click on *Add Reference, Assemblies, Framework,* and tick the check box next to the `System.ComponentModel.Composition` assembly.

Include in your code the `using` directives for the two assemblies mentioned above:

```
using VE3NEA.HamCockpit.PluginContracts;
using System.ComponentModel.Composition;
```

To export an interface to MEF, add the Export decorator to the class that implements the interface:

```
[Export(typeof(IPlugin))]
class ClockDemo : IPlugin
{
  ...
}
```

See the MSDN tutorial on MEF [1] for a detailed description of interface exports and imports.

Implementing the IPlugin Interface

The `IPlugin` interface defines the basic functionality that must be implemented by all plugins. It is defined like this:

```
public interface IPlugin
{
  string Name { get; }
  string Author { get; }
  bool Enabled { get; set; }
  object Settings { get; set; }
  UserControl SettingsEditor { get; }
  ToolStrip ToolStrip { get; }
  ToolStripItem StatusItem { get; }
}
```

- **`Name`**: a friendly name of the plugin, as it appears in the user interface;
- **`Author`**: the author id, the same as was used in the assembly name;
- **Enabled**: the user can enable and disable the plugins. The plugins may use this flag to turn their functions on and off. For example, a DX cluster client plugin could connect and disconnect to the cluster server when this property changes;
- **Settings**: a plugin can create a Settings object and place all of its settings in it. Using this object, the host application saves the settings of all plugins on exit an loads them on the next start, and provides a centralized way of editing the settings. See The Settings Property section for more detail;
- **Settings Editor**: a plugin may provide a custom editor for its settings if the standard `PropertyGrid` control is not enough. Not yet implemented;
- **ToolStrip**: a plugin can provide a `ToolStrip` that will appear on the toolbar when the plugin is enabled. The user can re-order the toolstrips using drag-and-drop. By default, the toolsrtips are aligned to the left side of the toolbar, but those that have the RightToleft property set to Yes are aligned to the right. Two examples of plugins with toolstrips described in the Sample Plugins section are Volume Control and Clock Demo.
-

Fig.1. The toolbar containing the standard toolstrip and the toolstrips of the Volume Control and Clock Demo plugins.

- **StatusItem**: a plugin can provide a `ToolStripItem` that will appear on the status bar. Typically this is used to show the status of the plugins that work in the background and do not have a visual interface. For example, a DX cluster client plugin could show the status of its connection to the server. By convention, the red background of the item indicates an error, and the yellow background indicates a warning. Conventional Radio is an example of a plugin with a status item.

Fig.2. The status item of the Conventional Radio plugin. The red background indicates an error, the mouse-over tooltip shows the error message. A mouse-click on the icon opens the OmniRig dialog.

The Settings, SettingsEditor, ToolStrip and StatusItem properties may return `null` if the plugin does not implement them.

.

Here is an implementation of the `IPlugin` interface in the Clock Demo plugin:

```
// IPlugin
public string Name => "Clock Demo";
public string Author => "VE3NEA";
public bool Enabled { get; set; }
public object Settings { get => GetSettings(); set => ApplySettings(value as Settings); }
public UserControl SettingsEditor => null;
public ToolStrip ToolStrip { get; } = new ToolStrip();
public ToolStripItem StatusItem => null;
```

The Settings Property

Ham Cockpit has a standard mechanism for saving and loading the plugin settings, and for allowing the user to edit the settings marked as editable. To make use of this mechanism, the plugin developer must create a class descending from `Object` that contains all of plugin's settings as public properties, and make it available to the host via the read-write Settings property.

A typical plugin has two kinds of settings: the settings that the user can change in the Settings dialog, such as the font size, and the settings that need to be saved and restored but should not be editable, such as the window size. Both types of the settings should be stored in the Settings object. The decorators define if the property is visible in the editor or not, and how it appears in the editor.

A plugin creates a Settings object and populates it with the default values of the properties.
On program start, the host reads the Settings property from the plugin, loads the saved values in it if they were previously saved, and assigns the modified Settings back to the plugin. When the user opens the Settings dialog, the host reads Settings from the plugin and displays them in the editor. When the user saves the changes, the modified Settings object is assigned to the Settings property of the plugin.

The `PropertyGrid` control is used in the Settings dialog for editing the plugin settings. There are many ways to customize the appearance of the properties in the control using the decorators. Good tutorials are available here: [2], [3].

Below is the Settings class used in the Clock Demo plugin. It has two editable properties and a non-editable one.

```
public class Settings
{
  [DisplayName("Blink")]
  [Description("Time separator blinks")]
  [DefaultValue(false)]
  public bool Blink { get; set; } = false;

  //this setting is saved/restored and editable by user
  [DisplayName("Dock to Right")]
  [Description("Dock to the right side of the window")]
  [DefaultValue(true)]
  public bool DockToRight { get; set; } = true;

  //this setting is saved/restored but not editable
  [Browsable(false)]
  public bool UtcMode{ get; set; } = false;
}
```

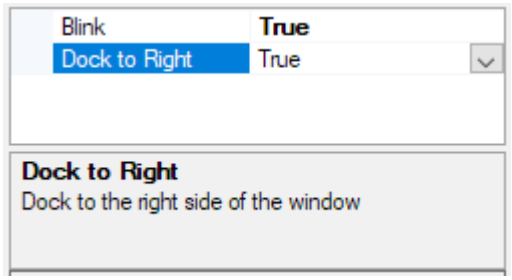

Fig.3. The settings of the Clock Demo plugin in the Settings dialog.

The *PluginAPI* assembly includes some type converters that the plugins can use to further customize the property grid control. The Conventional Radio plugin uses two such type converters, `InputSoundcardNameConverter` for the soundcard selection:

```
  [Description("Soundcard to be used for audio input from the radio")]
  [TypeConverter(typeof(InputSoundcardNameConverter))]
  public string Soundcard { get; set; }
```

and `EnumDescriptionConverter` for displaying the friendly names if the `enum` values instead of their identifiers:

```
enum InputType {
  [Description("Left Channel")]
  LeftChannel,
```

```
    [Description("Right Channel")]
    RightChannel,

    [Description("Diversity (stereo)")]
    DiversityStereo
  }
```

```
    [DisplayName("Soundcard Channel")]
    [Description("Audio channel to use")]
    [TypeConverter(typeof(EnumDescriptionConverter))]
    [DefaultValue(InputType.LeftChannel)]
    public InputType InputType { get; set; }
```

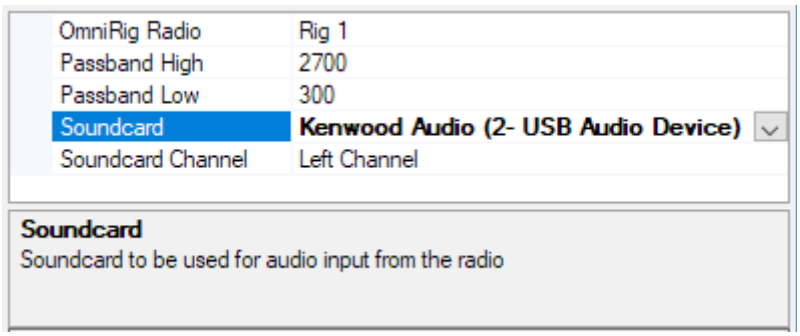

Fig.4. The settings of the Conventional Radio plugin in the Settings dialog.

Implementing the IVisualPlugin interface

By implementing the `IVisualPlugin` interface, a plugin can add its own panels to the program. The user opens the plugin panels using the View section in the main menu. The panels are created as floating windows, but the user can dock them to the main window of the program.

```
public interface IVisualPlugin
{
  bool CanCreatePanel { get; }
  UserControl CreatePanel();
  void DestroyPanel(UserControl panel);
}
```

- **CanCreatePanel**: true if the plugin can create a new panel, false otherwise. The DisplayPanelDemo plugin described in the Sample Plugins section can open an unlimited number of panels, so its CanCreatePanel is always true. Most other plugins allow creation of only one panel, and their CanCreatePanel property is true until the first panel is created. Some plugins may need to have a varying number of panels. For example, the waterfall display plugin may allow two panels if the radio has a sub-receiver, an one panel if it doesn't;
- **CreatePanel**: this method must return the plugin's panel, a descendant of the `UserControl`

class;

- **DestroyPanel**: this method is called when the user closes the plugin's panel. The plugins that allow only one panel should change their CanCreatePanel property back to true when the previously created panel is closed.

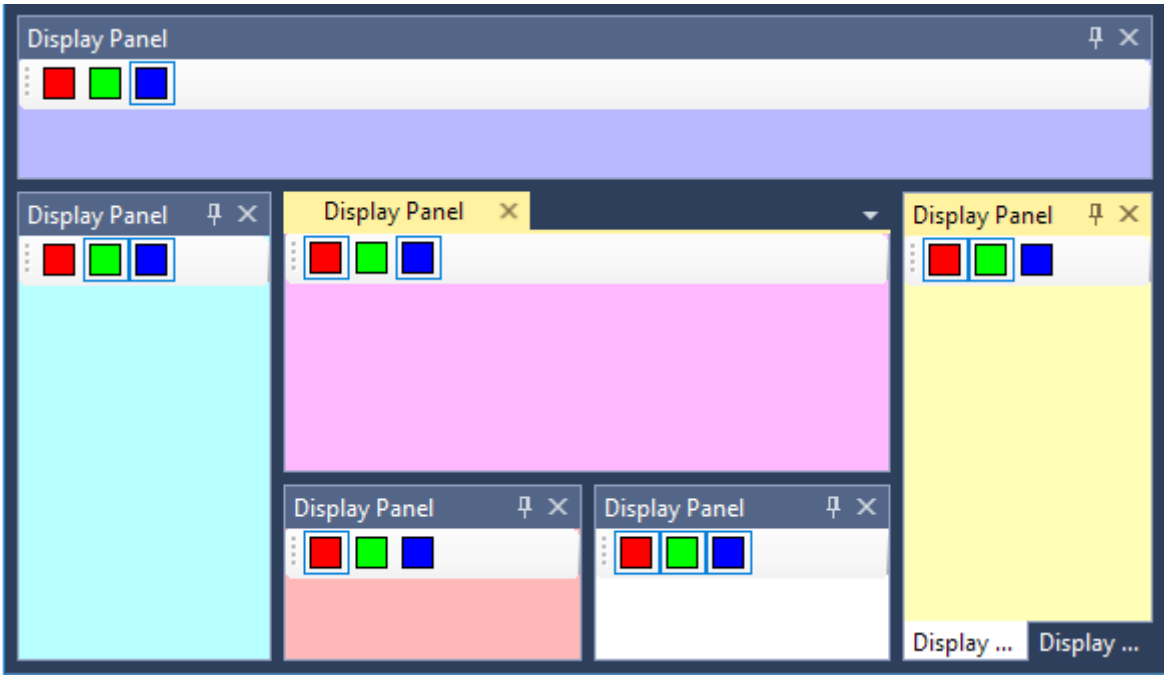

Fig. 5. Multiple panels of the DisplayPanelDemo plugin docked to the main window.

DSP Pipeline

The DSP Pipeline unit organizes the plugins that perform real-time processing of the I/Q and audio data in a multi-stage pipeline with the following stages:

- **data input**: performed by one of the plugins that implement the `ISignalSource` interface. Typically such plugins obtain I/Q data from an SDR radio, but some may also read it from a WAV file or receive from a VoIP system. The user selects which input will be used in the drop-down menu of the Start button on the toolbar. If a signal input plugin has a status item, it is visible on the status bar only when the plugin is selected;
- **I/Q processing**: zero or more plugins that implement the `IIQProcessor` interface. The user can change the order in which the processors are called by reordering them in the Plugin Settings dialog using drag-and-drop;
- **demodulation**: one of the plugins that implement the `IDemodulator` interface. The user selects which demodulator is used in the Mode drop-down menu on the toolbar;
- **audio processing**: the plugins that implement the `IAudioProcessor` interface. Similar to the I/Q processors, but process audio samples instead of I/Q data. An example of such plugin is Volume Control;
- **audio output**: one of the plugins that implement the `IAudioOutput` interface. The user selects the

output in the drop-down menu of the Audio Output button. Currently only the AudioOutputSoundcard plugin is available, the plugins to send audio to the DAC built into an SDR, or to VoIP, may be developed in the future.

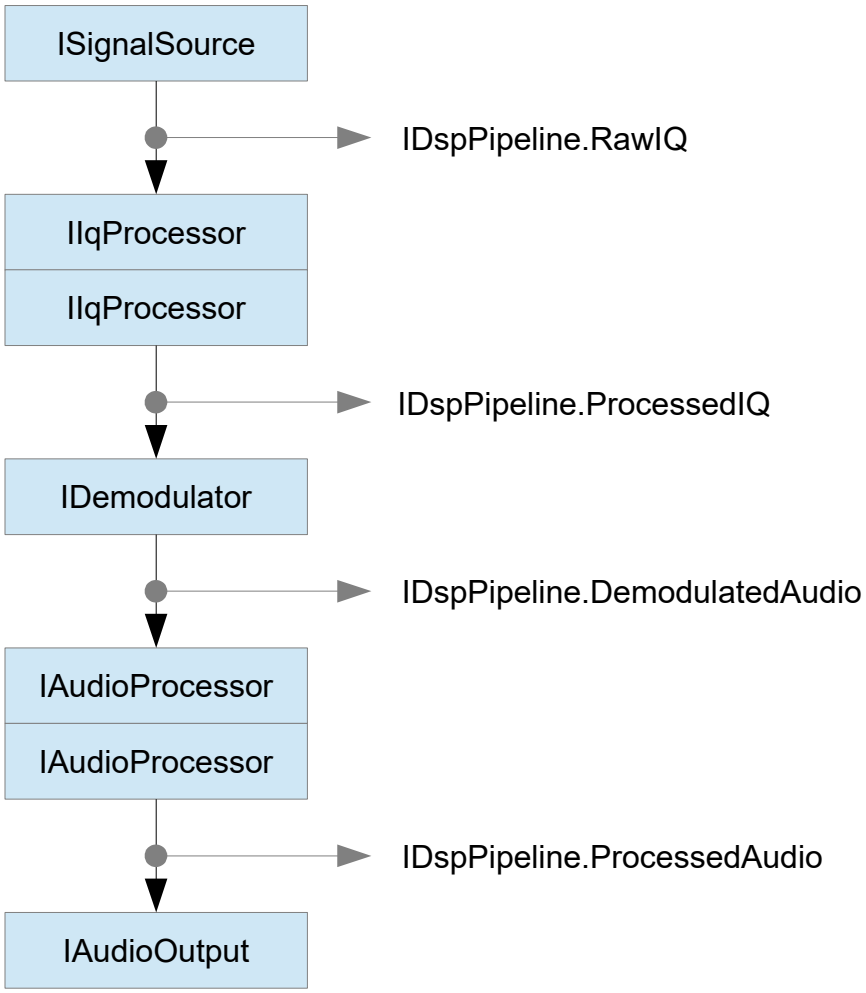

Fig.6. DSP pipeline

The plugins that process I/Q or audio data but do not participate in the DSP pipeline, such as band scopes, waterfall displays, decoders of the digital modes, skimmers, etc., can use the `IDspPipeline` interface exported by the host application to tap the pipeline at different points.

The `IDspPipeline` interface may be imported using the MEF decorator:

```
[Import(typeof(IDspPipeline))]
IDspPipeline pipeline;
```

or obtained using an importing constructor in the plugin class:

```
public class Waterfall : IPlugin, IVisualPlugin
{
  private IDspPipeline pipeline;

  [ImportingConstructor]
  Waterfall([Import(typeof(IDspPipeline))] IDspPipeline pipeline)
  {
    this. pipeline = pipeline;
  }

 ...

}
```

See the MEF documentation [1] for details.

Not all taps are available in all configurations. For example, if signal source is the audio output from a conventional radio, then only the DemodulatedAudio and ProcessedAudio taps are available.

All pipeline taps provide information about the signal format in the Format property of type `SignalFormat`:

```
public class SignalFormat
{
  public int SamplingRate { get; set; }
  public int PassbandLow { get; set; }
  public int PassbandHigh { get; set; }
  public bool IsMono { get; set; }
  public bool IsSync { get; set; }
  public int Channels { get; set; }
}
```

SamplingRate is the sampling rate of the data stream.

IsMono is true for audio signals, false for I/Q signals.

Channels is the number of real-valued audio signals or complex-valued I/Q signals in the data stream.

IsSync: true if the signals in the channels are sampled on the same RF frequency and phase-synchronized, false if sampled on the intependent RF frequencies.

PassbandLow and **PassbandHigh** define the useful part of the sampled range of frequencies. Examples:

- SamplingRate = 48000, IsMono = true, PassbandLow = 300, PassbandHigh = 2700;
- SamplingRate = 48000, IsMono = false, PassbandLow = -22000, PassbandHigh = 22000;

Sample Plugins

Clock Demo

Display Panel Demo

Conventional Radio

Volume Control

Bibliography

1: MSDN, Managed Extensibility Framework (MEF), 2017, https://docs.microsoft.com/en-us/dotnet/framework/mef/
2: Mark Rideout, Getting the Most Out of the .NET Framework PropertyGrid Control, , https://msdn.microsoft.com/en-us/library/aa302326.aspx
3: Witnes, Using PropertyGrid, , https://www.codeproject.com/Articles/22717/Using-PropertyGrid