How CAT Works

The purpose of CAT is to provide an interface between the FlexRadio 6000 series and legacy third-party programs. It is a standalone program that utilizes the FlexLib API to communicate with the radio. CAT incorporates a third-party virtual serial port server which is distributed with the CAT application.

Quick Overview

CAT consists of two main sections: The CAT engine where command classes are created and a serial port/data caching section where serial ports are created and incoming/outgoing data is managed. The user interface consists of a tabbed form that presents information and accepts configuration changes from the user.

At startup, CAT creates a reference to FlexLab.API and waits for a radio to be detected on the network. Once a radio is found, CAT checks its persisted data for that radio serial number and restores any pre-existing serial port configuration. If no data is found for that serial number, CAT creates an initial port.

When data is detected by the serial port received data event, the serial port module parses the data (looking for the semicolon terminator) and sends the result to the data queue. The queue calls the CAT factory and asks for an instance of a CAT Command Class matching the prefix (FA, MD, ZZPF, etc.) of the data. If the prefix is valid, a CAT Command Class of the specified type is created. Specific parameters for each supported CAT command are stored in an external XML file.

The CAT Command Class determines the command type (get or set) of the data and sends a token representing the FlexLib API command to be used to the CAT State Manager. The state manager does a lookup of the token and either gets or sets the appropriate radio property via the API. The results of a get command are returned to the CAT Command Class where it is formatted to the standard CAT format for that particular CAT command. The result is sent to the serial port for transmission.

The Devil Is In The Details

Initialization

Main.cs contains a reference to FlexLib and, in the constructor for the form, subscribes to the API.Radio_Added and API.Radio_Removed events. When the Radio_Added event fires (i.e., a radio has been detected on the network) a subscription to property change event of that radio is created and Main.UpdateRadioList() is called.

UpdateRadioList() counts the number of radios present on the network and, if there is only one, selects that radio. If more than one, it gives the user the opportunity to select which radio to use. The action of selecting the radio, either by the user or automatically, causes the listbox lstAvailableRadios selected item changed event to fire which calls the ConnectRadio(Radio radio) method. (Developer's note: this code needs to be rewritten to work with more than one radio).

ConnectRadio subscribes to SliceAdded and SliceRemoved events for the selected radio and then connects to the radio. ConnectRadio then calls GetAssignedPort which recovers the previously assigned serial port (or creates a new one if no persistence data is available) and two other methods to enumerate all ports on the operating system for future reference when creating new ports.

Once all the above is completed, CAT is ready to accept data coming in on the serial ports.

Serial Ports and Data Flow

CAT serial ports default to 9600 baud, 8 data bits, 1 stop bit, no parity, no flow control. This is accomplished via the DefaultSerialComSpec class. There is an additional class, SerialComSpec, available should custom port parameters be needed in the future. This class is currently not used.

Serial ports derive from the CatSerialPort class. This class contains the code to set the basic port parameters, monitor port pin changes (for bit bang PTT), and to monitor the SerialDataReceived event.

When a port is created (SerialPortManager.CreateCATPort), a corresponding data queue is created and assigned to that serial port. This permits multiple, simultaneous serial ports. The serial port SerialDataReceived event handler parses the incoming data and sends the result to the assigned queue.

```
StringBuilder comBuffer = new StringBuilder();
Regex regex = new Regex(".*?;");

void SerialReceivedData(object source, SerialDataReceivedEventArgs e)
{
        if (!can_ptt)
        {
                comBuffer.Append(this_port.ReadExisting());
                for (Match m = regex.Match(comBuffer.ToString());
                        m.Success; m = m.NextMatch())
                {
                try
                {
                        _cache.AddCmd(m.Value.TrimEnd(';'));
                }
                catch (Exception ex)
                {
                        _log.Error(ex.Message);
                }
                finally
                {
                        comBuffer = comBuffer.Replace(m.Value, "", 0, m.Length);
                }
        }
}
```

The regular expression ".*?;" simply looks for a string on any length that is terminated with a semicolon. Once this match is achieved, that string is added to the queue and removed from the data buffer.

The queue (CatCache.cs) acts as a FIFO buffer for CAT commands. When a command is dequeued (CatCache.ProcessCommand(string cmd) it calls for the CatCmdProcessor to execute the command:

```
private void ProcessCommand(string cmd)
{
        try
        {
                object answer = _om.CatCmdProcessor.ExecuteCommand(cmd);
                this._port.Write(answer.ToString());
        }
        catch (Exception ex)
        {
                _log.Error(ex.Message);
        }
}
```

When an answer is received from the CatCmdProcessor, that result is sent to the serial port and transmitted back to the program originally initiating the CAT request.

Serial Port Creation

Serial ports are created in SerialPortManager.cs. SerialPortManager.EnumerateWindowsPorts() creates an array of any ports currently detectable on the operating system. This array is used to determine what the next available pair numbers will be. If no virtual ports are detected, the default port starting number is COM4.

The virtual serial ports are created using the FabulaTech SDK under an OEM license. The FabulaTech API contains a complete set of commands for creation, destruction, and examination of the virtual ports created. "Ports" is really a misnomer, what is really created is a pair of virtual null-modem cables. The CAT application and the third-party application are responsible for creating the actual serial ports. In the case of CAT, the .NET SerialPort class is used.

CAT can create three types of ports: a "dedicated" port that has one end anchored to a .NET port in the CAT application, the other end is the low-numbered end of a virtual cable pair; a "shared" port which is simply as virtual cable pair that can be used by other apps like DDUtil; and a "PTT" port that is used for bit bang PTT. See SerialPortManager.cs for the code.

SerialPortManager also maintains the external Radios.xml file that contains the port persistence information for each radio serial number.

CAT Factory

CAT utilizes Spring.NET as its Dependency Injection and AOP platform. Spring has an excellent XMLObjectFactory class that generates classes from XML data. CAT class data for the factory is stored in CatClasses.xml. Here is a snippet from that file with an explanation of the elements:

```
<object id="ZZFA" type="Cat.Cat.CatBase, Cat" singleton="true" >
  <property name="API" value="Freq0"/>
  <property name="Prefix" value="ZZFA"/>
  <property name="SetLength" value="11"/>
  <property name="GetLength" value="0"/>
  <property name="AnswerLength" value="11"/>
  <property name="Range">
    <object type="Cat.Cat.Formatters.ValidRange, Cat">
      <property name="Min" value="10000"/>
      <property name="Max" value="77000000"/>
    </object>
  </property>
  <property name="CatReturnFormatter" ref="addLeadingZeros"/>
  <property name="StateMgr" ref ="CatStateManager" />
</object>
```

This is the widely used "Get or Set the Frequency of VFO A" CAT command. The matching API token is "Freq0", used for lookup in the StateManager. The format of the CAT command is defined in the SetLength, GetLength, and AnswerLength properties. There are two formatters defined, one to set the valid range of the command and the other to add leading zeros to the command to match the AnswerLength property value.

There are additional tags used in other commands. Whether or not the command contains a polarity sign can be detected and many other formatters are available to handle the variations in command structure between Kenwood commands and FlexRadio commands.

This xml file is read into the XMLObjectFactory at Factory.cs and an instance of the CAT factory is created:

```
internal Factory()
{
        IResource input = new FileSystemResource("Cfg/Libs/CatClasses.xml");
        _factory = new XmlObjectFactory(input);
 }
```

Thereafter, when a call is made to Factory.GetCatCommand(string cmd), the Spring factory instantiates a CatCommand class of the requested type and, conditionally adds two proxies to the command for use by the AOP interceptors.

More information on Spring can be found at their website http://springframework.net/index.html and I found *Dependency Injection in .NET* by Mark Seeman (Manning, 2012, ISBN 9781935182504) to be an excellent source on DI/AOP.

State Management

As mentioned above, when a CAT Command class is created, its API property is populated with a token. Tokenization helps associate CAT commands with slices. For instance, the slice property for frequency is "Freq". It is up to the user to decide which slice frequency is required. By using the tokens Freq0 and Freq1, the FA/ZZFA and FB/ZZFB commands can be processed using the correct slice frequency. This same logic applies to several other commands.

All radio state is obtained at the time the CAT request is received. The only state stored locally is that of the IF/ZZIF commands as they must be synthesized from multiple radio properties. A typical flow for a CAT command is:

> The CATCommand class (see CatBase.cs) receives the command from the data queue and decides (through the properties of the class) whether or not it is a get or set command and which formatters to use.
>
> If the command passes the validation test it is either sent to CatStateManager.Get or CatStateManager.Set methods.
>
> The CatStateManager method does a lookup of the API token and sends a radio property change if it is a set command or returns the result of a get command to the CATCommand class.
>
> The CATCommand class applies the formatters specified in its properties and returns the final result to the data queue where it is written out the serial port.

Here's an example of how the CatStateManager handles the retrieval of a CAT get command where the command was "FA" and the token was "Freq0":

```
case "Freq0":
        if (slice0 != null)
        {
        if (slice0.Freq.ToString().Contains(separator))
                parts = slice0.Freq.ToString().Split(Convert.ToChar(separator));
        else
                parts = new string[] { slice0.Freq.ToString(), "000000" };

        if (parts[1].Length > 6)
                parts[1] = parts[1].Substring(0, 6);

        result = parts[0] + parts[1].PadRight(6, '0');
        }
        break;
case …
```

The variable "slice0" is a local reference to the slice detected when CatStateManager.SliceAdded event handler detected a SliceAdded event.

Here is the same thing for a set command:

```
case "Freq0":
        nfreq = Double.Parse(Value) / 10e5;
        if(slice0 != null)
                slice0.Freq = nfreq;
        break;
case …
```

"Value" is the 11 character string suffix of the FA or ZZFA command ("00014100000").

Some CAT commands require a little more formatting. The MD/ZZMD demodulator mode commands , for instance, require totally different formats. In this case, formatting is handled by the ConvertDemodMode formatter specified in the XML configuration file for the command and executed in the CATCommand class.

Object Manager

The Object Manager (ObjectManager.cs) provides dependency injection for the major modules (State Manager, Data Manager, Cat Command Processor, etc.) in the CAT application. A single instance of each module is created as a property of ObjectManager at application start.

Each application module that needs access to one of the classes in the Object Manager can simply get an instance of Object Manager and have access to any class that is one of its properties. Example:

Main.cs needs access to SerialPortManager.EnumerateWindowsPorts(). Object manager is set as a field in Main:

```
ObjectManager _om;
```

and instantiated in the constructor:

```
_om = ObjectManager.Instance;
```

```
and used in the constructor:
```

```
_om.SerialPortManager.EnumerateWindowsPorts();
```

If, for some reason, I needed to reference the CatStateManager, all I would have to do in Main would be to call _om.CatStatemanager.MethodIWantToUse().

Summary

The CAT for SmartSDR application is difficult to analyze due to its event driven nature. Some of the code in the application has been superseded by different methods but the old code left in place waiting for a code cleanup or possible future use. I hope this short description helps you wade through the code.