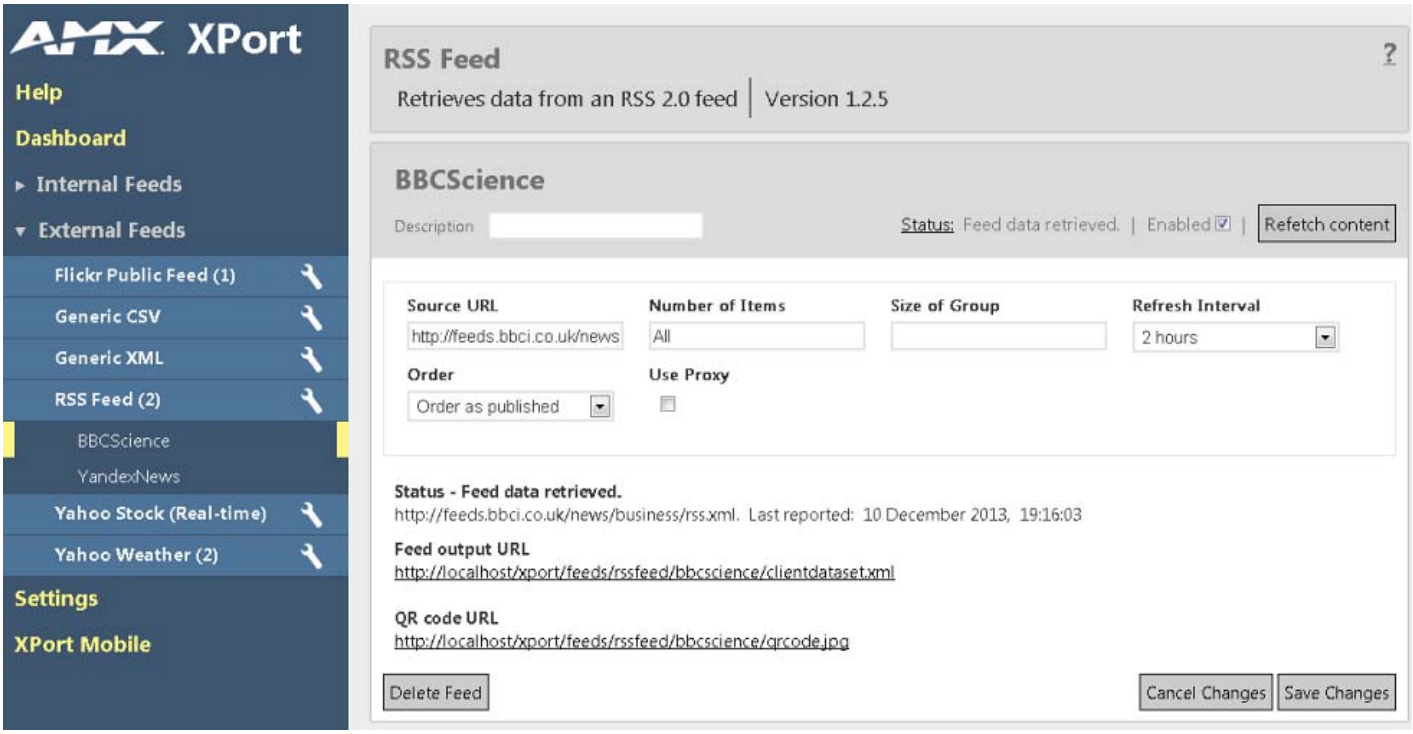
PROGRAMMER'S GUIDE

XPORT DATA INTEGRATION SOFTWARE

XPORT-SERV-100



AV FOR AN IT WORLD

IMPORTANT SAFETY INSTRUCTIONS

1. READ these instructions.
2. KEEP these instructions.
3. HEED all warnings.
4. FOLLOW all instructions.
5. DO NOT use this apparatus near water.
6. CLEAN ONLY with dry cloth.
7. DO NOT block any ventilation openings. Install in accordance with the manufacturer's instructions.
8. DO NOT install near any heat sources such as radiators, heat registers, stoves, or other apparatus (including amplifiers) that produce heat.
9. DO NOT defeat the safety purpose of the polarized or grounding type plug. A polarized plug has two blades with one wider than the other. A grounding type plug has two blades and a third grounding prong. The wider blade or the third prong are provided for your safety. If the provided plug does not fit into your outlet, consult an electrician for replacement of the obsolete outlet.
10. PROTECT the power cord from being walked on or pinched, particularly at plugs, convenience receptacles, and the point where they exit from the apparatus.
11. ONLY USE attachments/accessories specified by the manufacturer.

12. USE ONLY with a cart, stand, tripod, bracket, or table specified by the manufacturer, or sold with the apparatus. When a cart is used, use caution when moving the cart/apparatus combination to avoid injury from tip-over.
13. UNPLUG this apparatus during lightning storms or when unused for long periods of time.
14. REFER all servicing to qualified service personnel. Servicing is required when the apparatus has been damaged in any way, such as power-supply cord or plug is damaged, liquid has been spilled or objects have fallen into the apparatus, the apparatus has been exposed to rain or moisture, does not operate normally, or has been dropped.
15. DO NOT expose this apparatus to dripping or splashing and ensure that no objects filled with liquids, such as vases, are placed on the apparatus.
16. To completely disconnect this apparatus from the AC Mains, disconnect the power supply cord plug from the AC receptacle.
17. Where the mains plug or an appliance coupler is used as the disconnect device, the disconnect device shall remain readily operable.
18. DO NOT overload wall outlets or extension cords beyond their rated capacity as this can cause electric shock or fire.

The exclamation point, within an equilateral triangle, is intended to alert the user to the presence of important operating and maintenance (servicing) instructions in the literature accompanying the product.

The lightning flash with arrowhead symbol within an equilateral triangle is intended to alert the user to the presence of uninsulated "dangerous voltage" within the product's enclosure that may be of sufficient magnitude to constitute a risk of electrical shock to persons.

ESD Warning: The icon to the left indicates text regarding potential danger associated with the discharge of static electricity from an outside source (such as human hands) into an integrated circuit, often resulting in damage to the circuit.

WARNING: To reduce the risk of fire or electrical shock, do not expose this apparatus to rain or moisture.

WARNING: No naked flame sources - such as candles - should be placed on the product.

WARNING: Equipment shall be connected to a MAINS socket outlet with a protective earthing connection.

WARNING: To reduce the risk of electric shock, grounding of the center pin of this plug must be maintained.

COPYRIGHT NOTICE

LIABILITY NOTICE

No patent liability is assumed with respect to the use of information contained herein. While every precaution has been taken in the preparation of this publication, AMX assumes no responsibility for error or omissions. No liability is assumed for damages resulting from the use of the information contained herein. Further, this publication and features described herein are subject to change without notice.

AMX WARRANTY AND RETURN POLICY

The AMX Warranty and Return Policy and related documents can be viewed/downloaded at www.amx.com.

Table of Contents

AMX XPort Programmer's Guide

Overview

This document will show you what is needed to implement a simple Feed Handler by examining the structure and behavior of an RSS Feed Handler. Please note however, that this document does not explain how to program or provide a detailed explanation of Microsoft's Visual Studio. The document is targeted at those who are familiar with both but does not expect the reader to be an expert in either. Examples are provided in C# but this is an easy language to read so should not present a problem to anyone who has a working knowledge of programming.

NOTE: *Any version of Visual Studio 2012 - including the free "Visual Studio Express 2012 for Windows Desktop" - can be used to develop a Feed Handler. Instructions in this document assume the Express version.*

The RSS feed handler source code is zipped up inside the XPort zip file downloaded from the *AMX XPort* product page at www.amx.com.

NOTE: *To become familiar with editing Feed Handlers in XPort, experiment with making small changes to an existing Feed Handler, then immediately checking your changes to see the effects.*

XPORT-SERV-100 XPort Data Integration Software

XPORT-SERV-100 XPort Data Integration Software (FG1201-10) is a web-based application that makes it easy to configure data feeds for use by XPert and XPress. This software is simply referred to as "XPort" in this manual.

XPort is a framework; it does nothing useful until Feed Handler DLLs are installed (note that XPort is provided with a number of pre-installed Feed Handlers). Logically you consider XPort as being constructed as shown in FIG. 1:

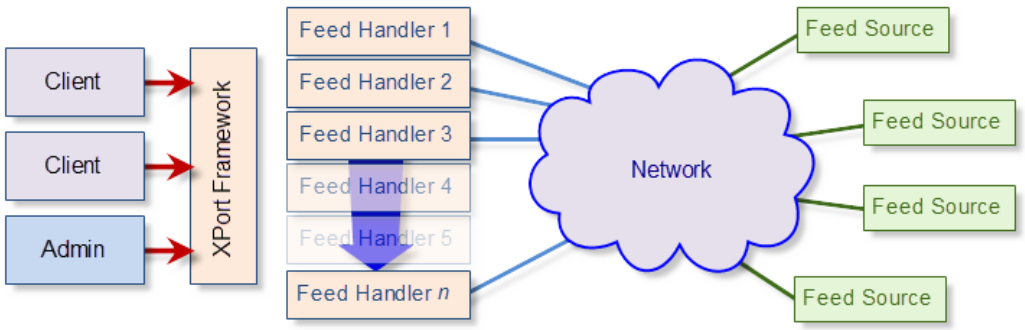


FIG. 1 XPort Framework

NOTE: *In Fig. 1, each "Client" represents a consumer of the data that XPort produces (e.g. XPert), and the "Admin" represents a user who configures the feeds.*

The XPort Framework can be regarded as being split into two layers:

1. The **Web Interface** layer - which provides (a) a user interface to allow administrators to configure the feeds and monitor status and (b) access mechanisms used by client machines to retrieve the XML files generated by the feeds.
2. The **Service** layer - which sits between the Web Interface Layer and the Feed Handlers and the programming interface for the Feed Handlers. The programming interface provides features to support:
   a. The creation of a user interface for configuring the Feed Handler.
   b. The production of correctly formatted XML files.
   c. The independent running of Feed Handlers; Feed Handlers will not interfere with each other even under error conditions.

NOTE: *When you install XPort you will find that these two layers correspond to the two folders that are created in your chosen target folder; two folders called - obviously enough - Web and Service. Throughout this document, the installed Service folder is known as the "XPort Service Folder".*

Getting Started

Before you write any code, you should understand how to build a Feed Handler.

Prerequisites

1. Microsoft Visual Studio Express 2012 for Windows Desktop (or better).

   Express 2012 is an ideal tool because it is free and because it has the capability to debug by attaching to a process (see *Testing and Debugging* on page 16). Earlier versions of Visual Studio Express lack this all important ability but paid versions of Visual Studio have had this capability for a long time. If you have an install of Visual Studio that supports .NET 4 and the ability to attach to a process you should have no problem creating your own Feed Handlers.

2. XPort must be installed on the same machine as Visual Studio. You do not have to license the installation of XPort. An unlicensed installation of XPort provides all the necessary functionality for development.

Building a Feed Handler

1. Open Visual Studio and create a new C# Class Library project (FIG. 2):

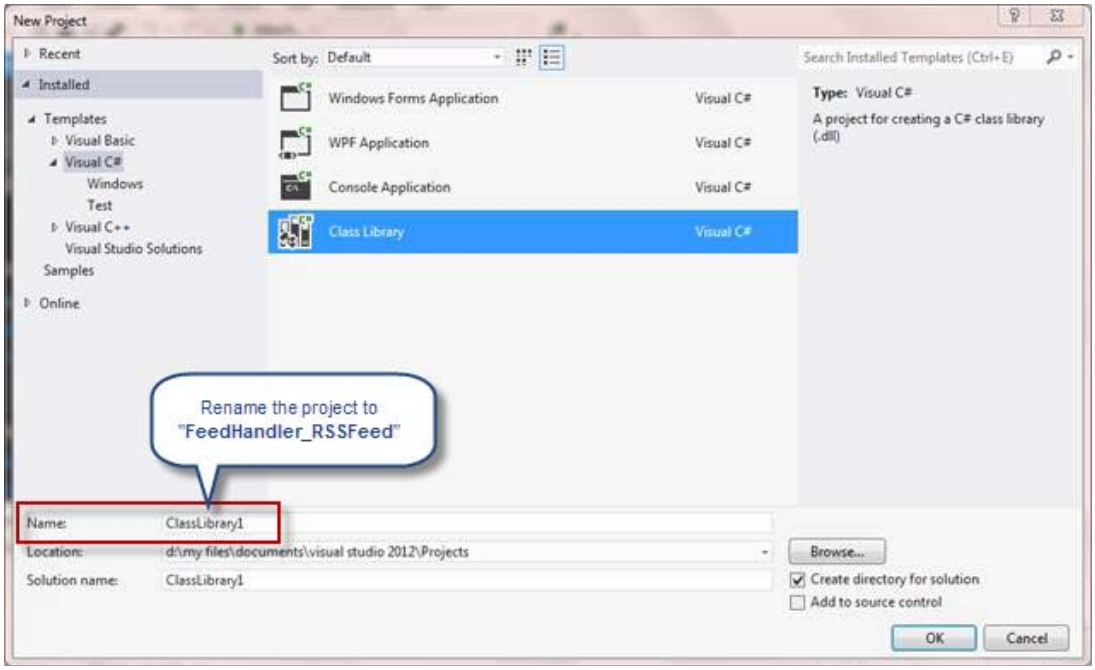


FIG. 2 Create new C# Class Library Project

2. Call it **FeedHandler_RSSFeed** rather than the default *ClassLibrary1*, and click **OK**.

NOTE: *This setting governs the name of the DLL that is created by the project, and Feed Handler file names must start with the text "FeedHandler_"*

3. Download the source of the RSS Feed from the *AMX XPort* product page at www.amx.com.

4. Locate the two source files for the RSS feed (**RSSFeed.cs** and **XMLReader.cs**) and save them to the directory created for your new project by Visual Studio (FIG. 3):

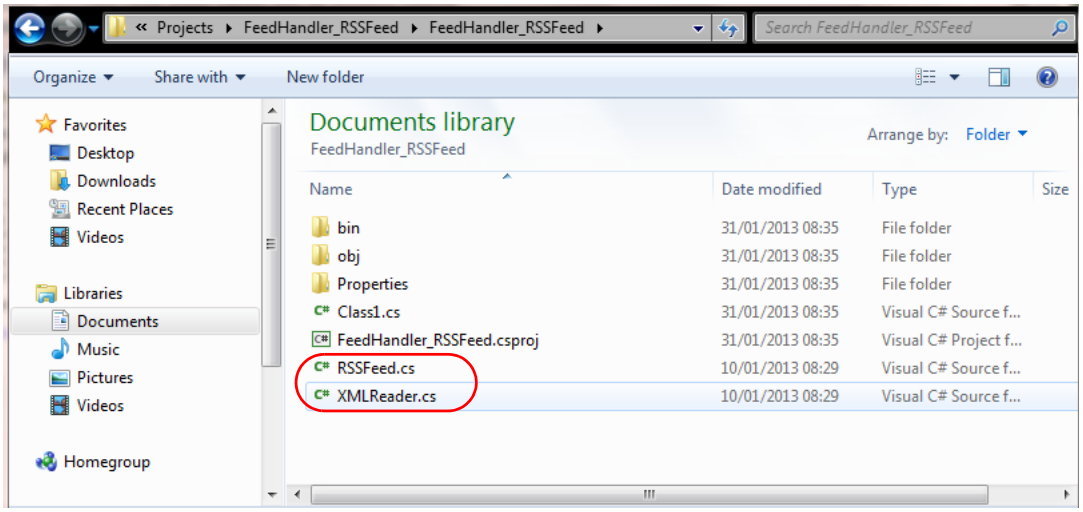


FIG. 3 RSS feed - Source Files

5. In the Visual Studio UI, right click on **FeedHandler_RSSFeed** in the Solution Explorer and select **Add > Existing Item...** to add *RSSFeed.cs* and *XMLReader.cs* to the project (FIG. 4):

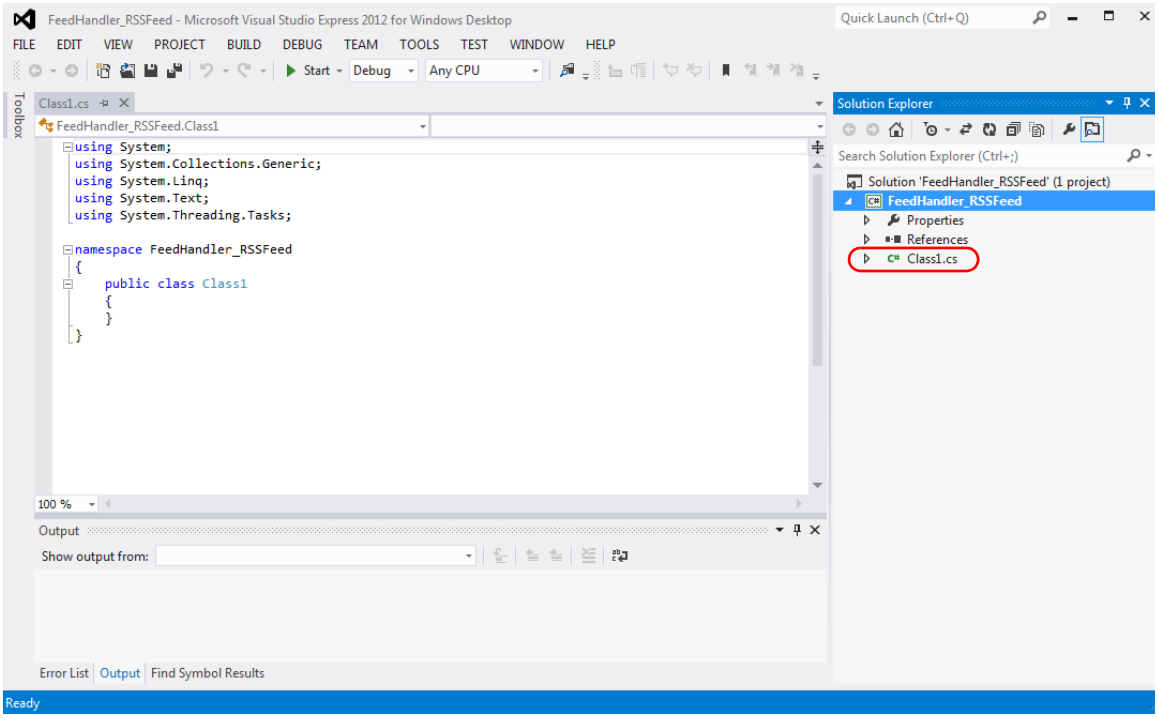


FIG. 4 Visual Studio UI Showing the Solution Explorer

6. Right-click on **Class1.cs** in the Solution Explorer and select **Delete**.
7. If you try to compile now you will find you have lots of unsatisfied references. To fix this you must right-click on **References**, select **Add Reference...** then navigate to the *XPort Service* Folder where you must select **XPortFeedHandler.dll**.

If you try to build your new project now, the build will complete successfully but there are other things that need to be done to ensure that the resulting Class Library runs:

On the PROJECT menu select **FeedHandler_RSSFeed Properties...**

1. Under *Application* select **.NET Framework 4** as the target Framework
2. For completeness, you should also change the default namespace to **Amx.InspiredSignage.XPort.FeedHandlers** (you will only notice the benefit of this when you create your own classes).

   The *Assembly Name* is the same as the public class provided in RSSFeed.cs. If this is not the case the feed will not run.

3. Under *Build* select **Any CPU** as the platform target.

You can modify the project that you have just created to produce any feed that you wish. In the following sections, we will look at the implementation of the RSS Feed Handler in more detail.

The Anatomy of a Feed Handler

Overview

As you will recall from FIG. 1 on page 1, a Feed Handler connects to the XPort framework at one "end" and the Network at the other.

The RSS Feed Handler that you imported into Visual Studio in the previous section reflects this duality; *XMLReader.cs* reads data from the data source (on the Intranet or Internet) and parses it, while RSSFeed.cs takes the parsed data and pushes it into the framework, so that the clients can collect the data in the format they require.

You don't have to structure your Feed Handlers this way but it keeps the implementation clean and makes the code easier to understand. The following two sections look at the two areas in more detail.

The code of the RSS Feed Handler is used as a case study in the following sections because it is a relatively simple Feed Handler and because the source code is freely available from www.amx.com . You do not need to refer to the source code to understand the following because code extracts are included to highlight the core functionality but you should note that:

- The code extracts may not be from the latest version of the code and
- The extracts do not paint the whole picture. The extracts are chosen to demonstrate only the essential functionality of a Feed Handler.

It is recommended that you download the source code, install Visual Studio or Notepad++ and review the source code while reading this document. This document does not explain every method but shows you how the methods are used. For more information about the methods provided by the XPort Framework, refer to the *Working With the Framework (RSSFeed.cs)* section on page 5.

Working With the Framework (RSSFeed.cs)

Registration

The XPort Framework determines the name of the class that implements the interfaces that it requires from the name of the Feed Handler DLL itself. Thus, FeedHandler_RSSFeed.dll contains the following class declaration:

```
public class FeedHandler_RSSFeed : IFeedHandlerRegistrant, IFeedHandler
```

Not only do these two names have to match, the name must also start with the text "FeedHandler_" or the Framework will not identify the DLL as a Feed Handler.

The XPort framework calls the Feed Handler to retrieve (a) its identity, version number and other information and (b) a list of the fields that must appear on the Web User Interface to allow a feed to be configured.

The `IFeedHandlerRegistrant` interface requires two methods to support this:

- Info()
- RegisterSettings(IFeedHandlerRegistrar.Registrar)

These methods are called when the Feed Handler is loaded. `Info()` is called first and the contents of the returned FHInfo class determine whether `RegisterSettings()` needs to be called.

If the "Identity" returned by `Info()` has been registered previously and the returned "Version" is the same as previously registered, `RegisterSettings()` is not called. However, this approach can present some difficulties for developers who would have to change the version number of a Feed Handler every time the User Interface changed so it is possible to force `RegisterSettings()` to be called by selecting the **Developer Mode** option on the *Settings* page of the XPort User Interface (FIG. 5):

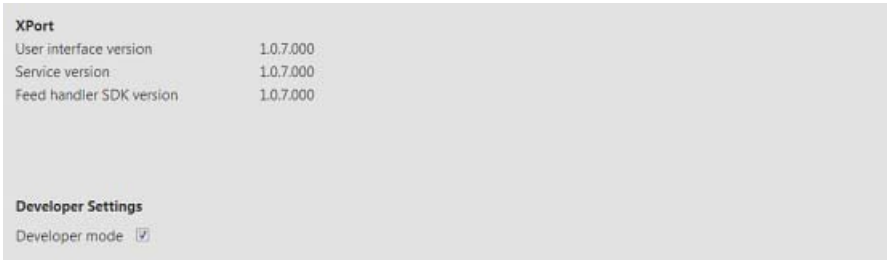


FIG. 5 XPort Admin - Settings Page

The following code performs registration for the RSS Feed:

```
private static readonly FHInfo Info = new FHInfo()
{
    Identity = "FH_RSSFeed",
    Name = "RSS Feed",
    Description = "Retrieves data from an RSS 2.0 feed",
   HelpText = "This Feed Handler will not follow any HTML links except to retrieve images from
XML
    nodes of type" + "media:thumbnail. All HTML tags found in text fields are ignored.",
    Version = "1.0.24"
};

FHInfo IFeedHandlerRegistrant.Info
{
    get { return Info; }
    set { }
}
 ////////////////////////////////////////////////////////// Define Settings
// Define the feed handler setting that will be seen on the GUI if any
private readonly FHS_TextBox SettingSourceURL = new FHS_TextBox()
{
    Identity = "sourceurl",
    Label = "Source URL",
    Tip = "You must enter the URL of the data source EG: http://rss.myfeed.com"
};

private const string DefaultNumItemsString = "All";
private readonly FHS_TextBox SettingNumItems = new FHS_TextBox
{
    Identity = "numitems",
    Label = "Number of Items",
    Tip = "Enter a positive integer to retrieve a specific number of items from the data
source." +
          "Enter the word \"All\" to retrieve every item.",
    DefaultValue = DefaultNumItemsString
};

private readonly FHS_TextBox SettingGroupSize = new FHS_TextBox
{
    Identity = "groupsize",
    Label = "Size of Group",
    Tip = "The number of items that will be grouped together into a single output record. The
           default is 1."
};
private const string choiceAuto = "Auto";
private readonly FHS_DropList SettingRefresh = new FHS_DropList

{
    Identity = "refresh",
    Label = "Refresh Interval",
    Tip = "Choose the time interval between each fetch of the feed. " +
          "Select \"Auto\" to automatically calculate the optimal interval.",
    DefaultValue = choiceAuto,
    Values = new List<String>()

 selected.",
    DefaultValue = choiceAsPublished,
    Values = new List<String>
    {
        choiceChronological, choiceReversed, choiceAsPublished, choiceRandom
    }
};
```

(cont. next page)

```
   {
        choiceAuto, "5 minutes", "15 minutes", "30 minutes", "1 hour", "2 hours", "5 hours",
"12 hours",
       "1 day"
    }
};

private const string choiceChronological = "Ordered oldest to newest";
private const string choiceReversed = "Ordered newest to oldest";
private const string choiceAsPublished = "Order as published";
private const string choiceRandom = "Random order";
private FHS_DropList SettingOrder = new FHS_DropList
{
    Identity = "order",
    Label = "Order",
    Tip = "Choose the sort order of the items." +
          "The items will be sorted before the required number of items or groups are
void IFeedHandlerRegistrant.RegisterSettings(IFeedHandlerRegistrar fhregistrar)
{
    const string DefaultGroup = "";
    fhregistrar.RegisterSetting(DefaultGroup, SettingSourceURL);
    fhregistrar.RegisterSetting(DefaultGroup, SettingNumItems);
    fhregistrar.RegisterSetting(DefaultGroup, SettingGroupSize);
    fhregistrar.RegisterSetting(DefaultGroup, SettingRefresh);
    fhregistrar.RegisterSetting(DefaultGroup, SettingOrder);
}
```

In the above code extract you will see that the RSS Feed Handler uses only two types of input control:

- `FHS_TextBox` - a text box that allows the user to enter any text that they like; there is no built-in input validation.
- `FHS_DropList` - the user can choose any one item from the provided list of string values; the items are displayed in the order they are provided. Note that there is no index property for the values on the list; items on the list are identified by name only.

You can see that both of these input controls share the following values that can be provided to the constructor or by changing the properties of the control:

- `Identity` - a string used to identify the control within the Feed Handler
- `Label` - a string that appears on the User Interface beside the control
- `Tip` - a string tooltip that appears on the User Interface when the mouse is hovered over the control
- `DefaultValue` - the initial value of the control which can be changed by the user

These two fields are probably the most useful to the writer of a Feed Handler but they are not the only types of field that are available. For more information about the input controls - and all the other methods support by the Feed Handler framework - look at the online documentation which can be found here:

http://www.amx.com/assets/documentation/XPORT-SERV-100/index.html

NOTE: *The online documentation splits the framework's methods into two namespaces:*

- **AMX.InspiredSignage.XPort.FeedHandlers** which provides the Interfaces and input controls and
- **AMX.InspiredSignage.XPort.FeedHandlers.Ouput** which provides the output methods

Both are covered in the online documentation.

NOTE: *A "namespace" provides a way to group types. The full name of a type includes the name of the namespace that includes it. This helps ensure that type names are unique.*

Turning attention to the `RegisterSettings()` method that appears at the end of the code fragment above, note that each calls to `RegisterSetting()` has DefaultGroup as the first parameter and DefaultGroup is set to the empty string. The first parameter to `RegisterSetting()` defines the group to which the registered UI control belongs. Groups appear on the User Interface as a bounded, named area that contains the controls that are registered within it. If the group name is the empty string - as is the case here - then the name is not output and takes no space on the UI. See the Yahoo Stock Feed Handler for an example of code that does display a group name.

Start up and Tear Down

The `IFeedHandler` interface requires three methods to start and stop different feeds. Unless incredibly specialized, one Feed Handler will be used to retrieve multiple feeds. The following methods are used to start and stop the different instances:

- **`Initialize(string sandboxpath):`** `Initialize()` is called after the "Create Feed" button has been pressed on the Web User Interface but before the User Interface has been created for the new feed. The string parameter "sandboxpath" identifies a directory in which files can be created as required. Experience suggests that you will not need to use this method (although it must be provided).

- **`OnStart()`:** `OnStart()` is called after the User Interface has been created for the new feed so the values provided by the user can be interrogated , validated and recorded for use when the fetch timer expires and the `OnFetch()` method is called (see See “Fetching Data” on page 9.). `OnStart()` will normally start the fetch timer by calling `SetFetchInterval()`.
- **`OnStop()`**: `OnStop()` should clear down any persistent data created during execution of the feed. Again, experience suggests that you are unlikely to use this method (although, again, it must be provided).

The `Initialize()`, `OnStart()` and `OnStop()` methods from the RSS Feed Handler are shown below. Note the frequent calls to `fhHost.LogEntry_Info()` to copy diagnostic/status information to the Telnet log:

```
bool IFeedHandler.Initialize(string sandboxpath)
{
    m_SandBoxPath = sandboxpath;
    fhHost.LogEntry_Info("Initialise() called.");

    return true; // Init went ok and we are ready to be started
}

bool IFeedHandler.OnStart()
{
    fhHost.LogEntry_Info("OnStart() called.");
    m_NumItems = GetNumItems(SettingNumItems);
    m_GroupSize = GetInt(SettingGroupSize, 1);
    m_FetchInterval = SettingRefresh.CurrentValue.ToString();
    m_IsAutoFetchInterval = (m_FetchInterval == choiceAuto);
    if (m_IsAutoFetchInterval)
    {
        // Have to set the fetch interval to make the fetch happen
        m_AutoFetchInterval = DefaultRefreshInterval;
        fhHost.SetFetchInterval(m_AutoFetchInterval);
    }

else
    {
        fhHost.SetFetchInterval(ToSeconds(m_FetchInterval));
    }

    if (SettingSourceURL.CurrentValue.ToString().Trim() == string.Empty)
    {
        fhHost.SetStatusInvalidConfig("You must supply a URL.", "The feed will not be
fetched.");
        fhHost.LogEntry_Info("No URL has been provided.");
    }
    fhHost.LogEntry_Info(SettingSourceURL.Identity + ": " + SettingSourceURL.CurrentValue);
    fhHost.LogEntry_Info(SettingNumItems.Identity + ": " + m_NumItems);
    fhHost.LogEntry_Info(SettingGroupSize.Identity + ": " + m_GroupSize);
    fhHost.LogEntry_Info(SettingRefresh.Identity + ": " + m_FetchInterval);
    fhHost.LogEntry_Info(SettingOrder.Identity + ": " + SettingOrder.CurrentValue);

    return true;
    }
    void IFeedHandler.OnStop()
{
    fhHost.LogEntry_Info("OnStop() called.");

    // TODO: Stop/teardown stuff
}
```

The call to `SetFetchInterval()` does two things:

1. It causes `OnFetch()` to be called immediately
2. It starts a timer that will fire every N seconds where N is the parameter supplied to `SetFetchInterval()`. Every time the timer fires, `OnFetch()` is called.

Fetching Data

The `OnFetch()` method is the point where it all happens. Up until now the two ends of our Feed Handler handler – the XPort Framework and the Internet - have been poles apart but this is where they meet. The following code fragment calls into XMLReader.cs to retrieve and parse data from the feed source:

```
var rssFeed = new XFeedReader();

try
{
    fhHost.SetTXActivity();
    rssFeed.Read(SettingSourceURL.CurrentValue);
   fhHost.SetStatusWorking("Feed data retrieved.", SettingSourceURL.CurrentValue.ToString() +
".");
    fhHost.LogEntry_Info("Feed data retrieved.");
    fhHost.SetRXActivity();
}
catch (Exception e)
{
    fhHost.SetStatusProblems("Failed to retrieve the feed data.", e.Message + ".");
    fhHost.LogEntry_Error("Failed to retrieve the feed data: " + e.Message);
    return;
}
```

Notice the try/catch construct. Under normal circumstances only the "try" code will be executed. So let's consider each of the five lines of "try" code individually:

1. **`fhHost.SetTXActivity()`**; Called to update the count of "Requests sent" on the Web UI Dashboard.
2. **`rssFeed.Read(SettingSourceURL.CurrentValue)`**; Called to retrieve the data from the URL provided by `SettingSourceURL.CurrentValue`. This is the only line of code in the "try" section that could raise an exception. If it does, none of the remaining lines will be executed. An exception will be raised if the URL does not point to a valid data source or if there is no connection to the data source.
3. **`fhHost.SetStatusWorking("Feed data retrieved.", SettingSourceURL.CurrentValue.ToString() + ".")`**; If this line of is executed then `rssFeed.Read()` must have succeeded so set the status accordingly. This status will be shown on the Web UI for the feed.
4. **`fhHost.LogEntry_Info("Feed data retrieved.");`** If this line of is executed then `rssFeed.Read()` must have succeeded so output a message to the Telnet log.
5. **`fhHost.SetRXActivity();`** Called to update the count of "Replies received" on the Web UI Dashboard.

If, for any reason, the fetch of the data fails, an exception will be raised, and the execution of the "try" code will stop at line 2 and the "catch" code will be executed. Note that this means the `SetRXActivity()` will not be called which means the number of request sent will be larger than the replies received. The "catch" code does nothing more than report the fetch error to the Web UI - using `SetStatusProblems()` - and the Telnet log – using `LogEntry_Error()`.

NOTE: *In this implementation, only RSSFeed.cs does any error reporting. XMLReader.cs will generate exceptions but they are all trapped in RSSFeed.cs. This is not the only way of handling errors but it is simple, adequate and it keeps the code tidy.*

Once the data has been retrieved from the feed source, it can be output. For the RSS Feed Handler, this is a very simple operation:

```
// Create some new records
var fout = fhOutput.PrepareNew();
fhHost.LogEntry_Info("Output prepared.");

var frec = rssFeed.GetFirst();
for (int count = 1; count <= numItemsOut; count++)
{
     if (frec == null) break;
     fout.AddRecord(frec);
     frec = rssFeed.GetNext();
}
fhOutput.WriteFeed(fout);
```

If you look at the full source code you will see there are over 50 lines of code between reading the data in and writing it out again. All of that intermediate code determines the correct order for the records, the grouping of the records and the number of records that should be output (`numItemsOut` in the extract above). All of that functionality is provided by XMLReader.cs so will be dealt with later.

The extract above is so simple because `PrepareNew()`, `AddRecord()` and `WriteFeed()` are provided by the Framework and `GetFirst()` and `GetNext()` are provided by XMLReader.cs. All this code has to do is cycle through the sorted and grouped input and copy it to the output.

Changing the Configuration

The XPort administrator is able to change the configuration of a feed on the fly. The method `OnConfigChanged()` is called whenever this happens. It is possible to simply read the user-supplied data from the user interface every time `OnFetch()` is called (see earlier) but there are two problems with this approach:

1. `OnFetch()` will not "know" whether the user-supplied data has been changed since it was last entered so will have to perform any validation pr calculation every time it is called. This approach would both clutter `OnFetch()` with unnecessary code and/or reduce the efficiency of the Feed Handler by processing the user-supplied data repeatedly rather than once when it was changed.
2. The fetch interval is usually provided on the user interface and `SetFetchInterval()`, which is used to set the fetch interval within the framework, triggers an immediate fetch with each subsequent fetch being the specified interval apart. Putting a call to `SetFetchInterval` inside `OnFetch()` is like asking the Framework to fetch continuously. The framework doesn't start another fetch when a fetch is in progress but it is more logical to place calls to `SetFetchInterval()`, outside `OnFetch()`.

The RSS Feed Handler code for `OnConfigChanged()` follows. Note as usual, the high number of calls to send information to the Telnet log:

```
void IFeedHandler.OnConfigChanged()
{
    fhHost.LogEntry_Info("OnConfigChanged() called.");

    m_NumItems = GetNumItems(SettingNumItems);
    m_GroupSize = GetInt(SettingGroupSize, 1);
    if (m_FetchInterval != SettingRefresh.CurrentValue)
    {
        // Refresh interval has changed
        m_FetchInterval = SettingRefresh.CurrentValue.ToString();
        m_IsAutoFetchInterval = (m_FetchInterval == choiceAuto);
        if (m_IsAutoFetchInterval)
        {
         //The interval has been changed from a definite value to "Auto"
         //Set the Auto interval to the default value. It will be changed after the first fetch
anyway
           m_AutoFetchInterval = DefaultRefreshInterval;
           fhHost.SetFetchInterval(m_AutoFetchInterval);
        }
        else
        {
            fhHost.SetFetchInterval(ToSeconds(m_FetchInterval));
        }
    }

    if (SettingSourceURL.CurrentValue.ToString().Trim() == string.Empty)
    {
        fhHost.SetStatusInvalidConfig("You must supply a URL.", "The feed will not be
fetched.");
        fhHost.LogEntry_Error("No URL has been provided.");
    }

        // Output the new values
        fhHost.LogEntry_Info(SettingSourceURL.Identity + ": " +
SettingSourceURL.CurrentValue);
        fhHost.LogEntry_Info(SettingNumItems.Identity + ": " + m_NumItems);
        fhHost.LogEntry_Info(SettingGroupSize.Identity + ": " + m_GroupSize);
        fhHost.LogEntry_Info(SettingRefresh.Identity + ": " + m_FetchInterval);
        fhHost.LogEntry_Info(SettingOrder.Identity + ": " + SettingOrder.CurrentValue);
}
```

Note that helper methods are used to retrieve and validate `m_NumItems` and `m_GroupSize`.

RSS feeds contain a value known as TTL which stands for "Time To Live". This value tells the reader how many seconds the data in the feed is valid for. At the end of the time interval specified by TTL, the feed should be refreshed. This is the optimal refresh interval, the data is only refreshed when it needs to be so the option of an "Auto" refresh interval is given to the user. If the user selects the Auto option, `m_IsAutoRefreshInterval` is set and the refresh interval is taken from the feed rather than the UI.

Working with the Data Source (XMLReader.cs)

Retrieving the Feed (Simple Version)

The Read() method is called by RSSFeed.cs to retrieve data from the source identified by *feedUri*. This method may generate an exception which must be handled by the caller. There is no exception handling in XMLReader.cs. This simpler version of the Read() method does not reliably handle multi-byte characters because the StreamReader is not told which encoding to use. See *Retrieving the Feed (Advanced Version)* on page 11 to find out how to handle different encodings reliably."

```
public void Read(string feedUri)
{
    // This will raise an exception if it can't reach the internet
    var request = WebRequest.Create(feedUri);
    var response = request.GetResponse();
    var reader = new StreamReader(response.GetResponseStream());
    var doc = XDocument.Load(reader);
    var feeds = GetChannelQuery(doc);
    var any = feeds.Any();
    if (feeds == null || !any) return;
    m_Feed = feeds.First();
    m_Items = m_Feed.fChannel_Items;
}
```

The logic of this code is simple there are nine lines of code as follows:

1. Create a web request from the supplied feedUri.
2. Get the response from the web request
3. The response should be a file so load it into a StreamReader
4. The file should be an XML file so load the StreamReader into an XDocument
5. Parse the XDocument
6. Set the flag "any" to indicate whether the feed is empty or not
7. If the feed is null or empty there is no more to do so return
8. The feed has content so set `m_Feed` to point to the first channel record (there should only be one)
9. The feed has content so point `m_Items` to the list of items contained by the channel record.

Retrieving the Feed (Advanced Version)

This version of the Read() method was created to allow XML files that are not UTF-8 encoded to be read. The code explicitly handles UTF-16 and standard Windows encodings which should handle most scenarios. The code is clearly much more complicated but, in reality, the only additional functionality is reading the encoding type from the file and creating a new StreamReader which uses the encoding.

```
private static Stream CopyAndClose(Stream inputStream)
{
    // This method reads inputStream into memory in chunks of size readSize.
    const int readSize = 256;
    byte[] buffer = new byte[readSize];
    MemoryStream ms = new MemoryStream();

    // Read the first chunk from inputStream
    int count = inputStream.Read(buffer, 0, readSize);
    while (count > 0)
    {
    // Write the chunk to memory and and read another from the input
        ms.Write(buffer, 0, count);
        count = inputStream.Read(buffer, 0, readSize);
    }
    ms.Position = 0;
    inputStream.Close();
    return ms;
}
```

(cont. next page)

```
 public void Read(string feedUri, string Node)
{
    // This will raise an exception if it can't reach the internet

    var request = WebRequest.Create(feedUri);   // Send the request
    var response = request.GetResponse();       // Read the response
    // Now copy the response to a MemoryStream to allow the stream to be re-read
    var stream = CopyAndClose(response.GetResponseStream());
    var reader = new StreamReader(stream);

   // StreamReader assumes UTF-8 encoding need to check if the XML file is encoded differently
    const string xmlhead = "<?xml version=\"1.0\" encoding=\"";
    const string xmlend = "\"?>";
    const string utf16 = xmlhead + "utf-16" + xmlend;
    const string windows = xmlhead + "windows-";

    // Read the entire stream into a string then reset the stream to the beginning
    var responseString = reader.ReadToEnd();
    stream.Position = 0;

  // Check if the XML file is UTF-16 encoded
    if (responseString.StartsWith(utf16, StringComparison.OrdinalIgnoreCase))
    {
        // Create a new streamReader that knows the encoding and re-read into responseString
        reader = new StreamReader(stream, Encoding.Unicode);
        responseString = reader.ReadToEnd();
    }

   // Check if the XML file is Windows encoded
    if (responseString.StartsWith(windows, StringComparison.OrdinalIgnoreCase))
    {
        // Read the number of the encoding
        var enc = responseString.Substring(windows.Length, 6);
        // Remove any non-numeric other chars at the end of the encoding string
        var number = Regex.Replace(enc, "[^0-9]", "").ToInteger();
        // Create a new streamReader that knows the encoding and re-read into responseString
        reader = new StreamReader(stream, Encoding.GetEncoding(number));
        responseString = reader.ReadToEnd();
    }

    var doc = XDocument.Parse(responseString);
    var items = GetItemQuery(doc, Node);
    var any = items.Any();
    if (items == null || !any) return;
    m_Items = items;
}
```

The CopyandClose() method is used to copy the response from the web request in order that it can be read multiple times. In fact it is read up to three times:

1. When it is first read
2. When the encoding is read out of the XML header and
3. When the stream has to be re-read because it is not UTF-8 encoded.

The need to read the encoding out of the XML file and to re-read the stream using a new encoding explain all the additional code in the Read() method. You can see that the first few lines and the last few lines are identical to the Simple Version (above).

Parsing the Feed

The following relatively short code fragment harnesses the power of LINQ (Language Integrated Query) to parse the entire feed. Microsoft say this about LINQ:

"Language-Integrated Query (LINQ) is a set of features introduced in Visual Studio 2008 that extends powerful query capabilities to the language syntax of C# and Visual Basic. LINQ introduces standard, easily-learned patterns for querying and updating data, and the technology can be extended to support potentially any kind of data store. Visual Studio includes LINQ provider assemblies that enable the use of LINQ with .NET Framework collections, SQL Server databases, ADO.NET Datasets, and XML documents."[1]

LINQ is a large topic in its own right as evidenced by the number of books that have been written about it so the unfamiliar reader should search the available literature for a tutorial that meets their needs. That said, there is one aspect of LINQ and the IEnumerable type (`GetChannelQuery()` returns an IEnumerable) that must be emphasized.

The IEnumerable employs "deferred execution" which means - in simple terms - that the query is not evaluated until the data is required. Just when the data is required is not entirely obvious (it depends on the code itself), but it means that the following code will not generate any errors because it doesn't actually parse the data that is retrieved; it creates instructions for parsing the data that will be executed later when the results of the query are accessed.

```
private static IEnumerable<fChannel> GetChannelQuery(XDocument xdoc)
{
    var random = new Random();
    xdoc = RemoveNamespace(xdoc);
    return from channels in xdoc.Descendants("channel")
         select new fChannel
         {
             fChannel_Title = GeneralField.Parse(channels.Element("title")),
             fChannel_Image = ImageField.Parse(channels.Element("image"), true),
             fChannel_TTL = GeneralField.Parse(channels.Element("ttl")),
             fChannel_Items = from items in channels.Descendants("item")
                              select new fItem
                              {
                                random = random.Next(1000),
                                fItem_Title = GeneralField.Parse(items.Element("title")),
                                  fItem_Description =
GeneralField.Parse(items.Element("description")),
                                fItem_Guid = GeneralField.Parse(items.Element("guid")),
                                fItem_PubDate = DateField.Parse(items.Element("pubDate")),
                            fItem_Thumbnails = from thumbnails in items.Elements("thumbnail")
                                               select ImageField.Parse(thumbnails, true)
                              }
         };
}
```

Note the call to `random.Next()` and the addition of a random value to the data structure created by the parse. This is used to randomize the order of the records read from the data source if required by the user (see *Ordering the Feed* on page 13). The call to `RemoveNameSpace()` strips any additional name spaces that are used by the feed. This is done to (a) make the parse a little simpler and (b) ensure that name spaces are not prefixed to any field names at the output.

Ordering the Feed

Because the feed is parsed using LINQ and held as an IEnumerable, sorting is easy. Here, for example, is the code to sort the feed by the random value (remember the call to `random.Next()` in the previous section). This is how the order of the feed is randomized:

```
public bool RandomizeOrder()
{
    if (m_Items != null)
    {
        m_Items = m_Items.OrderBy(item => item.random);
        return true;
    }
    return false;
}
```

1. http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx

We can do this without fear of problem because we know that item.random must exist. The same is not true of date/time ordering. Because it is possible that the `pubDate` field is absent from an item or cannot be parsed in one or more items, date sorting has to check whether the date exists. Normally you would expect the problem to appear in `GetChannelQuery()` when the field is first parsed but deferred execution means that the problem does not appear until now when the field is first used. The code to sort by date is thus:

```
public bool OrderByDate(bool ascending=true)
{
    if (m_Items != null)
    {
        bool SafeToSort = true;
        foreach (var i in m_Items)
        {
            // Because of deferred execution, any null timestamps will not be spotted until
later.
            // Check now so an appropriate error can be reported
            SafeToSort = SafeToSort && (i.fItem_PubDate != null);
        }
        if (SafeToSort)
        {
            m_Items = @ascending ? m_Items.OrderByDescending(item => item.fItem_PubDate.Date):
            m_Items.OrderBy(item => item.fItem_PubDate.Date);
        }

    else
        {
            throw(new Exception("Some records do not contain a valid pubDate field."));
        }
        return true;
    }
    return false;
}
```

Note that, if a timestamp is missing, an exception is deliberately thrown to ensure a meaningful error is reported on the UI. Remember it is *RSSFeed.cs* that reports all errors.

GroupItems()

This method causes the specified number of input records to be grouped together into a single output record. For an RSS feed, the number of records is chosen by the user via the UI but not all feeds require this facility. The code for this method is a little long but the logic is simple enough. It simply adds input records to the current output record until either the desired number is reached (`groupSize`) or the input is exhausted (`numItems`). There are many different transformations that could be performed upon a feed and this is only one of them. You could, for example, group items according to the value of one specific field or filter the feed to present only records that have some specific properties.

GetFirst() and GetNext()

Fundamentally, `GetFirst()` is only required to return the first item for output. If `GroupItems()` has been called then `GetFirst()` doesn't have very much to do because the output has already been assembled into `m_OutputFeed`. If `GroupItems()` has not been called, `GroupItems()` first has to assemble the output into `m_OutputFeed`.

Note that `GetFirst()` and `GroupItems()` create output records that have different "headers"; grouping two (or more) input records means that e.g. the "*Title*" field cannot be labeled `CTITL` because there is more than one contender for that label. So, `GetFirst()` returns the first output record and each successive call to `GetNext()` provides the next record in series. Revisiting some code from *RSSFeed.cs* that we looked at earlier, you can see how this works:

```
var frec = rssFeed.GetFirst();
for (int count = 1; count <= numItemsOut; count++)
{
    if (frec == null) break;
    fout.AddRecord(frec);
    frec = rssFeed.GetNext();
}
```

Installing A New Feed Handler

Before you can run your newly built Feed Handler, you must first install it:

1. Locate the Feed Handler DLL from the either the **bin\Release** or **bin\Debug** directory of your Visual Studio project. You will want the Debug version until you are sure you have completed development.
2. Copy the Feed Handler DLL to the *XPort Service* Folder, located immediately under the directory to which you first installed XPort.
3. Restart the XPort Service by running services.msc (FIG. 6). Right click on *AMX XPort* and select **Restart**:

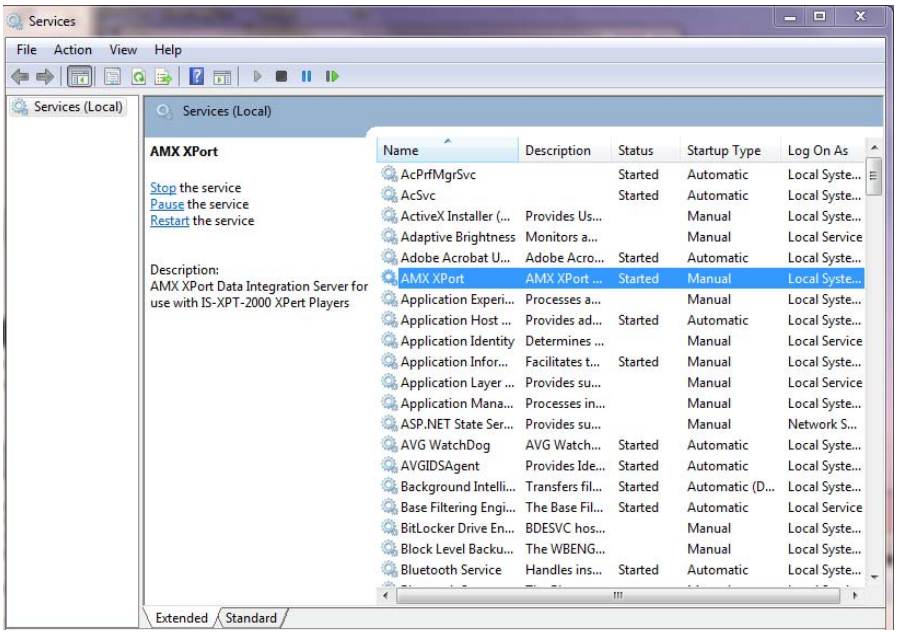


FIG. 6 Microsoft Services Console - Restart AMX XPort Service

Upon restart, the XPort service will detect your new Feed Handler. Use the Web UI to verify that your Feed handler is working. If not, please see the Troubleshooting section.

Testing and Debugging

Overview

There are two ways of debugging your Feed Handler: EXE

1. Use a Telnet client to monitor the diagnostic information that your code and the XPort framework output. The output may not provide all the information that you need but it is easy to do and is always a good starting point.
2. Use the interactive Visual Studio .NET debugger to set breakpoints within your Feed Handler and examine the content of variables.

The following sections deal with these two approaches In more detail.

Using Telnet

It is advisable to use a third party Telnet client rather trying to use the client built into the Windows command line. If you do want to use the Windows Telnet client please remember that it has to be explicitly enabled in Windows 7.

In the following example, Putty is used. Putty is simple, stable and free and can be downloaded from:

http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

You will only need to download the file putty.exe from the section "For Windows on Intel x86". The EXE file does not need to be installed and can be run as soon as you copy it onto your machine.

When you run putty.exe, the following dialog will appear. The fields should be filled in as shown before clicking on the "Open" button (FIG. 7):

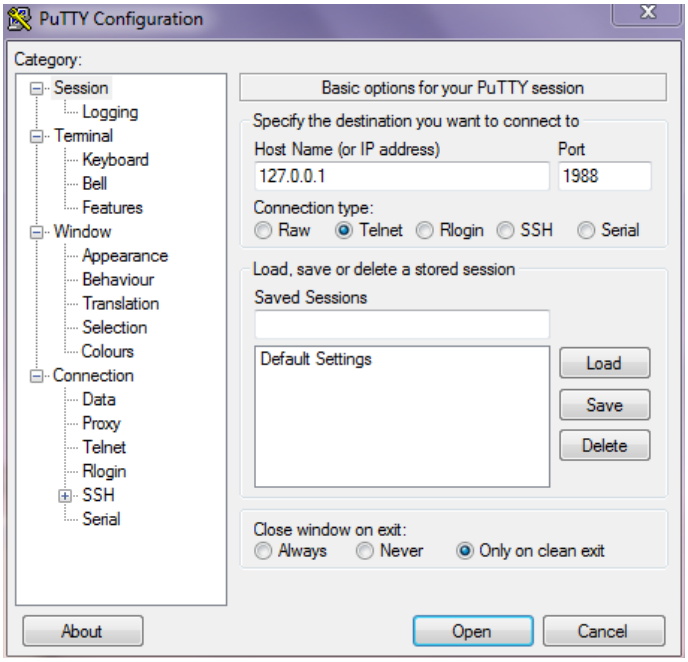


FIG. 7 Putty Configuration

Note that the port number can be changed if port 1988 presents problems. When you install XPort, two subfolders are created; Web and Service. To change the Telnet port number edit the file XPortService.exe.config from the *XPort Service* Folder and change the line

```
<add key="DiagnosticPort" value="1988"/>
```

to have the required value.

When you click on open, the Telnet window will open (FIG. 8):

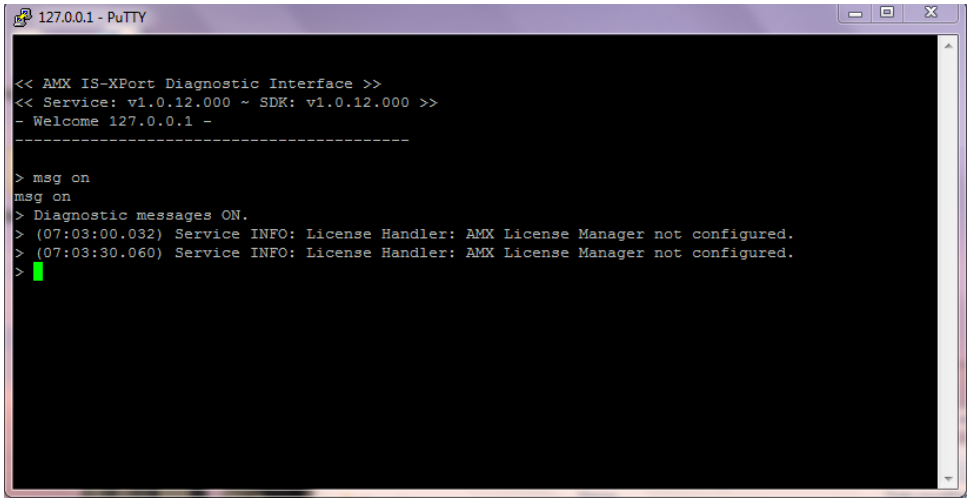


FIG. 8 Putty Telnet Window

If the XPort Service is running, it will announce itself on screen. If it is not running, the Telnet window will close immediately. To see the diagnostic messages that your feeds are currently generating you have to type the command "**msg on**". This is currently the only command that is implemented.

In the screenshot above, no feeds are running but the XPort framework is sending out reminders that the installation is unlicensed. No development systems need to be licensed as an unlicensed installation allows one active feed deliberately to support development. When a single feed has been activated on this installation, there is suddenly a lot more information to absorb. The following screenshot is for a Flickr feed (FIG. 9):

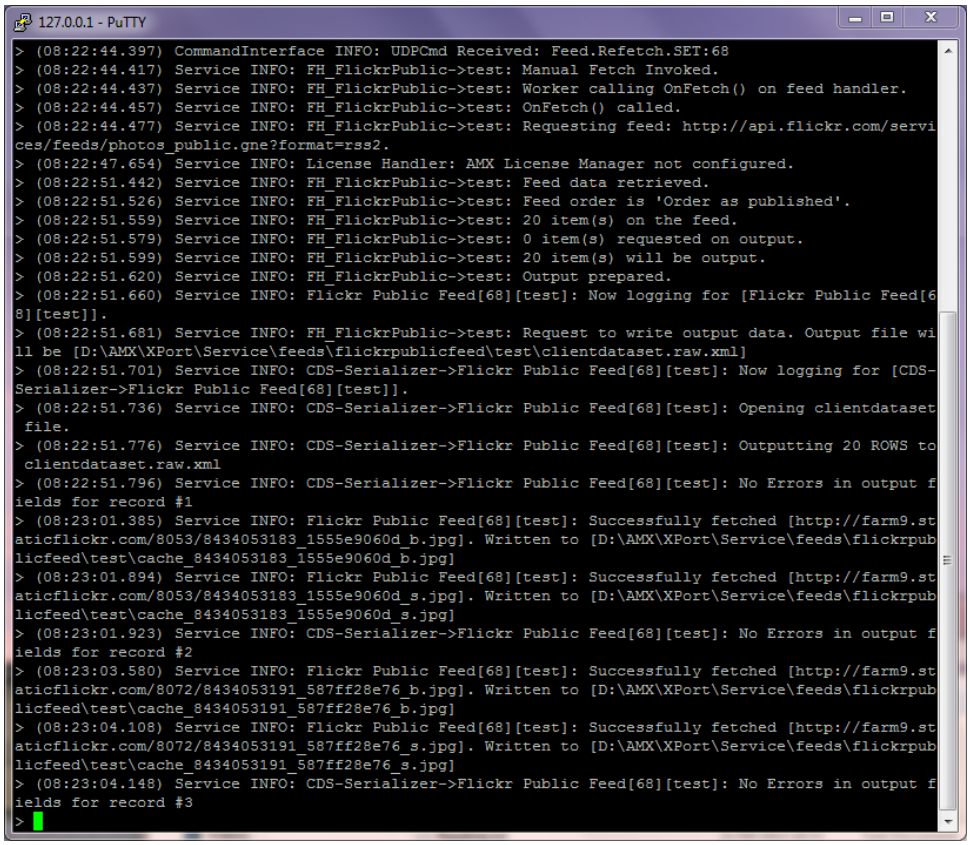


FIG. 9 Putty Telnet Window

This hints at one of the problems of Putty. It doesn't have a very big buffer so quickly loses information if there is a lot of activity. If this becomes a problem for you, you could use a more sophisticated Telnet client like the shadeBlue Indigo Terminal Emulator but it shouldn't be too much of an issue given that you will only ever be looking at a single feed at one time.

Using Visual Studio

When you have problems with a Feed Handler, there are two approaches open to you: You can either send more diagnostic information to the Telnet log or you can use a special purpose debugger. The former approach is more useful if you have an inkling what the problem is and determine that it would probably be better if the diagnostic information was permanently available. You don't really want to have to keep writing log statements to solve a problem, only to find that you have to remove the statements as soon as the problem is fixed in order to see the wood for the trees.

The .NET debugger is the ideal solution when you're not really sure what the problem is or if you just want to make sure that your code is responding as it should to a specific test case.

This section explains how to invoke the .NET bugger in Visual Studio Express 2012 to debug Feed Handlers. It does not explain how to use the debugger when it is up and running. These are the steps you have to perform:

1. First build and install your Feed Handler (see *Installing A New Feed Handler* on page 15).
2. Then, with your Feed Handler project still open in Visual Studio, select **Attach to Process...** from the DEBUG menu. The following popup will appear (FIG. 10):

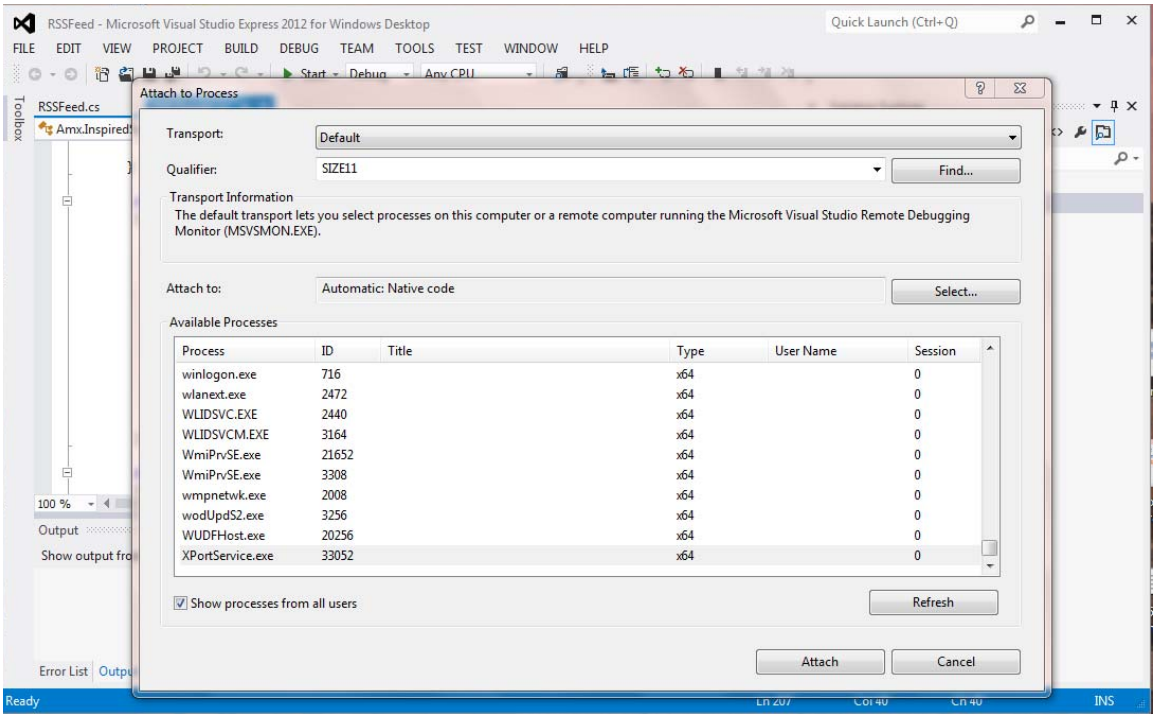


FIG. 10 Visual Studio - Attach To Process popup

NOTE: *Verify that "Show processes from all users" is selected then click on XPortService.exe in the main window before clicking on the Attach button.*

Now, you should be able to set breakpoints and inspect the values of variables without issue.

Troubleshooting

The best source of information about the behavior of a Feed Handler is the Telnet log. Any errors reported by the XPort framework can be found there along with messages output directly by the Feed Handler. There are, however, some problems that deserve special attention:

1. If you have gone through the installation process but your new feed does not appear on the UI you may have either:
   a. Given your Feed Handler an inappropriate file name; the file name of a Feed Handler DLL must start with the text "*FeedHandler_*"
   b. Given your Feed Handler class - which implements the *IFeedHandler* and *IFeedHandlerRegistrant* interfaces - a different name from the file name. If the file name is FeedHandler_RSSFeed.dll then the class declaration will look like this:

```
public class FeedHandler_RSSFeed : IFeedHandlerRegistrant, IFeedHandler
```

2. If UI controls do not appear on screen you may have created two controls with the same identity or you may have omitted an identity altogether. The Identity property must exist and must be different for every control.

3000 RESEARCH DRIVE, RICHARDSON, TX 75082
AMX.com | 800.222.0193 | 469.624.8000 | +1.469.624.7400 | fax 469.624.7153

Last Revised:
7/31/2015