Porting an Existing VBVoice™ Application into a Windows® Service

The information in this document applies to the following:

- Applications that need to run automatically at startup.
- Applications running on unattended servers.
- Situations where the decision to turn the application into a service occurred late in the development process, or the application was originally developed from a VBVoice shipped example.
- Developers who want the ability to run the application in both desktop and web service modes, as debugging a desktop application is easier than debugging a Windows service.

Upgrading a VBVoice example to the latest version of Visual Studio:

- Watch this **video** for a quick review of the upgrade process.

General overview of the porting process:

1. The first step is to configure the main form to start the VBVoice telephony system automatically.
2. The second step is to make the application accept command-line switches. Command-line options determine how the application will behave on startup.
3. A service component is then added to the project.
4. An installer component is added to the project.
5. Finally, the developer must add more command-line switches to control the service.

Step One: Configuring the main form to start VBVoice automatically

This action is accomplished by using a *Timer Class* in the form. The auto-start will then work in both desktop and web service modes. This process is fairly standard for VBVoice users; for a quick refresher on how this is done using one of our examples, watch the video **"Porting an existing VBVoice application into a Windows Service"** starting at 6:09.

```
private void InitializeComponent()
{
   …
   this.timer1.Enabled = true;
   this.timer1.Interval = 5000;
   this.timer1.Tick += new System.EventHandler(this.timer1_Tick);
   …
}

private void timer1_Tick(object sender, EventArgs e)
{
   vbvFrame1.StartSystem(false);
   timer1.Enabled = false;
}
```

Step Two: Making the application accept command-line switches

The steps below will allow you to determine how the application will behave on startup. Adding arguments to the *Main method* allows you to enable the application to run as a service. Otherwise, the application runs as a standalone desktop application.

1. Move the static *Main method* to its own class, if the method is not there already.
2. In order to check the commandline switches you need to add arguments to the *Main method*, which is an array of strings.
3. When the application runs with no arguments, the args array (arguments array of strings) will be empty; this is the default behavior. We recommend running an application with no arguments as a standalone desktop application.
4. Choose the commandline switch “service” to make the application run as a service.

```
class Program
{
   static void Main(string[] args)
   {
      if (args.Length < 1)
      {
         var thr = new Thread(delegate()
         {
            var frm = new frmEZFlow();
            Application.Run(frm);
         });
         thr.SetApartmentState(ApartmentState.STA);
         thr.Start();
      }
      Else
      {
         switch (args[0].ToLower())
         {
            case "service":
               var srvcs = new ServiceBase[] { new EZFlowService() };
               ServiceBase.Run(srvcs);
               break;
         }
      }
   }
}
```

Step Three: Adding a service component

This step describes how to create the object that will enable the application to run as a service.

1. Inherit from `System.ServiceProcess.ServiceBase`
2. Give it a unique name.
3. Impliment *OnStop method*: `Application.Exit();`
4. Impliment *OnStart method*:
   a. Create new STA thread and start it
   b. Instantiate the callflow form and pass it to `Application.Run`
5. Execute the same (or similar) code to start as a desktop.
6. Remove Single-Threaded Apartment (STA) from the original thread served by *Main*.
7. Instantiate the service component and pass it to `ServiceBase.Run` in *Main*.

```
public class EZFlowService : ServiceBase
{
   public const string MyServiceName = "EZFlowService1";

   public EZFlowService()
   {
      ServiceName = MyServiceName;
   }

   protected override void OnStart(string[] args)
   {
      var thr = new Thread(delegate()
      {
         var frm = new frmEZFlow();
         Application.Run(frm);
      });
      thr.SetApartmentState(ApartmentState.STA);
      thr.Start();
   }

   protected override void OnStop()
   {
      Application.Exit();
   }
}
```

Step Four: Adding an installer component

This section lists the steps to adding an installer to the application so that the service can be installed. For service applications to run, an installer component MUST be added to your project.

1. Inherit from `System.Configuration.Install.Installer`
2. Add to members of types:
   a. `System.ServiceProcess.ServiceProcessInstaller`
   b. `System.ServiceProcess.ServiceInstaller`
3. Initialize both members in the constructor. To do this:
   a. Define which account you want the service to run on.
   b. Give the service a user friendly name, but unique.
   c. Define how you want the service to start.
   d. Add the dependancy as needed.
4. Mark your installer component with the attribute `RunInstaller(true)`
5. Alter the registry to change the commandline used to start the service.
6. At this point you could register and unregister the service using the commandline: `InstallUtil <yourapp.exe>`

```
[RunInstaller(true)]
public class EZFlowServiceInstaller : Installer
{
   private ServiceProcessInstaller ezFlowProcessInstaller;
   private ServiceInstaller ezFlowServiceInstaller;

   public EZFlowServiceInstaller()
   {
      ezFlowProcessInstaller = new ServiceProcessInstaller();
      ezFlowProcessInstaller.Account = ServiceAccount.LocalSystem;
      ezFlowProcessInstaller.Username = null;
      ezFlowProcessInstaller.Password = null;

      ezFlowServiceInstaller = new ServiceInstaller();
      ezFlowServiceInstaller.ServiceName = EZFlowService.MyServiceName;
      ezFlowServiceInstaller.DisplayName = "EZFlow Service1";
      ezFlowServiceInstaller.StartType = ServiceStartMode.Manual;
      ezFlowServiceInstaller.ServicesDependedOn = new string[]
      { "Dialogic", "RuntimeManager" };

      Installers.AddRange(new Installer[]
      { ezFlowProcessInstaller, ezFlowServiceInstaller });
   }

   protected override void OnAfterInstall(System.Collections.IDictionary savedState)
   {
      using (var key = Registry.LocalMachine.OpenSubKey(
      @"SYSTEM\CurrentControlSet\services\" + EZFlowService.MyServiceName, true))
      {
         if (key == null)
            throw new Exception("Failed to open the registry");
         key.SetValue("ImagePath", key.GetValue("ImagePath") + " service");
      }
      base.OnAfterInstall(savedState);
   }
}
```

Step Five: Adding more commandline switches

The final step in porting your existing application into a Windows Service is adding the commandlines *start and stop* and *register and unregister*. The process for accomplishing this is very similar to adding the "service" commandline switch described above in Step Two.

Start and Stop:

```
case "start":
   var starter = new ServiceController(EZFlowService.MyServiceName);
   starter.Start();
   break;
```

Register and Unregister:

```
case "register":
   var installer = new AssemblyInstaller(typeof(EZFlowService).Assembly, null);
   installer.UseNewContext = true;

   var state = new Hashtable();
   try
   {
      installer.Install(state);
      installer.Commit(state);
   }
   catch
   {
      installer.Rollback(state);
   }
   break;
```

Conclusion

Often, it is useful or even required to turn an existing desktop application into a web service. Maintaining both a desktop application and a web service application is useful for the purposes of debugging, ensuring that your application runs smoothly. Using commandline switches allows you to easily establish a self-contained application. Commandline switches also simplify the installation and deployment process, especially on remote machines using terminal services.

About VBVoice

Pronexus VBVoice™ Interactive Voice Response (IVR) toolkit's latest release, VBVoice 8.20, synchronizes all Pronexus' offerings with support for development in Microsoft® Windows® 8, 8.1 and Windows Server® 2012 using the latest Visual Studio® 2013. VBVoice 8.20 includes Microsoft® Lync™ interoperability, secure media and proprietary call control. Download VBVoice IVR toolkit today to discover how easy it is to develop an IVR application.