

ProcSee .NET API tutorial

This tutorial is intended to give the user a hands-on experience on how to develop an external application for ProcSee using the .NET API. It is not a tutorial in designing a ProcSee application. This information is found elsewhere in the ProcSee tutorial distributed with ProcSee. Instead, this tutorial focuses on the external application, i.e. how to interact with ProcSee from a .NET application. In this tutorial we will develop the external application in C# since it has become the most popular programming language for the .NET platform.

You do not have to be an expert, but some knowledge about the .NET platform and the C# language is recommended, or you have some experience from other .NET languages, like Managed C++, Visual Basic .NET, etc. It is required that you do have some knowledge of the fundamental concepts of OOP. Therefore, this tutorial will not describe in detail the C# code needed to implement the framework, i.e. how to create the menus, message boxes, etc. However, this tutorial will describe in detail each line of code that involves the .NET API.

It is assumed that you are acquainted with one of the Microsoft Visual Studio integrated development environments (IDE). In this tutorial the screen shots and the code examples will be based on the Microsoft Visual Studio 2010.

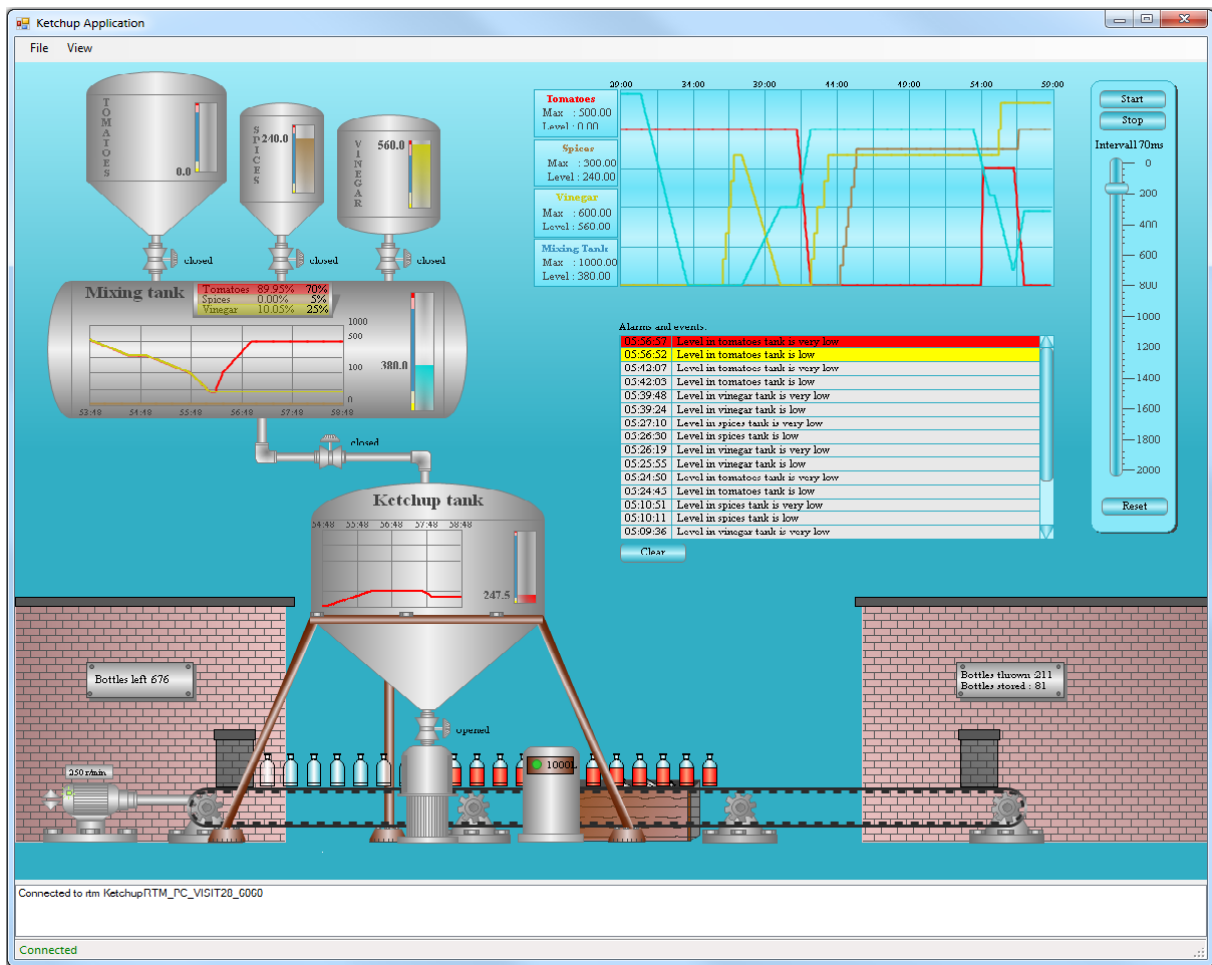
This tutorial is based on the Ketchup demo application distributed with ProcSee. It is located in the %PROCSEE_DIR%/demo/ketchup/%ARCH% directory.

Before we start developing the external application in C# you should familiarize yourself with this demo. This ProcSee application is the base for the external application we are going to develop in this tutorial. Try running it, either by double clicking on the KetchupApp.ptx file, or selecting the Ketchup Demo under the demo folder in the ProcSee menu. Note that this demo application uses a simulator developed in the pTALK language to get live updated values in the pictures. Next in this document we will be developing parts of this simulator in the C# language.

In the tutorial, you will develop an external application that step by step covers most parts of the .NET API. The external application will have functionality to start an RTM, update variables, respond to method calls from an RTM, and export a window where an RTM can display its pictures. When completing this tutorial you should have learned how to:

1. Prepare the IDE for the .NET API.
2. Create and initialize the external application using the .NET API.
3. Create a message output provider.
4. Initialize and register the external application with the ProcSee control server.
5. Initiate a connection to a ProcSee RTM.
6. Export a window to the ProcSee RTM where pictures can be displayed.
7. Start the ProcSee RTM from the external application.
8. Create .NET wrapper classes for the struct definitions.
9. Create and link to variables in the ProcSee RTM.
10. Update and transfer variables to the ProcSee RTM.

11. Register remote functions which can be called from the ProcSee RTM.
12. Use the execute function to run pTALK commands in the ProcSee RTM.



1. Prepare the IDE for the .NET API

This description applies to the Microsoft Visual Studio 2010 IDE only. We begin by working through the New Project dialogue window. At the end we will have a basic C# project needed when building the external application. Start the Microsoft Visual Studio 2010 IDE. Select New and Project from the File menu. This action opens the New Project window. In the Installed Templates tab, select Visual C# language. This action displays the available C# project types. In the project types list, select Windows Forms Application. This is the project type that we will be using in this tutorial. It creates an application with a Windows Forms user interface. Next, click on the Name input field and enter the name KetchupApp. This will be the name of our project. Click the Browse... button to change the default project location. Move to, or create a new destination folder for this tutorial and click the Select Folder button. We have now made the necessary changes and settings in the New Project window. Click the Ok button to complete the creation of the C# application. The Microsoft Visual Studio 2010 has now generated the KetchupApp project containing sources and project metadata needed in order to build the C# application.

The next step is to check whether the project uses the correct .NET framework target or not. In the Solution Explorer view select the KetchupApp node. Right click on this node and select Properties from the popup menu. Verify that the Target framework isn't one of the client profile frameworks, i.e. .NET framework 3.5 Client Profile or the .NET framework 4 Client Profile. The target framework should be .NET framework 2 or later. Select the newest .NET Framework, for instance 4.

The final step is to setup the dependency to the ProcSee .NET API library. The C# compiler will issue errors in the output window complaining about unknown classes if you forget to add this dependency before building the application. In the Solution Explorer view right click on either the KetchupApp node or the References node and select Add Reference... This action opens the Add Reference window.

We provide two versions of our ProcSee .NET library due to changes starting from .NET 4.0 on activation and mixed-mode assemblies. Please use the one that fits your target framework. In both cases the DLL name is the same, but its parent folder varies. See below for more information.

- dotNetApi\v2\ProcSeeApiDotNet.dll targeted for use with .NET 2.0 to 3.5sp1
- dotNetApi\v4\ProcSeeApiDotNet.dll targeted for use with .NET 4.0 and later

In the Browse tab, move to the %PROCSEE_DIR%/lib/%ARCH%/dotNetApi directory and then select ProcSeeApiDotNet.dll under either folder v2 or v4, depending on your choice of framework. Click on the OK button. The ProcSeeApiDotNet assembly should now appear in the References list in the Solution Explorer view.

2. Create and initialize the external application using the .NET API

Now, we can start developing the external application. The main focus in this chapter and the remaining parts of this tutorial will be on the class CProcSee. This class contains the functionality needed to get the ketchup application up and running.

In the previous chapter we created the tutorial project and setup the reference to the .NET API. That step generated a class that we so far haven't given any attention. The name of this class is Form1 which is derived from the class Form located in the System.Windows namespace. This is our main class. But, before we continue we will rename this class from Form1 to CProcSee. Select the Solution Explorer view. In this view right click on the Form1.cs node and select Rename from the popup menu. Rename this class to CProcSee.cs and press return. Click the Yes button in the question message box that appears on the display when renaming the Form1 class.

The IDE shows the CProcSee class in a design view and a source view. In the design view you can draw your user interface elements in a WYSIVYG view. The design view allows you to add user interface elements and control their size and position using the mouse. It also

provides a properties view where the user interface element properties can be set. In addition properties can be set from special popup menus. Notice that the IDE has created a partial class called CProcSee.Designer.cs where the user interface components and their property settings are put. Notice that all changes made in the design view will be synchronized with this source view. Leave this source file as is. The IDE provides the CProcSee.cs source view where we will add the C# code needed in this tutorial.

Next, we are going to add code which displays a question dialog box asking the user whether he/she wants to close the external application or not. This dialog will be displayed in the external application when clicking the close button in the frame window (the x) or when selecting Exit in the File menu. The CloseApplication method returns true when the user clicks the Yes button, otherwise false. Add the following code:

```
private bool CloseApplication()
{
    if (MessageBox.Show(this, "Do you really want to quit", "Ketchup Application",
        MessageBoxButtons.YesNo, MessageBoxIcon.Question,
        MessageBoxDefaultButton.Button2) == DialogResult.Yes)
    {
        return true;
    }

    return false;
}
```

When the if statement in the source code above evaluates to true the function just returns true, i.e. when the Yes button is clicked. Later on in this tutorial we will add more code in this block.

Now, switch to the CProcSee design view. The next code catches the close action when the user clicks the close button (the x) in the window frame. In the design view, select the form dialog. In the Events tab in the properties view select the FormClosing event. Name the action listener OnClosing and press return. In the action part of this method call the method CloseApplication. Notice that the FormClosingEventArgs property Cancel is set to true if false is returned from the CloseApplication method (The No button is clicked). That will prevent the external application from closing. Add the following code:

```
private void OnClosing(object sender, FormClosingEventArgs e)
{
    if ( !CloseApplication() )
    {
        e.Cancel = true;
    }
}
```

Switch back to the design view again. Create a menu bar (MenuStrip) with a file menu (ToolStripMenu) and an exit button (ToolStripMenuItem). Add an action listener for the exit button and name it for instance OnExit. Switch to the source view and implement the exit action listener. In this method call the method CloseApplication. If this method returns true, then close the external application. Add the following code:

```
private void OnExit(object sender, EventArgs e)
{
    if (CloseApplication())
    {
        Environment.Exit(1);
    }
}
```

You can now build the project which should compile without any errors and warnings. For the time being this application is not very interesting. It displays a frame window with a menu bar. Try running it to check that you have added the source code described so far in this tutorial.

3. Create a message output provider

Before continuing with the `PcProcess` class we need to explain the message output functionality in the .NET API, which is provided by the class `PcMessageOutput`. The messages issued in the .NET API is by default sent to standard output, which simply doesn't fit very well into a GUI application. By providing a message output class the messages can be redirected to an output window, like a list box component or a log file, for example.

Each message issued in the .NET API is associated with a message category, where the message category is one of the predefined categories defined in the .NET API, like error, warning, debug, flow, etc. The `PcMessageOutput` class offers functionality to suppress these message categories. By default fatal errors, errors, warnings and information messages are enabled. Note that fatal errors and errors should never be suppressed by the external application. Such information should always be reported to the user so she/he can take appropriate actions. Normally the debug categories (five different debug levels exist in the API) are suppressed by default and should only be enabled when running the external application in a development phase.

The `PcMessageOutput` class also offers functionality to customize the message format in the .NET API. The message format is a concatenation of pure text and directives. A directive is a sequence of characters surrounded by curly braces, {}, recognized by the `PcMessageOutput` class, for example "Message: {MSG}", "{TIME:T} {MSG}", etc. The default message format in the .NET API is "{TIME:T} {CAT} {MSG}", which is a concatenation of the date and time ({TIME:T}, plus the category ({CAT}) and the actual message to print ({MSG}).

We recommend that you always create an instance of the class `PcMessageOutput` before calling any method or instantiating any other class in the .NET API. The information provided in the message outputs can be very useful when the application fails for various reasons. For GUI applications the messages issued in the .NET API is silently ignored without an output provider.

Next we will add a `Listbox` component which is used by the external application to print messages from the external application and messages issued in the .NET API. Switch to the `CProcSee` design view. Locate the `Listbox` component in the Toolbox. Click and drag this

component on to the design surface. In the properties view, locate the Anchor property and anchor the ListBox component to the left, right and bottom. Name the component `mMessageOutputList`.

Switch back to the source view again. Before we instantiate the default message output provider, add the following using statement in your `CProcSee` source file.

```
using ProcSee.API;
```

Next, in the `CProcSee` class, create the message output provider attribute used by the .NET API. Name the attribute `mMessageOutput`. The data type is of the class `PcMessageOutput`.

```
private PcMessageOutput mMessageOutput;
```

In the class constructor after the call to the initialization routine `InitializeComponent()` make a call to the method `InitMessageOutput()`. Add the following lines of code to your `CProcSee` class.

```
internal void PrintMessage(String message)
{
    mMessageOutputList.Items.Add(message);
    System.Diagnostics.Debug.Print(message);
}

private void OnPrintMessage( PcMessageCategory category, string message )
{
    PrintMessage( category + ": " + message );
}

private void InitMessageOutput()
{
    mMessageOutput = new PcMessageOutput("{TIME:T} {MSG}",
                                         new PdMessageListener(OnPrintMessage));
    mMessageOutput.Disable(PcMessageCategory.PeInformation);
}
```

In the source code above the output message is printed to the output window in the debugger in addition to being added to the list box component. In a real application, the message would probably be dumped to a file as well (using another message output instance providing more information). Notice that the .NET API supports multiple output providers with different or equal message formats. But this is out of the scope for this tutorial. In the source code above is the information category disabled (the call to the class `PcMessageOutput` method `disable`). This category produces lots of messages that we don't need to pay any attention to, but can be useful in a development phase for debugging purposes.

4. Initialize and register the external application with the ProcSee control server

So far in this tutorial you have learned how to setup the dependency to the `ProcSeeApiDotNet.dll` library, create a `CProcSee` class and initialize a message output provider. Now, it's time to initialize the .NET API and register the external application, i.e.

register the name of the external application and supply other process related information to the ProcSee control server. The class *PcProcess* provides this functionality in the .NET API.

Create a new attribute in the *CProcSee* class of type *PcProcess* and give it the name *mProcess*. In the source code declare this attribute after the *PcMessageOutput* attribute, like:

```
private PcProcess mProcess;
```

In the *CProcSee* constructor, put the call to the method *InitProcess()* after the call to the method *InitMessageOutput()*, like:

```
public CProcSee()  
{  
    InitializeComponents();  
    InitMessageOutput();  
    InitProcess();  
}
```

The *InitProcess()* method creates the *PcProcess* instance which registers the process in the ProcSee control server. The first argument to the constructor is the name of the process. This name must be unique. The *PcProcess* constructor will otherwise fail to initialize the instance and throw an exception. Notice that the .NET API provides a helper method which creates names that are guaranteed to be unique. The name returned is based on a user defined prefix (and postfix), plus the name of the host computer and the PID (process identifier). In the tutorial the prefix is “KetchupApp”. An example of a unique name is: “KetchupApp_GANDALF_2469”, where GANDALF is the name of the computer, and 2469 is the PID. In this tutorial we will be using this method to create unique process names. See the source code below.

The process class argument to the *PcProcess* constructor is optional. In this tutorial the process class is called *KETCHUP_APP*, which provides extra information about the process. This data can be useful when requesting data from the *control* server or when running the utility application *controlutil*. Therefore, we recommend that you always provide the process class when initializing the external application.

```

private void InitProcess()
{
    try
    {
        String name = PcControlUtils.CreateUniqueName("KetchupApp");
        mProcess = new PcProcess(name, "KETCHUP_APP");
    }
    catch (PcException ex)
    {
        MessageBox.Show(this, "ex.Message", "PcProcess failed",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        Environment.Exit(1);
    }
}

```

Before compiling and running the external application we need to dispose the PcProcess instance before closing the external application. In the CloseApplication method call the PcProcess method Exit.

```

private bool CloseApplication()
{
    if (MessageBox.Show(...)
    {
        mProcess.Exit();
        return true;
    }
    return false;
}

```

Now, compile and run the external application. This is done to test that you have the environment variables set correctly. If the external application fails to start, check that the environment variables PROCSEE_DIR and ARCH are set properly.

5. Initiate a connection to a ProcSee RTM

The .NET API provides two classes which can be used to represent an RTM in an external application. The names of these classes are PcRTMAsClient and PcRTMAsServer (derived from the base class PcRTM). The class PcRTM is an abstract base class which cannot be instantiated directly in the external application. It is used as arguments in methods calls invoked by the .NET API. These classes represent the external application's view or link to a specific ProcSee RTM.

In this tutorial the external application will be a client to the RTM, i.e. the external application initiates the connection to the RTM. Therefore, we will be using the class PcRTMAsServer. Notice that the class PcRTMAsClient is used when the RTM is responsible for establishing the connection to the external application, i.e. the RTM is the client. Instances of the class PcRTMAsClient are created on requests from RTMs in the PdConnectFactory delegate.

The following functionality is provided by the PcRTM derived classes:

- executing pTALK commands.
- read and link to variables in the RTM.

- register remote functions.

In the CProcSee class add a private attribute of the class PcRTMAsServer. Give this attribute the name mRTM.

```
private PcRTMAsServer mRTM;
```

Now, we have to initialize the mRTM attribute. The first argument to the PcRTMAsServer constructor is the name of the RTM to connect to. If the RTM with the specified name isn't running, you have to manually start an RTM with that name (start it with the `-n` option and the `-r KetchupApp.pctx` arguments). Later in this tutorial you will learn how to start the RTM from the external application. The second argument specifies the name of the ProcSee application in the RTM. This is the current ProcSee application the external application will be connected to in the RTM. If the application doesn't exist in the RTM, an empty application will be created with the specified name. The last argument is a connection state listener which provides information when the connection state changes, for instance when a connection is established or broken. Extra information is provided in the arguments to the callback methods in this interface. For simplicity, we use in this tutorial the adapter class PaConnectionStateListener instead of the interface PiConnectionStateListener. Notice that you should always implement the `onError` callback of the PiConnectionStateListener interface when running more complex systems.

The `InitRTM` method creates a new instance of the PcRTMAsServer class and calls the member method `Connect`. The external application will never try to connect to the RTM if you forget to call this `Connect` method. The last argument to the PcRTMAsServer constructor is a PiConnectionStateListener instance. In the source code below is the ConnectionState instance an inner class which is derived from the class PaConnectionStateListener. Add the following method to the CProcSee class:

```
private void InitRTM()
{
    try
    {
        mRTM = new PcRTMAsServer("rtm", "KetchupApp", new ConnectionState(this));
        mRTM.Connect();
    }
    catch (PcException ex)
    {
        PrintMessage("PcRTMAsServer error " + ex.Message);
    }
}
```

Implement the class `ConnectionState` as an inner class in the class `CProcSee`. This class overloads the `OnConnect` and `OnDisconnect` methods declared in the `PiConnectionStateListener` interface. These methods call the `OnConnect` and `OnDisconnect` methods in the `CProcSee` class, respectively. We will be adding these methods later in this chapter. But for now, add the following source code to the `CProcSee` class:

```
class CProcSee
{
    ...
```

```

private class ConnectionState : PaConnectionStateListener
{
    private CProcSee mOwner;

    public ConnectionState(CProcSee owner)
    {
        mOwner = owner;
    }

    public override void OnConnect(PcRTM rtm, PeConnect type)
    {
        mOwner.OnConnect(rtm);
    }

    public override void OnDisconnect(PcRTM rtm,
                                       PeDisconnect type,
                                       PeDisconnectInfo info)
    {
        mOwner.OnDisconnect(rtm);
    }
}

```

```

...
}

```

The external application does not offer any component in the GUI that visualizes the connection state. We will now add a status bar in the lower part of the design surface which provides this connection status. Switch to the design view again. Locate the StatusStrip component in the ToolBox. Click and drag it on to the design surface. Attach it to the lower part of the design surface. Click on the StatusStrip which will display a drop down list. Click on the list and select StatusLabel. Name this label mRTMState and set the Text property to Disconnected. Switch back to the source view again and implement the following code:

```

internal void OnConnect(PcRTM rtm)
{
    PrintMessage("Connected to rtm " + rtm.Name);
    mRTMState.Text = "Connected";
    mRTMState.ForeColor = Color.Green;
}

internal void OnDisconnect(PcRTM rtm)
{
    PrintMessage("Lost connection to the rtm " + rtm.Name);
    mRTMState.Text = "Disconnected";
    mRTMState.ForeColor = Color.Red;
}

```

In the CProcSee constructor after the call to *InitProcess()* add a call to the method *InitRTM()*.

```

public CProcSee() {
    InitializeComponents();
    InitMessageOutput();
    InitProcess();
    InitRTM();
}

```

Now, compile and run the external application again. Be sure that the RTM is still running with the ProcSee application, KetchupApp. This time the external application should be able to connect to the RTM. A message is issued in the message output window when the connection is established, and the connection state is displayed in the status bar.

6. Export a window to the ProcSee RTM where pictures can be displayed

This chapter focuses on how to make a drawing surface in the external application available to the RTM, i.e. a window the RTM can use to display pictures.

This description applies to the Microsoft Visual Studio 2010. The class PcWindow must be added to the toolbox palette in the IDE before we can start creating an exported window. Follow the next steps to add this class. We will add the PcWindow element to the ProcSee category in the Toolbox palette. The first we have to do is to create the ProcSee category. Now, right click in the Toolbox palette and select Add Tab from the popup menu. Enter the name ProcSee in the input field in the Toolbox palette and press return. Move the mouse to the ProcSee tab and click the right mouse button. In the popup menu, select Choose Items... This action opens the Choose Toolbox Items dialogue. In this dialogue select the .NET Framework Components tab and click the Browse... button, which opens the file dialogue window. Move to the %PROCSEE_DIR%/lib/%ARCH%/dotNetApi directory and select the ProcSeeApiDotNet.dll, i.e. the .NET API assembly file. Click the Open button in the file dialogue window and the OK button in the Choose Toolbox Items window. The class PcWindow should now appear in the Toolbox palette under the ProcSee category that you created.

Now, we are ready to create an instance of the class PcWindow. Switch to the design view. In the Toolbox locate PcWindow. Click on the PcWindow object and drag it to the design surface. You don't need to struggle with the layout manager to control its size and position. It will automatically fit into the position and the size where you drop it. Attach it to the upper left corner and the lower right corner of the drawing surface. Let it be anchored to the left, top, right and bottom (set with the Anchor property). Name this component mExportedWindow. In the properties window, locate the WindowName property, and give this property the name ExportedWindow. This is the name of the exported window within the ProcSee RTM.

Next, find the PictureName property and enter the name of the initial picture to display in the window. The name of this picture in the KetchupApp demo application is KetchupProcess. Switch back to the source view again.

There is one issue that has to be resolved before we can compile and run the external application. That is, which RTM is associated with the PcWindow instance. The .NET API does not provide this functionality in the design view. The class PcWindow has a property called RTM which creates this association. Assign the PcRTM instance mRTM to this property. Scroll down to the InitRTM() method. Insert the following code line after the call to mRTM.Connect().

```
mExportedWindow.RTM = mRTM;
```

That's it! If you have followed this tutorial step by step the external application should now be in a state where the RTM has a drawing surface inside the external application where it can display its pictures. Compile the project and run it. Be sure that the RTM is still running with the ProcSee application, KetchupApp. As you should see, the ketchup process picture is automatically displayed inside the external application when the connection to the RTM is established. You can play a little bit around with it. The simulator code is still created entirely in the pTALK language. So far there are no variable updates or any interaction between the RTM and the external application. We will be coming back to this later in this tutorial.

7. Start the ProcSee RTM from the external application

This chapter focuses on how to start the RTM from the external application instead of connecting to an already running RTM, which we have been doing so far in this tutorial. The question is how do we do that? The answer to the question is the class PcRTMProcessBuilder. This class contains functionality to start an RTM on the host computer. In addition it manages the collection of process attributes needed by the RTM, i.e. options to pass on to the RTM, environment settings, and the RTM's current working directory. These settings must to be set programmatically before invoking the method which starts the RTM.

The class PcRTMProcessBuilder offers functionality to add new, modify or remove existing environment settings. It is also possible to start an RTM with an explicit set of environment variables, i.e. the RTM does not inherit any environment settings from the external application. If you have cleared the environment settings, always remember to set the PROCSEE_DIR and the ARCH environment variables before invoking the start method.

The class PcRTMProcessBuilder also stores the options which are passed on to the RTM. It provides two overloaded add argument methods. The first add method accepts an option and an option argument, like “-r KetchupApp.pctx”, where “-r” is the option, and “KetchupApp.pctx” is the option argument. The second method accept one option, like “-g”. Note that this class does not check whether the options are recognized by the RTM or not. It only maintains a list of the options to pass on to the RTM. For more information about the options the RTM accepts, see the RTM executable manual page in the ProcSee Reference Manual.

There are two issues that have to be described before you can add the source code below. First, when the RTM is started it must be given a unique name. The reason is that we do not want to get a name conflict from the control server when starting the RTM. A call to the static PcControlUtils method CreateUniqueName generates a unique name. This name is added to the PcRTMProcessBuilder instance with the AddArgument method. Second, always set the RTM's current working directory (CWD) before invoking the start method. Normally the CWD should be set to the directory where the ProcSee application (*.pctx) file is located. In this tutorial the KetchupApp.pctx file is located in the %PROCSEE_DIR%/demo/ketchup/%ARCH% directory.

Add the following method to the CProcSee class.

```
private String StartRTM()
{
    try
    {
        String rtmName = PcControlUtils.CreateUniqueName("KetchupRTM");
        String workingDirectory = Environment.GetEnvironmentVariable("PROCSEE_DIR") +
            @"\demos\ketchup\" +
            Environment.GetEnvironmentVariable("ARCH");

        PcRTMProcessBuilder processBuilder = new PcRTMProcessBuilder();
        processBuilder.AddArgument("-n", rtmName);
        processBuilder.AddArgument("-r", "KetchupApp.pctx");
        processBuilder.AddEnvironment("KETCHUP_APP_VERSION", "1");
        processBuilder.AddEnvironment("KETCHUP_APP_NAME", mProcess.ProcessName);
        processBuilder.SetWorkingDirectory(workingDirectory);

        processBuilder.Start();
        return rtmName;
    }
    catch (PcException ex)
    {
        MessageBox.Show(this, ex.Message, "Failed to start the RTM",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        Environment.Exit(1);
    }

    return null;
}
```

Two environment variables were added to the process builder instance in the StartRTM method. These variables are needed by the constructor function in the Ketchup application. The KETCHUP_APP_NAME is used in the ProcSee application to create and load the pdat files into the correct process. The environment variable KETCHUP_APP_VERSION is required when the ketchup application updates the simulator. Notice that the value of the environment variable KETCHUP_APP_VERSION will be modified as we proceed in this tutorial. It is used to disable functionality in the ketchup simulator (pTALK code) that we instead implement in the external application. These environment variables will not be described any further in this tutorial.

Call the StartRTM method from InitRTM, and modify the PcRTMAsServer constructor's first argument to use the name returned from the StartRTM method, like:

```
private void InitRTM() {
    try {
        String rtmName = StartRTM();
        mRTM = new PcRTMAsServer( rtmName, "KetchupApp", ... ) {
```

The last changes we have to do before we can test the external application is to stop the RTM when the application terminates. In the CloseApplication method add the following lines of code.

```
private bool CloseApplication()
{
```

```

    if (MessageBox.Show(... ) )
    {
        try
        {
            mRTM.StopRTM();
            mProcess.Exit();
        }
        catch (PcNoConnection ex)
        {
        }
        return true;
    }
    return false;
}

```

Now, compile and run the external application. Try to start several instances of the application. They should start without errors. That is because we have created unique names for both the external application and the RTM.

8. Create .NET wrapper classes for the struct definitions

The next chapters focus is on how to update the ProcSee variables from the external application. But, first we have to create C# wrapper classes for the ProcSee struct definitions. The ProcSee struct definition resembles the “C” programming language syntax which does not fit very well into a .NET world. The .NET API provides wrapper classes for the primitive data types and arrays of the primitive data types, but not for the complex struct definitions used by the Ketchup demo application.

Therefore, wrapper classes must be created for the struct definitions used by the Ketchup application. The utility program pdattoapi provides this functionality. It is located in the %PROCSEE%/bin/%ARCH% folder. It accepts one or several ProcSee variable definition (*.pdat) files, and generates a file containing the wrapper classes for the struct definition in the C# language (including array wrapper classes for the struct definitions).

We need to explain some of the pdattoapi program arguments before running the command which generates the wrapper classes.

The pdattoapi utility program generates output for multiple target languages like C#, Managed C++ and Java. To generate wrapper classes for the C# language, specify `-language C#`. Note that the language argument is a required option. If this argument isn't specified the program terminates without generating any wrapper classes.

All the .NET wrapper classes generated by pdattoapi can be prefixed with a string using the prefix option, i.e. `-prefix`. In this tutorial we will use the three letters Pwc, which is an abbreviation for ProcSee Wrapper Class.

Now, it's time to generate the struct definitions. Open a command shell. Change the folder to the solution directory under your .NET project. Run the following command from the terminal window (or add it as a pre-build step in the IDE).

```
pdattoapi -language C# -prefix Pwc -namespace KetchupApp.Structs
          -file RecordDefinitions.cs
          %PROCSEE_DIR%\demos\ketchup\RecordDefinitions.pdat
```

If the command is entered exactly as in the example above it should produce the wrapper classes needed by the external application. The classes are put in the file `RecordDefinitions.cs` (specified with the `-file` option), and organized in the namespace scope; *KetchupApp.Structs* (specified with the `-namespace` option).

Next, add the generated file to the KetchupApp project. Switch back to the IDE and select `Add Existing Item...` from the Project menu. Add the file `RecordDefinitions.cs` and click the `Add` button in the file selection box. The `RecordDefinitions.cs` file should now be added to your KetchupApp project.

Let's take a look at these classes. Select the Class View tab in the IDE and expand the KetchupApp node and the structs node. As you can see, six different classes have been generated and added to the project. Note that you should never modify any of these classes (because they are generated). However, you can create new wrapper classes by deriving from these generated classes, and overload the `PiReadScriptListener` method `OnCreateVariable` or `OnCreateArrayVariable`. But, that is out of the scope of this tutorial. If you have made modifications to the struct definitions, always remember to rerun the `pdattoapi` command to generate new wrapper classes. Otherwise, it may lead to unpredictable results and heap corruption. It is recommended that these wrapper classes are generated in a pre-build step.

The next chapter focuses on how to utilize these wrapper classes in the external application.

9. Create and link to variables in the ProcSee RTM

In the previous chapter the wrapper classes for the ProcSee struct definitions were created. This chapter's main focus is on how to create and get access to the instances of these classes in the external application.

The `pdat` variables must be created before they can be accessed in the external application. These variables also needs to be linked to the variables in the RTM. A link means that it's automatically updated in the RTM when it's modified in the external application, and vice versa. This is the default behavior in the .NET API which can be changed by `pragma` directives in the `pdat` file (`#pragma`). We will not describe the `pragma` directives any further in this document because it's out of the scope for this tutorial. For more information about the `pragma` directives, see the "ProcSee database file format" documentation in the ProcSee Reference Manual.

The `ReadScript` method in the class `PcProcess` provides functionality to read the contents of the `pdat` file(s) and create the struct definitions and the variables in the external application. At the same time the links are created from the variables in the external application to the variables in the RTM.

First, add the namespace `KetchupApp.Structs` to the `CProcSee.cs` file, i.e. the namespace of the wrapper classes that we created in the previous chapter.

```
using KetchupApp.Structs;
```

In the attribute section of the class `CProcSee` add the following private attributes. These attributes will be needed in the `CreateVariables` method and when updating the Ketchup process.

```
private PwcSTank mTankTomatoes = null;
private PwcSTank mTankSpices = null;
private PwcSTank mTankVinegar = null;
private PwcSTank mTankMixing = null;
private PwcSValve mValveTomatoes = null;
private PwcSValve mValveSpices = null;
private PwcSValve mValveVinegar = null;
private PcInt32 mSimTime = null;
private PcFloat mTomatoesInMixingTank = null;
private PcFloat mSpicesInMixingTank = null;
private PcFloat mVinegarInMixingTank = null;
```

Now, it's time to start implementing the `CreateVariables` method. First, create the filename where the `pdat` files are located. The ProcSee Ketchup application is located in the `%PROCSEE_DIR%/demos/ketchup/%ARCH%` folder. In the tutorial we use the `Environment.GetEnvironmentVariable` method to get the values of the environment variables. Note that no null pointer check is made on the return value from the `Environment.GetEnvironmentVariable` method. It is assumed that the environment variables `PROCSEE_DIR` and `ARCH` are set before running this external application. The name of the struct and the variable definition files are `RecordDefinitions.pdat` and `Variables.pdat`, respectively.

The class `PcProcess` offers several overloaded `ReadScript` methods. In this tutorial we will be using the `ReadScript` method accepting one argument of type string array, i.e. an array of filenames. This method reads the struct and the variable definitions in the `pdat` files and creates instances of the default wrapper classes (both built-in and generated wrapper classes). The try-catch block surrounding the `ReadScript` methods is needed because this method throws an exception if an unrecoverable error is detected when reading the files.

The `PcProcess` class provides a method called `variable` that returns an instance of a `pdat` variable. It is used in the `CreateVariables` method to initialize the attributes we declared in the `CProcSee` class. This is a generic method returning an error compile time if the type doesn't match the target variable. We advise you to always use the generic method because it provides compile time checks instead of run-time checks as the non-generic method do. The method `variable` returns a pointer of the correct type or a null pointer if the variable doesn't match the variable it is assigned to. The generic argument (in `<>`) is the return type, and the first argument is the name of the variable, which is the name of the variable in the `pdat` file.

Implement the following private method in the `CProcSee` source file.

```

private void CreateVariables()
{
    try
    {
        String path = Environment.GetEnvironmentVariable("PROCSEE_DIR") +
            @"\demos\ketchup\" +
            Environment.GetEnvironmentVariable("ARCH");

        mProcess.ReadScript(new String[] { path + @"\RecordDefinitions.pdat",
            path + @"\Variables.pdat" });

        // Get the variables...
        mTankTomatoes = mProcess.Variable<PwcSTank>("TankTomatoes");
        mTankSpices    = mProcess.Variable<PwcSTank>("TankSpices");
        mTankVinegar   = mProcess.Variable<PwcSTank>("TankVinegar");
        mTankMixing    = mProcess.Variable<PwcSTank>("TankMixing");

        mValveTomatoes = mProcess.Variable<PwcSValve>("ValveTomatoes");
        mValveSpices   = mProcess.Variable<PwcSValve>("ValveSpices");
        mValveVinegar  = mProcess.Variable<PwcSValve>("ValveVinegar");

        mTomatoesInMixingTank = mProcess.Variable<PcFloat>("TomatoesInMixingTank");
        mSpicesInMixingTank   = mProcess.Variable<PcFloat>("SpicesInMixingTank");
        mVinegarInMixingTank  = mProcess.Variable<PcFloat>("VinegarInMixingTank");

        mSimTime = mProcess.Variable<PcInt32>("SimTime");
    }
    catch (PcException ex)
    {
        PrintMessage("CreateVariables error : " + ex.Message);
    }
}

```

Next, we have to call this `CreateVariables` method. Move to the `OnConnect` method in the `CProcSee` class. In the body of this method after the `mRTMState.ForeColor` make a call to the method `CreateVariables`, like:

```

internal void OnConnect(PcRTM rtm)
{
    ...
    mRTMState.ForeColor = Color.Green;
    CreateVariables();
}

```

We are now finished creating and initializing the variables needed to update the Ketchup process.

10. Update and transfer variables to the ProcSee RTM

In this chapter you will learn how to update and transfer variable values to the RTM. The .NET API is implemented so it can run in a multi-threaded environment, but that puts some constraints when it comes to variable updates. Before a thread can start updating variables, it needs to gain ownership of a variable lock, i.e. a write lock. The variable lock is implemented so that only one thread can have ownership of a write lock, while several threads can have access to a read lock. Therefore, it is very important that a thread releases the ownership of

the lock when it is no longer needed, e.g. after the variables have been updated and transferred to the RTM. If not, the external application will sooner or later end up in a deadlock situation.

Now, it's time to change the small ketchup simulator. This is a simple simulator developed in the pTALK language. We will not develop a complete simulator since that is outside the scope of this tutorial. Instead, only parts of this simulator will be modified to utilize variables updated from the external application. In the external application the following variable groups are updated, i.e. the variables controlling the tomatoes, spices, vinegar valves and tanks, plus different control variables, like simulator time. The remaining variables are not used by the simulator in the external application, and are therefore left as is.

Add the following class attributes which is needed by the simulator code.

```
private System.Timers.Timer mTimer = null;

private const int    mTimerInterval = 1000;
private const float mFlowFactor    = 10.0f;
```

The attribute `mTimerInterval` is the simulator speed, i.e. the number of milliseconds between each invocation from the timer. The interval is initially set to once per second. Later on in this tutorial you will learn how to change the simulator speed from a remote function. The `mFlowFactor` is just a factor used in the calculation of the fluid through the valves (pipes).

Locate the `OnConnect` method in the class `CProcSee`. In the body of this method add a call to the method `CreateSimulator` after the call to `CreateVariables`.

```
private void OnConnect()
{
    ...
    CreateVariables();
    CreateSimulator();
}
```

The `CreateSimulator` method creates the timer object which is the engine in the simulator process. Initially, the timer runs periodically invoking the event handler `OnUpdateSimulator` with an interval of 1000 milliseconds

```
private void CreateSimulator()
{
    mTimer = new System.Timers.Timer(mTimerInterval);
    mTimer.Elapsed += new System.Timers.ElapsedEventHandler(OnUpdateSimulator);
    mTimer.Enabled = true;
}
```

The `OnUpdateSimulator` method (invoked by the timer object) calls the `UpdateSimulator` method which updates the simulator process. Notice that the call to the `UpdateSimulator` is surrounded by the static `PcVariableAccess` class methods `BeginWriteValues` and `EndWriteAndSendValues`. The update of the process variables must be surrounded by these calls. Forgetting to do so will throw exceptions in the .NET API. The method `BeginWriteValues` gains the ownership of the write lock, while `EndWriteAndSendValues`

copies the updated variable values into a transfer buffer, sends the values to the RTM and releases the write lock.

```
private void OnUpdateSimulator(object source, System.Timers.ElapsedEventArgs e)
{
    PcVariableAccess.BeginWriteValues();
    try
    {
        UpdateSimulator();
    }
    catch (PcException ex)
    {
        PrintMessage("OnUpdateSimulator: " + ex.Message);
    }
    PcVariableAccess.EndWriteAndSendValues();
}
```

The method `UpdateSimulator` updates the ketchup process. We will not describe in details the simulator code, as it is out of the scope for this tutorial. Just copy the contents of this method to your `CProcSee` class. To summarize, the source code:

- Increments the simulator time.
- Checks if any of the valves are open.
- Calculates the flow through the valves (pipes).
- Checks the contents in the tanks.
- Calculates the contents of the mixing tank.
- Closes the valves if the content of the mixing tank is full.

```

private void UpdateSimulator() throws PcException {

    float  tomatoes      = 0.0f;
    float  spices        = 0.0f;
    float  vinegar       = 0.0f;
    boolean valvesClosed = true;

    mSimTime.value( mSimTime.value() + 1 );

    if (mValveTomatoes.Open() != 0) {
        valvesClosed = false;
        tomatoes      = mValveTomatoes.Flow() / mFlowFactor;
    }

    if (mValveSpices.Open() != 0) {
        valvesClosed = false;
        spices        = mValveSpices.Flow() / mFlowFactor;
    }

    if (mValveVinegar.Open() != 0) {
        valvesClosed = false;
        vinegar       = mValveVinegar.Flow() / mFlowFactor;
    }

    if ( valvesClosed ) return;

    float inT = checkTankLevel(mTankTomatoes, mValveTomatoes, tomatoes);
    float inS = checkTankLevel(mTankSpices,    mValveSpices,    spices);
    float inV = checkTankLevel(mTankVinegar,   mValveVinegar,   vinegar);

    increaseLevelInMixingTank( inT, mTomatoesInMixingTank );
    increaseLevelInMixingTank( inS, mSpicesInMixingTank );
    increaseLevelInMixingTank( inV, mVinegarInMixingTank );
}

```

The CheckTankLevel method performs a validation check on the tank level, i.e. whether the tank level is inside the legal limits or not. If not, the inlet valve is closed. This method is called from UpdateSimulator. It returns the change of the fluid through the valve at a given timestamp.

```

private float CheckTankLevel(PwcSTank tank, PwcSValve valve, float inValue)
{
    float level = tank.Level - inValue;
    if (level < 0.0f)
    {
        inValue = tank.Level;
        tank.Level = 0.0f;
        valve.Open = 0;
    }
    else if (level > tank.MaxLevel)
    {
        inValue = tank.MaxLevel - tank.Level;
        tank.Level = tank.MaxLevel;
    }
    else
    {
        tank.Level = level;
    }

    return inValue;
}

```

The last function required by the ketchup process is `IncreaseLevelInMixingTank`. This function performs a check which closes the inlet valves if the level exceeds the maximum limit in the mixing tank.

```

private void IncreaseLevelInMixingTank( float inValue, PcFloat inTank )
{
    if ((mTankMixing.Level + inValue) > mTankMixing.MaxLevel)
    {
        inValue = mTankMixing.MaxLevel - mTankMixing.Level;
        inTank.Value = inTank.Value + inValue;
        mTankMixing.Level = mTankMixing.MaxLevel;
        mValveTomatoes.Open = 0;
        mValveSpices.Open = 0;
        mValveVinegar.Open = 0;
    }
    else
    {
        inTank.Value = inTank.Value + inValue;
        mTankMixing.Level = mTankMixing.Level + inValue;
    }
}

```

One last modification has to be done to the source code before we can compile and run the external application. Move to the `StartRTM` method and modify the environment variable `KETCHUP_APP_VERSION`. This time change the value from 1 to 2. This is one of the secrets in the Ketchup application that we are not going to describe any further. Compile and run the external application again. This time, the simulator engine is run from the external application.

11. Register remote functions which can be called from the ProcSee RTM

This chapter's focus is to learn how to register a remote function in the external application. A remote function is a function that can be called from the pTALK language in the RTM and executed in the external application. For those of you who are familiar with the native ProcSee "C" API, know how difficult and time consuming it is to create remote functions. It is very easy to make mistakes when unpacking the arguments. This is much easier in the .NET API. It provides something called annotations, which is a special form of syntactic metadata used in the .NET API that can be applied to methods and method parameters. This information is retrieved run-time when registering the remote functions. How this is done is described in detail later in this chapter.

The ketchup demo uses three remote functions which must be implemented in the CProcSee class. The tutorial needs to implement one method to start the simulator, one method to stop the simulator and one method to modify the simulator interval. These methods are more or less self-explanatory. This tutorial will therefore not go into the methods' implementation details. This chapter's focus is instead on how to register the functions and what's needed in the source code so that the .NET API can recognize these methods.

So, how do we register remote functions? The RegisterFunctions method of the class PcRTM accepts one instance argument derived from the class Object. It uses a feature in .NET called reflection which run-time examines the Object instance and detects which method to export as remote functions and which methods to ignore. The methods to export to the RTM must be annotated with the metadata attribute PaRegisterFunction. When the .NET API detects this annotation, it gathers information about the method, i.e. the type of the arguments and the name of the method. These data are then used by the .NET API to create the remote function in the RTM.

The PaRegisterFunction annotation accepts an optional argument, name. This argument provides functionality to change the remote function's default name. If it isn't specified, the default name is got from the C# method. All the remote functions implemented in this tutorial assign a different name than the default. The names of the remote functions in the pTALK language will be StartSimulator, StopSimulator and SetSimulatorInterval, respectively.

There are some constraints when it comes to the data types allowed in remote functions. Only simple data types (int, uint, float, double, short, ...), and arrays of these data types can be used, plus the class string which represents char pointers in ProcSee. However, one class is handled specially when used as a parameter in remote functions, and that's PcRTM. This class is legal only when used as the first argument in a remote function. It isn't exported as an argument to the pTALK function, but provides information about the RTM that called the remote function. This information can be useful when an external application is connected to several RTMs.

The .NET API also provides an annotation which can be applied to method parameters. The name of this annotation is PaParam. This annotation is not used in this tutorial, but is used to specify the size of an array parameter, like:

```
[PaRegisterFunction] private void test( [PaParam( 10 )] int[] arg ) { ... }
```

Now, add the following remote functions to the CProcSee class.

```

[PaRegisterFunction("StartSimulator")]
private void RfStartSimulator()
{
    if (!mTimer.Enabled)
        mTimer.Enabled = true;
}

[PaRegisterFunction("StopSimulator")]
private void RfStopSimulator()
{
    if (mTimer.Enabled)
        mTimer.Enabled = false;
}

[PaRegisterFunction("SetSimulatorInterval")]
private void RfSetSimulatorInterval(int interval)
{
    bool isRunning = mTimer.Enabled;
    if (isRunning)
        mTimer.Enabled = false;
    mTimer.Interval = interval;
    if (isRunning)
        mTimer.Enabled = true;
}

```

Also implement the RegisterRemoteFunctions in the CProcSee class. This method makes a call to the PcRTM class RegisterFunctions to register all the remote functions in the CProcSee class instance (note that the this pointer is passed as argument to this method).

```

private void RegisterRemoteFunctions() {
    mRTM.RegisterFunctions( this );
}

```

Now, we have to make a call to the method RegisterRemoteFunctions. Move to the OnConnect method in the class CProcSee. In the method body make a call to the method RegisterRemoteFunctions after the call to CreateSimulator.

```

internal void OnConnect(PcRTM rtm)
{
    ...
    CreateVariables();
    CreateSimulator();
    RegisterRemoteFunctions();
}

```

As in the previous chapter we have to modify the environment variable KETCHUP_APP_VERSION before we can compile and run the external application. This time change the environment variable from 2 to 3 (in the StartRTM method). Compile and run the external application again. This time, the start, stop and set interval buttons call the remote functions that we just added in the external application.

12. Use the execute function to run pTALK commands in the ProcSee RTM

This chapter focus is on how to send commands to the RTM using the pTALK language from an external application. The .NET API provides two methods that executes a pTALK command in the RTM (available in the classes PcProcess and PcRTM), these are:

- ExecuteAsync
- Execute

The method ExecuteAsync executes a pTALK command and returns to the caller immediately. The .NET API offers several overloaded ExecuteAsync methods which can be used to send a pTALK command to one, several or all connected RTMs. The method Execute executes a pTALK command, waits for the result, and returns the result back to the caller as a string. Notice that the Execute method can be sent only to a single RTM due to the nature of synchronous method calls which returns a result.

In this chapter you will learn how to change the picture's resize mode from the external application. We will use the methods Execute and ExecuteAsync for this purpose. Before we start on the implementation details, we need a small introduction to the resize functionality in ProcSee. A picture in ProcSee has an attribute called resizeMode that controls how it is resized inside the window it's displayed. This attribute provides three resize modes, which are:

- normal size (this is the default resize mode and the one that we have used so far).
- resize the picture to fit inside its window.
- resize the picture to fit inside its window, but maintain the picture's width to height aspect ratio.

In this tutorial we will be developing a menu where these options are available. In the IDE switch to the design view for the CProcSee class. Next, click on the MenuStrip (where the *File* menu is located) which enables an input field. In this input field enter the text *View* and press return. Press the arrow down key on the keyboard and enter the text *Resize*. Next, we will create a sub-menu where the resize options for the picture will be available. Press the tab key and enter the text *Normal Size*. Create the menu items *Fit To Window* and *Maintain Aspect Ratio* in this sub-menu as well. In the properties window change the names of the menu items in the *Resize* sub-menu (Name property). Name the menu items mNormalSizeMenu, mFitToWindowMenu and mMaintainAspectRatioMenu, respectively. At the same time set the property Tag to the values 0, 1 and 2 for the menu items. We will come back to why later on in this chapter. Set the Checked property to true for the mNormalSizeMenu menu item.

Now, we will add the event handler for the menu items. Click on the Events tab in the properties window. Locate the Click action and enter the name of the event handler. Use the same event handler for all the menu items. Give the Click event handler the name OnSetResizeMode. We have now created the menu items.

Now, switch to the source view and locate the call-back method `OnSetResizeMode`. Enter the following code in this method.

```
private void OnSetResizeMode(object sender, EventArgs e)
{
    ToolStripMenuItem item = sender as ToolStripMenuItem;
    mNormalSizeMenu.Checked = false;
    mFitToWindowMenu.Checked = false;
    mMaintainAspectRatioMenu.Checked = false;
    item.Checked = true;

    String mode = item.Tag as String;
    try
    {
        mRTM.ExecuteAsync("::KetchupApp.KetchupProcess",
            "{ resizeMode = " + mode + "; update(); }");
    }
    catch (PcNoConnection ex)
    {
        PrintMessage("OnSetResizeMode: " + ex.Message);
    }
}
```

The property `Tag` in the `ToolStripMenuItem` returns the value that we set for the property `Tag` in the design view, i.e. the resize modes. This string is assigned to the `resizeMode` attribute in the `ExecuteAsync` method. These are the values of the different resize modes in the `ProcSee RTM` which can be applied to the `resizeMode` attribute in the picture. The first argument in the `ExecuteAsync` method is the scope, i.e. `KetchupApp.KetchupProcess`. This is the full qualified name of the picture displayed in the exported window. The next argument to the `ExecuteAsync` method is the `pTALK` command. This command contains the `resizeMode` attribute followed by a call to the update function. The update function is needed to refresh the picture after the resize mode modification.

Finally, we need to initialize the menu item with current resize mode. To get current resize mode we need to use the `Execute` method. The scope argument is the same that we used in the `OnSetResizeMode` method described earlier in this chapter, and the `pTALK` command is `resizeMode`. This `pTALK` command returns the current value of the picture's `resizeMode` attribute. We will not go into the implementation details here, but the source code uses the string method `CompareTo` to check whether the `ToolStripMenuItem`'s `Tag` property matches the `resizeMode` or not.

```
private void SetResizeMode()
{
    try
    {
        String mode = mRTM.Execute("::KetchupApp.KetchupProcess", "resizeMode");
        if (mode != null)
        {
            mNormalSizeMenu.Checked =
                (mode.CompareTo(mNormalSizeMenu.Tag) == 0) ? true : false;
            mFitToWindowMenu.Checked =
                (mode.CompareTo(mFitToWindowMenu.Tag) == 0) ? true : false;
            mMaintainAspectRatioMenu.Checked =
                (mode.CompareTo(mMaintainAspectRatioMenu.Tag) == 0) ? true : false;
        }
    }
}
```

```
    }  
  }  
  catch (PcNoConnection ex)  
  {  
    PrintMessage("SetResizeMode: " + ex.Message);  
  }  
}
```

Locate the `OnConnect` method and call the `SetResizeMode` method after the call to `RegisterRemoteFunctions`.

```
internal void OnConnect(PcRTM rtm)  
{  
    ...  
    CreateSimulator();  
    RegisterRemoteFunctions();  
    SetResizeMode();  
}
```

Now, we are finished with this tutorial. You can now compile and run the external application again. Try to change the resize modes. As you can see the picture is resized differently depending on the selected resize mode.

If you have walked through this tutorial you should now be in a position to create your own external application utilizing the .NET API. This tutorial has addressed the most fundamental aspects of the .NET API. We hope that you now have learned the basic concepts and the fundamental building blocks to develop your own external application using the .NET API.