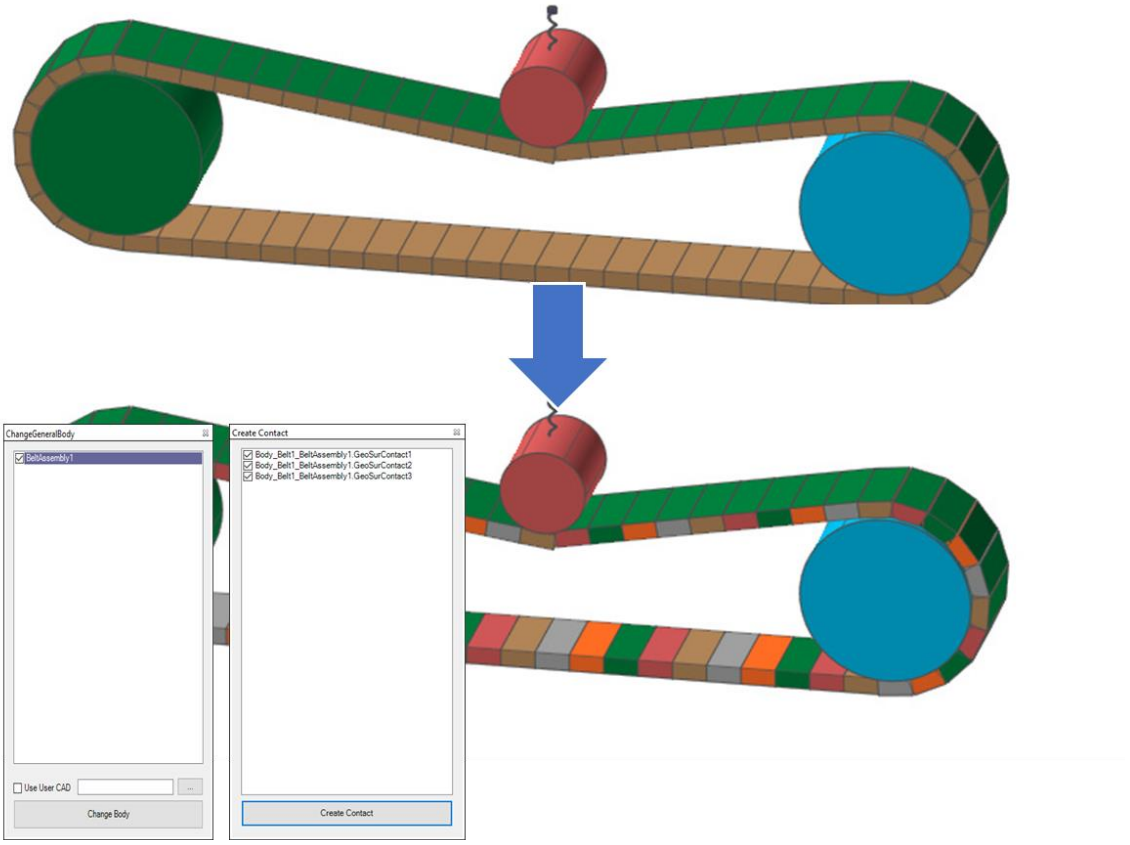
Simple Belt System (ProcessNet VSTA)



☑ BeltAssembly1

☐ Use User CAD

☑ Body_Belt1_BeltAssembly1.GeoSurContact1
☑ Body_Belt1_BeltAssembly1.GeoSurContact2
☑ Body_Belt1_BeltAssembly1.GeoSurContact3

Information described herein is furnished for general information only, is subjected to change without notice, and should not be construed as a warranty or commitment by FunctionBay, Inc. FunctionBay, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

Registered Trademarks of FunctionBay, Inc. or Subsidiary

Edition Note

This document describes the release information of **RecurDyn V9R4**.

Table of Contents

Chapter 1

Overview

Task Objectives

The link body of the belt assembly provided by RecurDyn/Belt is a clone body. So you cannot use general entities such as Contact and Force supported by RecurDyn/Professional, nor can you directly create FFlex Body using **RecurDyn/Mesh**. One of the recommended ways to use them is to convert existing clone bodies into general bodies. In this tutorial, we will use ProcessNet to convert the clone link body of the belt assembly into a general body, creating a Force and a Contact between the bodies.

Prior Learning Requirements

We recommend that you learn all the tutorials for the RecurDyn/Belt Toolkit and all existing ProcessNet tutorials.

Prerequisites

It is assumed that you have already learned or worked on a ProcessNet tutorial and have a basic knowledge of physics.

Task

This tutorial consists of the following tasks. The following table outlines the time required to complete each task.

Task	Duration (minutes)
Starting ProcessNet	5
Clone Body Convert Code	20
Create Contact Code	20
Register DLL	5
Model Analysis	10
Total	60
Task	Duration (minutes)

Estimated Time to Complete

60 minutes

Chapter

2

Starting ProcessNet

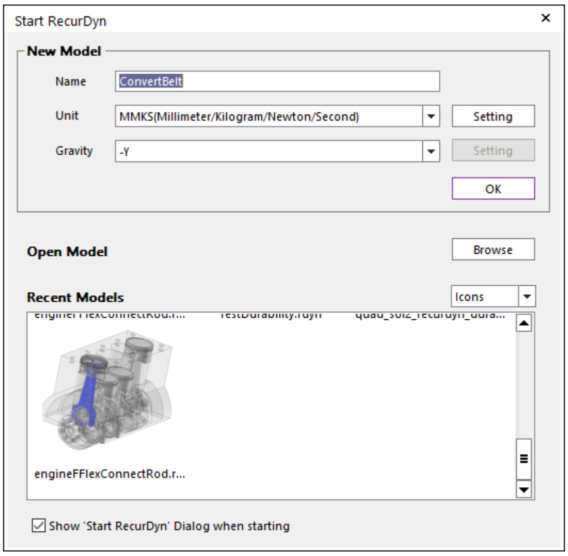
Task Objectives

Take a look at how to start ProcessNet in RecurDyn to use ProcessNet.

Estimated Time to Complete

5 minutes

Starting RecurDyn



Start RecurDyn

New Model

Name: ConvertBelt

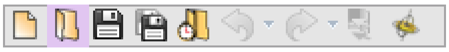
Unit: MMKS(Millimeter/Kilogram/Newton/Second)

Gravity: -Y

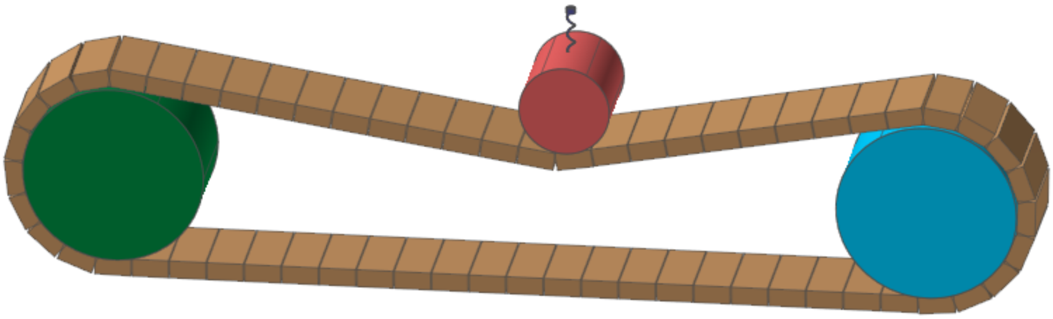
☑ Show 'Start RecurDyn' Dialog when starting

To start RecurDyn:

1. Run **RecurDyn**.
2. When the **Start RecurDyn** dialog window appears, close it as you will need to use an existing model rather than a new one.
3. Click **Open** in the **Quick Access Toolbar**.



4. Select **SimpleBeltAssembly.rdyn**. (Path of the file: <InstallDir>/Help/Tutorial/ProcessNet/VSTA/SimpleBeltSystem)
5. Click **Open**.



To save the model:

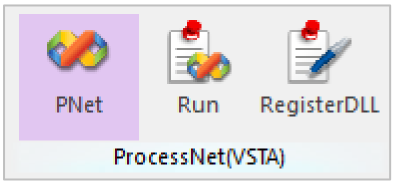
1. In the **File** menu, click **Save As**.
   - Save the file **as SimpleBeltAssembly_01.rdyn**.

   (You cannot perform the simulation if the model is in the tutorial path, so you must save the model in a different path.)

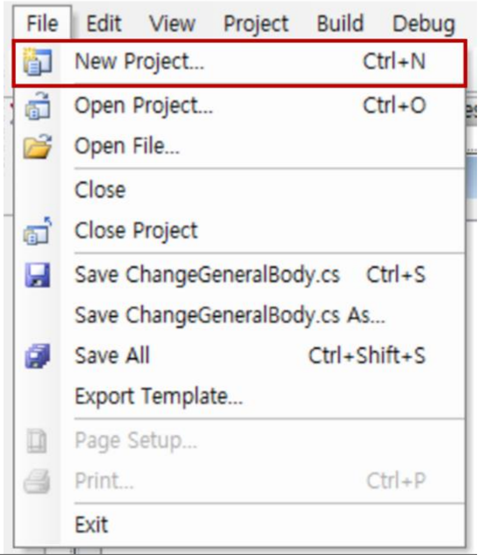
Starting ProcessNet

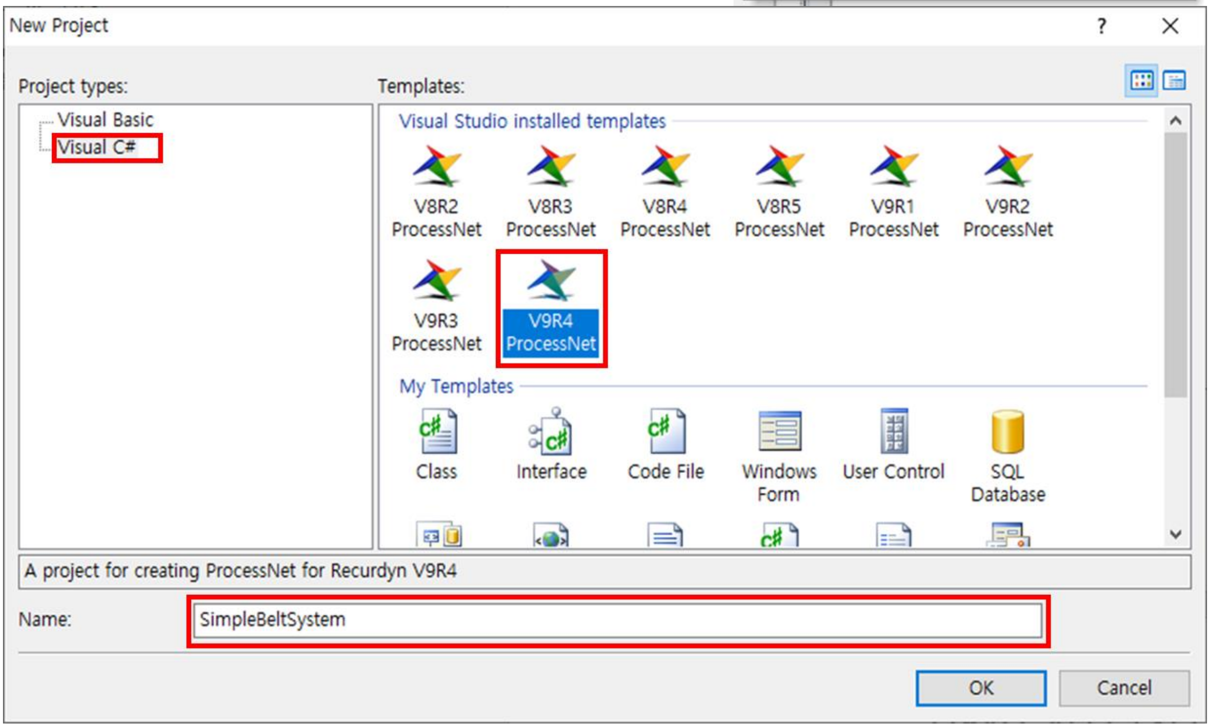
To start and initialize ProcessNet:

1. To open the **ProcessNet** integrated development environment (IDE), on the **Customize tab**, in the **ProcessNet(VSTA)** group, click **PNet**.



2. When the **ProcessNet IDE** starts, open the **File** menu, and then click **New Project**.





A project for creating ProcessNet for Recurdyn V9R4

Name: SimpleBeltSystem

3. When the **New Project** dialog window appears, select the **Template** that corresponds to your version of **RecurDyn**.
   - This tutorial is based on **V9R4.** Therefore, select **V9R4**.

Note: You must use the ProcessNet project that is compatible with your version of RecurDyn. If the version is incorrect, then ProcessNet application may not execute properly. Templates are displayed according to the version of RecurDyn installed.

4. Select **Visual C#** for **Project Types**, enter **SimpleBeltSystem** for Name, and click OK.
5. Once the **SimpleBeltSystem** Project is created, click **File - Save SimpleBeltSystem** and save the ProcessNet project in the desired location.

   Now, you are ready to develop a **ProcessNet** application.

Chapter

3

Change General Body Code

In this chapter, you will create a dialog window that converts a clone link body into a general body, configure its design, and then establish a link between the dialog window and the code.

Task Objectives

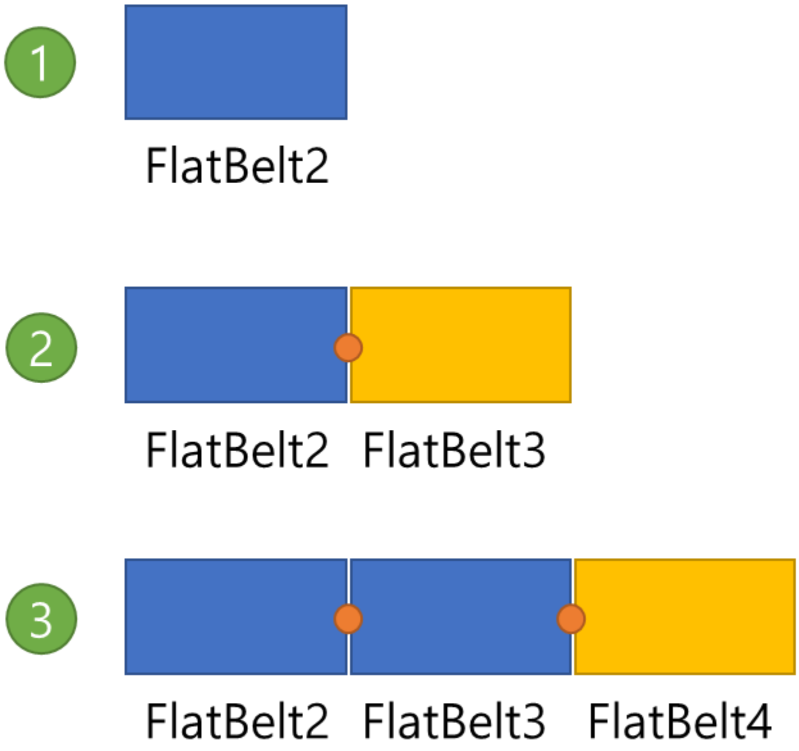
In ProcessNet, learn how to create a dialog window, a function that calls the dialog window, and the code to convert a clone link body into a general body.

Estimated Time to Complete

20 minutes

Configuring Algorithms

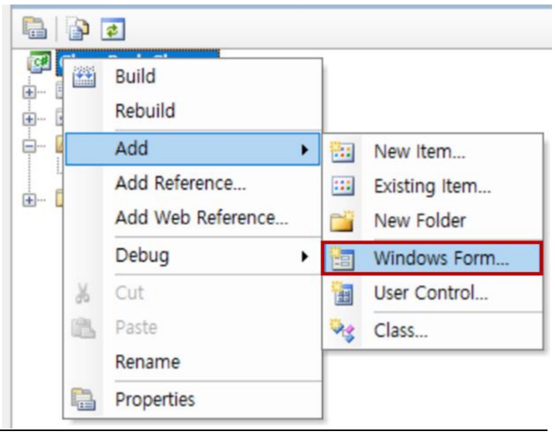
The following shows the algorithm for converting a clone link body in an assembly into a general body and creating a connector between bodies.



1. Convert the first clone body.
2. Convert the second clone body and create a connector between the bodies converted in step 1.
3. Convert the third clone body and create a connector between the bodies converted in step 2.

Creating a Dialog Window

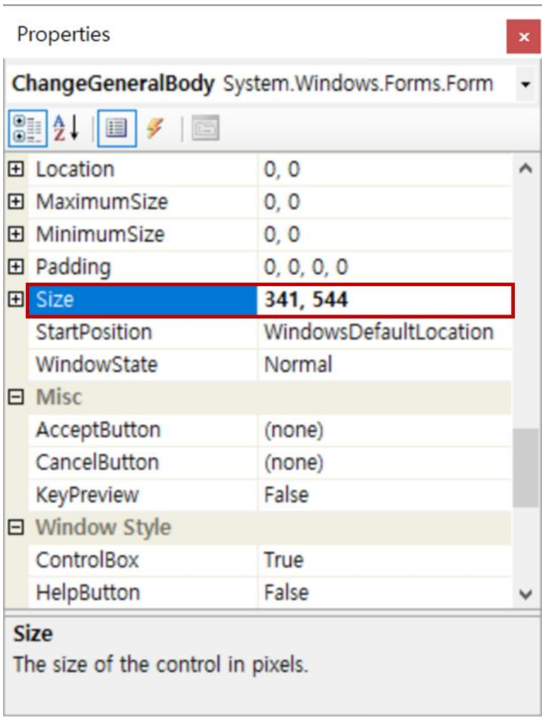
You will learn how to create a **ChangeGeneralBody** dialog window that you can use to implement the above algorithm.



To create a ChangeGeneralBody dialog window:

1. In **ProcessNet IDE**, in the Project Explorer window, right-click **SimpleBeltSystem**.
2. Click **Add – Windows Form**.
3. When the **Add New Item** dialog window appears, click the **Windows Form** icon, and then type **ChangeGeneralBody** in the Name field.

4. Click **Add**.
5. **ChangeGeneralBody.cs[Design],** a design window for Windows Form, appears on the **IDE Project Editor** window.
6. Click the **ChangeGeneralBody** dialog window in the top left corner of the screen.
7. In the bottom right corner of the screen, in **Properties Window**, the information about **ChangeGeneralBody** appears, where you need to set the Size to 341, 544.
8. Also, set **FormBorderStype** to **FixedToolWindow**.
9. Move the cursor over the **ToolBox** Toolbox in the top left corner of the screen. You will see the menu with which you can add a dialog window, button, and other control functions.



Properties

ChangeGeneralBody System.Windows.Forms.Form

⊞ Location	0, 0
⊞ MaximumSize	0, 0
⊞ MinimumSize	0, 0
⊞ Padding	0, 0, 0, 0
⊞ Size	341, 544
StartPosition	WindowsDefaultLocation
WindowState	Normal
⊟ Misc	
AcceptButton	(none)
CancelButton	(none)
KeyPreview	False
⊟ Window Style	
ControlBox	True
HelpButton	False

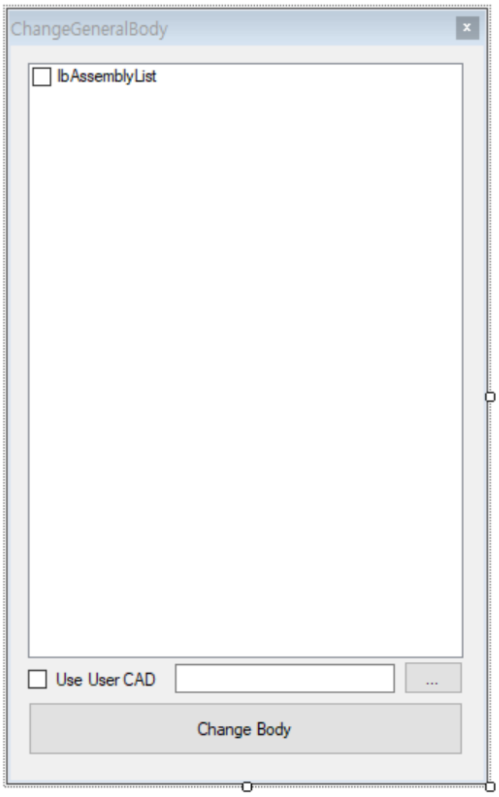
Size
The size of the control in pixels.

Note: If you cannot see the **ToolBox**, click **View-ToolBox** or press Ctrl + Alt + X.

10. In the **Common Controls** list, select **CheckedListBox**, and then drag and drop the Label in the top left corner of the dialog window that you want to design.
11. In the **Properties** window, change Location to 12, 12, and enter 299, 409 for Size.
12. For the **Name**, enter **lbAssemblyList**.
13. For **Button1, Button2, TextBox1**, and **CheckBox1**, change the **Text** and **Name** values using the same procedure as described above by referring to the following table.

Dialog Element	Text	Name	Location	Size
Button1	...	btCADPath	272, 422	41, 23
Button2	Change Body	btChagneBody	12, 451	301, 37
CheckBox1	Use User CAD	cbUseCAD	12, 428	
TextBox1		tbCADPath	113, 425	151, 20

14. After setting all of the values described above, the dialog window should resemble the figure on the right.
15. Click **File – Save ChangeGeneralBody.cs** to save the file.

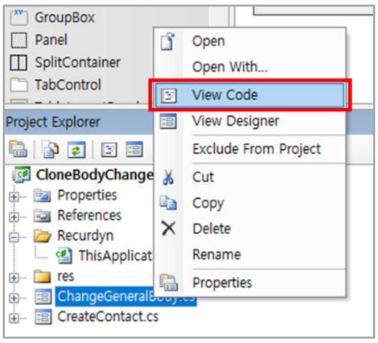


Note: The size or location of the dialog window may differ depending on the PC environment.

Configuring the Initial Settings of the Dialog Window

So far we have configured the appearance of the dialog window. Now, you must add variables to define what values users can enter in the dialog window and the event generated when users click a button. Also, you will learn how to use the ProcessNet function in the dialog window.

To configure the initial settings of the ChangeGeneralBody dialog window:

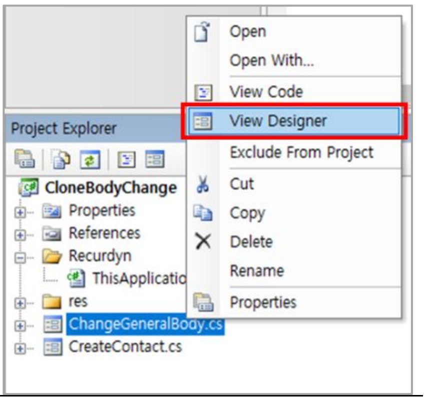


1. In the **Project Explorer**, right-click **ChangeGeneralBody.cs**. In the context menu, click **View Code** to display the source code of **ChangeGeneralBody.cs** in the **Edit IDE Project** window.
2. In the edit window, insert the following code as the variable to use in the dialog window.

```
using System.IO;

using FunctionBay.RecurDyn.ProcessNet;
using FunctionBay.RecurDyn.ProcessNet.BNP;

namespace SimpleBeltSystem
{
        public partial class ChangeGeneralBody : Form
        {
                string path = System.Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)
        + @"\RecurDyn\V9R4";
                bool closeAssembly = true;

                IModelDocument modelDocument;
                ISubSystem sub;
                public ChangeGeneralBody(IModelDocument modelDoc)
                {
                        InitializeComponent();
                        modelDocument = modelDoc;
                        sub = modelDocument.GetDataStorage() as ISubSystem;
                }
        }
}
```

- **FunctionBay.RecurDyn.ProcessNet** and **FunctionBay.RecurDyn.ProcessNet.BNP** are the references for using the ProcessNet functions.
- **path** is the location to save the exported CAD file. In this tutorial, we will save the CAD file in the RecurDyn folder under the Document directory.

3. In the **Project Explorer,** right-click the **ChangeGeneralBody.cs** file, and click **View Designer** to display the dialog window you created in the previous procedure.
4. In the dialog window, double-click the **...** button to create **the btCADPath_Click()** function.
5. Insert the following code in the newly created function.



```
private void btCADPath_Click(object sender, EventArgs e)
{
        OpenFileDialog openFileDlg = new OpenFileDialog();
        openFileDlg.DefaultExt = "db";
        openFileDlg.Filter = "ParaSolid File Files(*.x_t;*.x_b)|*.x_t;*.x_b";
        if (openFileDlg.ShowDialog() == DialogResult.OK)
        {
                this.tbCADPath.Text = openFileDlg.FileName.ToString();
        }
}
```

- If you want to use your own CAD file, open a **Dialog** window for selecting a CAD file.
- Click the **...** button to execute the **btCADPath_Click()** function and open the **Folder Dialog** window.

6. In the **Project Explorer**, right-click **ChangeGeneralBody.cs**, and then click **View Designer**. In the dialog window, double-click the **Change Body** button to create the **btChagneBody_Click()** function.

7. Enter the following code in the **btChangeBody_Click()** function.

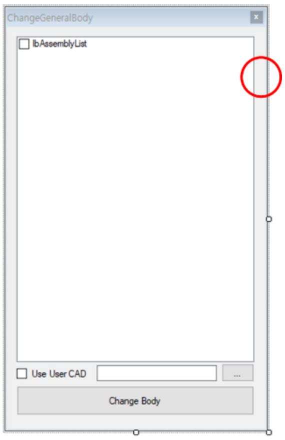
```
private void btChagneBody_Click(object sender, EventArgs e)
{
        bool useUserCAD = this.cbUseCAD.Checked;
        string useUserCADFilePath = this.tbCADPath.Text;

        if (useUserCAD)
        {
                if (File.Exists(useUserCADFilePath) == false)
                {
                        MessageBox.Show("The file does not exist");
                        return;
                }
        }

        if (this.lbAssemblyList.CheckedItems.Count != 0)
        {
                for (int count = 0; count <= this.lbAssemblyList.Items.Count - 1; count++)
                {
                        if (this.lbAssemblyList.GetItemCheckState(count) == CheckState.Checked)
                        {
                                string assemblylistName = this.lbAssemblyList.Items[count].ToString();
                                changeBNPGeneralBody(assemblylistName, useUserCAD,
useUserCADFilePath);
                                this.lbAssemblyList.SetItemChecked(count, false);
                        }
                }
        }
        modelDocument.UpdateDatabaseWindow();
        modelDocument.Redraw();
}
```

- Convert the clone link body into a general body by passing the name of the assembly selected in **lbAssemblyList** and the information on whether User CAD is used or not to the **changeBNPGeneralBody()** function. The **changeBNPGeneralBody()** function will be explained in the next chapter.

8. Once again, in the **Project Explorer**, right-click the **ChangeGeneralBody.cs** file, and click **View Designer**.

9. Double-click an empty area in the dialog window to create the **ChangeGeneralBody_Load()** function.

10. Enter the following code in the **ChangeGeneralBody_Load()** function.



```
private void ChangeGeneralBody_Load(object sender, EventArgs e)
{
        IBNPSubSystem bnpSub = sub.BNPSubSystem as IBNPSubSystem;
        if (bnpSub == null)
                return;

        IBNPAssemblyCollection bnpAssemblies = bnpSub.AssemblyCollection;
        foreach (IBNPAssembly assembly in bnpAssemblies)
        {
                this.lbAssemblyList.Items.Add(assembly.Name);
        }
        IBNPAssembly2DCollection bnp2DAssemies = bnpSub.Assembly2DCollection;
        foreach (IBNPAssembly2D assembly in bnp2DAssemies)
        {
                this.lbAssemblyList.Items.Add(assembly.Name);
        }
}
```

- This function retrieves the belt assembly from the current subsystem when the dialog window opens.
- Since there are two types of belts (2D belt and 3D belt), the collection is made separately.

Creating Body and Connector with Assembly Information

Using **ProcessNet**, you can import clone link body and connector information of the belt assembly. With this information you will learn how to create a general body and a Force.

To convert a body:

You will import the information of a **clone link body** and convert it into a **general body** based on the information.

1. Create the **changeBNPGeneralBody(), copyBeltBodyInfor(), and convertCloneBody()** functions. Exporting a clone body, importing it to a CAD file, and entering the information of the clone link body in the imported general body are repetitive works. So we will create the **copyBeltBodyInfor()** function that copies the clone link body information to a general body and the **convertCloneBody()** function that converts the clone link body into a general body using a CAD file.

```
public void changeBNPGeneralBody(string assemblyName,bool useUserCAD,string useUserCADFilePath)
{
}
private IBody convertCloneBody(IReferenceFrame refFrame1,bool useUserCAD,string
useUserCADFilePath ,IBNPBodyBelt linkBody, string generalBodyName)
{
}
private void copyBeltBodyInfor(IBNPBodyBelt BeforeBody, IBody AfterBody)
{
}
```

2. Create the **changeBNPGeneralBody()** function. Copy the following code and insert it into the function.

```
public void changeBNPGeneralBody(string assemblyName,bool useUserCAD,string useUserCADFilePath)
{
          IReferenceFrame refFrame1 = modelDocument.CreateReferenceFrame();
          IBNPAssembly assembly = sub.GetEntity(assemblyName) as IBNPAssembly;
          IBNPBodyBeltCollection linkBodies = null;
          IBNPAssembly2D assembly2D = null;


          IGroupGeneral genearlBodyGroup = sub.CreateGroupGeneral("Body_" + sub.Name + "_" +
assemblyName, new object[] { });
          IGroupGeneral genearlForceGroup = sub.CreateGroupGeneral("Connector_" + sub.Name + "_" +
assemblyName, new object[] { });
          IGroupGeneral genearlJointGroup = null;

          if (assembly != null)
          {
                    linkBodies = assembly.BNPBodyBeltCollection as IBNPBodyBeltCollection;
          }
          else
          {
                    assembly2D = sub.GetEntity(assemblyName) as IBNPAssembly2D;
                    linkBodies = assembly2D.BNPBodyBeltCollection as IBNPBodyBeltCollection;
                    genearlJointGroup = sub.CreateGroupGeneral("Joint_" + sub.Name + "_" + assemblyName,
new object[] { });
          }

          int linkCount = linkBodies.Count;


          IBody generalBodyfirst = convertCloneBody(refFrame1,useUserCAD, useUserCADFilePath,linkBodies[0],
"ImportedBody1");
          genearlBodyGroup.AddEntities(new object[] { generalBodyfirst });

          Dictionary<string, string> generalLinkNames = new Dictionary<string, string>();
          generalLinkNames.Add(linkBodies[0].Name, generalBodyfirst.Name);

          for (int iCount = 1; iCount < linkBodies.Count; iCount++)
          {
                    IBody generalBody01 = convertCloneBody(refFrame1, useUserCAD, useUserCADFilePath,
linkBodies[iCount], "ImportedBody" + (iCount + 1).ToString());
                    IMarker generalBody01Marker = generalBody01.GetEntity("CMMarker1") as IMarker;
                    IBody generalBody02 = sub.GetEntity("ImportedBody" + iCount.ToString()) as IBody;
                    IMarker generalBody02Marker = null;
                    if (generalBody02.Name == "ImportedBody1")
                    {
                              generalBody02Marker = generalBody02.GetEntity("CMMarker1") as IMarker;
                    }
                    else
                    {
                              generalBody02Marker = generalBody02.GetEntity("CMMarker2") as IMarker;
                    }
                    if (assembly2D == null)
                    {
                              IForceMatrix matrixForce = createMatrixForce(assembly.Name + "_Matrix_" + "_"
+ linkBodies[iCount].Name.ToString(), assembly, generalBody02, generalBody01,
generalBody01Marker.RefFrame, generalBody02Marker, generalBody01Marker);
                    }
                    else
                    {
                              IForceBushing busingForce = create2DBeltBusingForce(assembly.Name +
"_Busing_" + "_" + linkBodies[iCount].Name.ToString(), assembly2D, generalBody02, generalBody01,
generalBody01Marker.RefFrame, generalBody02Marker, generalBody01Marker);
                    }
                    if (iCount == linkCount - 1 && closeAssembly)
                    {
                              IMarker generalBodyFirstMarker = generalBodyfirst.GetEntity("CMMarker2") as
IMarker;
```

```
                              IMarker generalBody01Marker02 = generalBody01.GetEntity("CMMarker2") as
IMarker;
                              if (assembly2D == null)
                              {
                                        IForceMatrix matrixForceFinal = createMatrixForce(assembly.Name +
"_Matrix_" + "_" + linkBodies[0].Name.ToString(), assembly, generalBody01, generalBodyfirst,
generalBody01Marker02.RefFrame, generalBody01Marker02, generalBodyFirstMarker);
                              }
                              else
                              {
                                        IForceBushing busingForceFinal =
create2DBeltBusingForce(assembly.Name + "_Busing_" + "_" + linkBodies[0].Name.ToString(), assembly2D,
generalBody01, generalBodyfirst, generalBody01Marker02.RefFrame, generalBody01Marker02,
generalBodyFirstMarker);
                              }

                    }
                    genearlBodyGroup.AddEntities(new object[] { generalBody01 });
                    generalLinkNames.Add(linkBodies[iCount].Name, generalBody01.Name);
          }

          modelDocument.SetUndoHistory("BNP Body Change");
          modelDocument.DeleteEntity(linkBodies[0]);

          foreach (KeyValuePair<string, string> generalLinkName in generalLinkNames)
          {
                    IBody linkBody = sub.GetEntity(generalLinkName.Value) as IBody;
                    linkBody.Name = generalLinkName.Key;

          }
          modelDocument.SetUndoHistory("Delete Assembly");

}
```

A description of the **changeBNPGeneralBody()** function is provided below.

```
IReferenceFrame refFrame1 = modelDocument.CreateReferenceFrame();
IBNPAssembly assembly = sub.GetEntity(assemblyName) as IBNPAssembly;
IBNPBodyBeltCollection linkBodies = null;
IBNPAssembly2D assebly2D = null;
```

The code above declares a variable to use in the changeBNPGeneralBody() function.

- **assembly**: It casts the belt assembly in the subsystem using the GetEntity function. It uses the as operator and returns and enters Null if the belt assembly is not in the subsystem.
- **linkBodies**: It saves the information of the link bodies included in the belt assembly.
- **assembly2D**: Declare it in case that the belt assembly is 2D.

```
IGroupGeneral genearlBodyGroup = sub.CreateGroupGeneral("Body_" + sub.Name + "_" + assemblyName, new
object[] { });
IGroupGeneral genearlForceGroup = sub.CreateGroupGeneral("Connector_" + sub.Name + "_" +
assemblyName, new object[] { });
IGroupGeneral genearlJointGroup = null;
```

The code above creates a general group that will bind the Body and Force created with the information of the body and connector contained in the assembly.

```
if (assembly != null)
{
      linkBodies = assembly.BNPBodyBeltCollection as IBNPBodyBeltCollection;
}
else
{
      assebly2D = sub.GetEntity(assemblyName) as IBNPAssembly2D;
      linkBodies = assebly2D.BNPBodyBeltCollection as IBNPBodyBeltCollection;
      genearlJointGroup = sub.CreateGroupGeneral("Joint_" + sub.Name + "_" + assemblyName, new object[]
{ });
}
```

The code above is a conditional statement to judge whether the belt assembly is 2D or 3D. Since we used the as keyword when declaring an assembly, it judges whether the assembly is 2D or 3D by utilizing the fact that Null is returned if the assembly is not 3D. In the case of a 2D assembly, create an additional general group to tie the joints together.

```
IBody generalBodyfirst =convertCloneBody(refFrame1, useUserCAD, useUserCADFilePath, linkBodies[0],
"ImportedBody1");
genearlBodyGroup.AddEntities(new object[] { generalBodyfirst });

Dictionary<string, string> generalLinkNames = new Dictionary<string, string>();
generalLinkNames.Add(linkBodies[0].Name, generalBodyfirst.Name);

for (int iCount = 1; iCount < linkBodies.Count; iCount++)
{
}
```

The code above converts the clone link body belonging to the belt assembly into a general body.

- Convert the first clone link body into a general body using the **convertCloneBody()** function. Add the converted body to the general group. In addition, to convert other clone link bodies, create a loop statement. Since we changed the first clone body, the function should be executed repeatedly from 1 to the number of link bodies.
- **generalLinkNames**: To match the name of the imported CAD file with the link body name, save the CAD body name and link body name in the dictionary.

```
IBody generalBody01 = convertCloneBody(refFrame1, useUserCAD, useUserCADFilePath, linkBodies[iCount],
"ImportedBody" + (iCount + 1).ToString());
IMarker generalBody01Marker = generalBody01.GetEntity("CMMarker1") as IMarker;
IBody generalBody02 = sub.GetEntity("ImportedBody" + iCount.ToString()) as IBody;
IMarker generalBody02Marker = null;
if (generalBody02.Name == "ImportedBody1")
{
            generalBody02Marker = generalBody02.GetEntity("CMMarker1") as IMarker;
}
else
{
            generalBody02Marker = generalBody02.GetEntity("CMMarker2") as IMarker;
}
//Create Connector
if (assebly2D == null)
{
            IForceMatrix matrixForce = createMatrixForce(assembly.Name + "_Matrix_" + "_" +
linkBodies[iCount].Name.ToString(), assembly, generalBody02, generalBody01, generalBody01Marker.RefFrame,
generalBody02Marker, generalBody01Marker);
}
else
{
            IForceBushing busingForce = create2DBeltBusingForce(assembly.Name + "_Busing_" + "_" +
linkBodies[iCount].Name.ToString(), assebly2D, generalBody02, generalBody01,
generalBody01Marker.RefFrame, generalBody02Marker, generalBody01Marker);
}
if (iCount == linkCount - 1 && closeAssembly)
{
            IMarker generalBodyFirstMarker = generalBodyfirst.GetEntity("CMMarker2") as IMarker;
            IMarker generalBody01Marker02 = generalBody01.GetEntity("CMMarker2") as IMarker;
            if (assebly2D == null)
            {
                        IForceMatrix matrixForceFinal = createMatrixForce(assembly.Name + "_Matrix_" + "_" +
linkBodies[0].Name.ToString(), assembly, generalBody01, generalBodyfirst, generalBody01Marker02.RefFrame,
generalBody01Marker02, generalBodyFirstMarker);
            }
            else
            {
                        IForceBushing busingForceFinal = create2DBeltBusingForce(assembly.Name + "_Busing_" +
"_" + linkBodies[0].Name.ToString(), assebly2D, generalBody01, generalBodyfirst,
generalBody01Marker02.RefFrame, generalBody01Marker02, generalBodyFirstMarker);
            }

}
genearlBodyGroup.AddEntities(new object[] { generalBody01 });
generalLinkNames.Add(linkBodies[iCount].Name, generalBody01.Name);
```

The above code is what is entered in the loop statement.

- Starting from the second link body, create a general body and also a Force.
- Declare a general body converted from the link body, the marker created in the body, the previously converted general body, and the marker created in it as generalBody01, generalBody02, generalBody01Marker, and generalBody02Marker. If the name of the general body is "ImportedBody1", the position of the TMarker is different from other bodies, so you must declare them in the opposite way.
- If BeltAssembly is 2D, use Bushing Force and, if 3D, use Matrix Force to create a connector.
- If the belt system is a closed loop, the last link body needs to create the general body converted first and the connector.

```
modelDocument.SetUndoHistory("BNP Body Change");
modelDocument.DeleteEntity(linkBodies[0]);
```

The code above creates an Undo history and deletes the assembly after the loop statement. This can reduce abnormal termination of Undo in RecurDyn UI.

```
foreach (KeyValuePair<string, string> generalLinkName in generalLinkNames)
{
            IBody linkBody = sub.GetEntity(generalLinkName.Value) as IBody;
            linkBody.Name = generalLinkName.Key;
}
modelDocument.SetUndoHistory("Delete Assembly");
```

The code above changes the name of the created CAD body to Link Body.

3. Create the **convertCloneBody**() function. Copy the following code and insert it into the function.

```
private IBody convertCloneBody(IReferenceFrame refFrame1,bool useUserCAD,string
useUserCADFilePath ,IBNPBodyBelt linkBody, string generalBodyName)
{
            string cadFileName = "CloneBody1.x_t";
            if (useUserCAD)
            {
                        cadFileName = useUserCADFilePath;
            }
            else
            {
                        linkBody.GeneralBody.FileExport(path + @"\" + cadFileName, true);
            }
            IBody generalBody = sub.CreateBodyGeneral(generalBodyName);
            generalBody.FileImport(path + @"\" + cadFileName);
            copyBeltBodyInfor(linkBody, generalBody);

            IMarkerCollection dummyMarkers = linkBody.GeneralBody.MarkerCollection;
            List<IMarker> linkMarkers = new List<IMarker>();
            foreach (IMarker marker in dummyMarkers)
            {
                        if (marker.Name.Contains("TMarker"))
                        {
                                    linkMarkers.Add(marker);
                        }
            }
            if (linkMarkers.Count == 1 && generalBodyName == "ImportedBody1")
            {
                        closeAssembly = false;
            }
            int markerCount = 0;
            IMarker[] generalLinkMarkers = new IMarker[2];
            foreach (IMarker linkMarker in linkMarkers)
            {
                        EulerAngle eulerType;
                        double[] eulerAngles = { 0, 0, 0 };
                        linkMarker.RefFrame.GetEulerAngleDegree(out eulerType, out eulerAngles[0], out
eulerAngles[1], out eulerAngles[2]);
                        refFrame1.SetOrigin(linkMarker.RefFrame.Origin.x, linkMarker.RefFrame.Origin.y,
linkMarker.RefFrame.Origin.z);
                        refFrame1.SetEulerAngleDegree(eulerType, eulerAngles[0], eulerAngles[1], eulerAngles[2]);
                        generalLinkMarkers[markerCount] = generalBody.CreateMarker("CMMarker" + (markerCount
+ 1).ToString(), refFrame1);
                        markerCount++;
            }

            return generalBody;
}
```

- Export the clone link body to the CAD file and import the exported CAD file to convert the clone body into a general body.

A description of the **convertCloneBody()** function is provided below.

```
string cadFileName = "CloneBody1.x_t";
if (useUserCAD)
{
            cadFileName = useUserCADFilePath;
}
else
{
            linkBody.GeneralBody.FileExport(path + @"\" + cadFileName, true);
}
IBody generalBody = sub.CreateBodyGeneral(generalBodyName);
generalBody.FileImport(path + @"\" + cadFileName);
copyBeltBodyInfor(linkBody, generalBody);
```

The above code uses the CAD file if the user selects the **User CAD** option, and exports and re-imports the clone link body entered if the option is not selected. Copy the clone link body information using the **copyBeltBodyInfor()** function.

```
IMarkerCollection dummyMarkers = linkBody.GeneralBody.MarkerCollection;
List<IMarker> linkMarkers = new List<IMarker>();
foreach (IMarker marker in dummyMarkers)
{
            if (marker.Name.Contains("TMarker"))
            {
                        linkMarkers.Add(marker);
            }
}
if (linkMarkers.Count == 1 && generalBodyName == "ImportedBody1")
{
            closeAssembly = false;
}
```

The above code saves the TMarker information of the clone link body. TMarker indicates the position of the connector between clone link bodies. If the number of TMarkers in a link is 1, the belt assembly is an open loop, not a closed loop.

```
int markerCount = 0;
IMarker[] generalLinkMarkers = new IMarker[2];
foreach (IMarker linkMarker in linkMarkers)
{
            EulerAngle eulerType;
            double[] eulerAngles = { 0, 0, 0 };
            linkMarker.RefFrame.GetEulerAngleDegree(out eulerType, out eulerAngles[0], out eulerAngles[1], out
eulerAngles[2]);
            refFrame1.SetOrigin(linkMarker.RefFrame.Origin.x, linkMarker.RefFrame.Origin.y,
linkMarker.RefFrame.Origin.z);
            refFrame1.SetEulerAngleDegree(eulerType, eulerAngles[0], eulerAngles[1], eulerAngles[2]);
            generalLinkMarkers[markerCount] = generalBody.CreateMarker("CMMarker" + (markerCount +
1).ToString(), refFrame1);
                markerCount++;
}
```

The code above creates a marker in the general body.

4. Create the **copyBeltBodyInfor()** function to copy the information of the clone link body to the general body. Enter the following code.

```
private void copyBeltBodyInfor(IBNPBodyBelt BeforeBody, IBody AfterBody)
{
        AfterBody.RefFrame.Origin.x = BeforeBody.GeneralBody.RefFrame.Origin.x;
        AfterBody.RefFrame.Origin.y = BeforeBody.GeneralBody.RefFrame.Origin.y;
        AfterBody.RefFrame.Origin.z = BeforeBody.GeneralBody.RefFrame.Origin.z;

        AfterBody.RefFrame.EulerAngle.Type = BeforeBody.GeneralBody.RefFrame.EulerAngle.Type;
        AfterBody.RefFrame.EulerAngle.Alpha.Value =
BeforeBody.GeneralBody.RefFrame.EulerAngle.Alpha.Value;
        AfterBody.RefFrame.EulerAngle.Beta.Value = BeforeBody.GeneralBody.RefFrame.EulerAngle.Beta.Value;
        AfterBody.RefFrame.EulerAngle.Gamma.Value =
BeforeBody.GeneralBody.RefFrame.EulerAngle.Gamma.Value;
        string strMaterialType = BeforeBody.GeneralBody.MaterialInput.ToString();
        if (strMaterialType == "Density")
        {
                AfterBody.MaterialInput = BeforeBody.GeneralBody.MaterialInput;
                if (BeforeBody.GeneralBody.Density2.ParametricValue == null)
                {
                        AfterBody.Density2.Value = BeforeBody.GeneralBody.Density2.Value;
                }
                else
                {
                        AfterBody.Density2.ParametricValue =
BeforeBody.GeneralBody.Density2.ParametricValue;
                }
        }
        else if (strMaterialType == "UserInput")
        {
                AfterBody.MaterialInput = BeforeBody.GeneralBody.MaterialInput;
                if (BeforeBody.GeneralBody.Mass.ParametricValue == null)
                {
                        AfterBody.Mass.Value = BeforeBody.GeneralBody.Mass.Value;
                }
                else
                {
                        AfterBody.Mass.ParametricValue = BeforeBody.GeneralBody.Mass.ParametricValue;
                }

                if (BeforeBody.GeneralBody.Ixx.ParametricValue == null)
                {
                        AfterBody.Ixx.Value = BeforeBody.GeneralBody.Ixx.Value;
                }
                else
                {
                        AfterBody.Ixx.ParametricValue = BeforeBody.GeneralBody.Ixx.ParametricValue;
                }
                AfterBody.Mass.ParametricValue = BeforeBody.GeneralBody.Mass.ParametricValue;

        if (BeforeBody.GeneralBody.Ixx.ParametricValue == null)
        {
                AfterBody.Ixx.Value = BeforeBody.GeneralBody.Ixx.Value;
        }
        else
        {
                AfterBody.Ixx.ParametricValue = BeforeBody.GeneralBody.Ixx.ParametricValue;
        }

        if (BeforeBody.GeneralBody.Ixy.ParametricValue == null)
        {
                AfterBody.Ixy.Value = BeforeBody.GeneralBody.Ixy.Value;
        }
        else
        {
                AfterBody.Ixy.ParametricValue = BeforeBody.GeneralBody.Ixy.ParametricValue;
        }

        if (BeforeBody.GeneralBody.Iyy.ParametricValue == null)
        {
                AfterBody.Iyy.Value = BeforeBody.GeneralBody.Iyy.Value;
        }
        else
        {
                AfterBody.Iyy.ParametricValue = BeforeBody.GeneralBody.Iyy.ParametricValue;
        }

        if (BeforeBody.GeneralBody.Iyz.ParametricValue == null)
        {
                AfterBody.Iyz.Value = BeforeBody.GeneralBody.Iyz.Value;
        }
        else
        {
                AfterBody.Iyz.ParametricValue = BeforeBody.GeneralBody.Iyz.ParametricValue;
        }

                if (BeforeBody.GeneralBody.Izx.ParametricValue == null)
                {
                        AfterBody.Izx.Value = BeforeBody.GeneralBody.Izx.Value;
                }
                else
                {
                        AfterBody.Izx.ParametricValue = BeforeBody.GeneralBody.Izx.ParametricValue;
                }

                if (BeforeBody.GeneralBody.Izz.ParametricValue == null)
                {
                        AfterBody.Izz.Value = BeforeBody.GeneralBody.Izz.Value;
                }
                else
                {
                        AfterBody.Izz.ParametricValue = BeforeBody.GeneralBody.Izz.ParametricValue;
                }

        }
        else if (strMaterialType == "DefaultMaterial")
        {
                AfterBody.MaterialInput = BeforeBody.GeneralBody.MaterialInput;
                AfterBody.Material = BeforeBody.GeneralBody.Material;
        }
        else if (strMaterialType == "UserMaterial")
        {
                AfterBody.MaterialInput = BeforeBody.GeneralBody.MaterialInput;
                AfterBody.Material = BeforeBody.GeneralBody.Material;
        }

}
```

- Enter the body information of the clone link in the general body.

To create a connector:

1. Create the **createMatrixForce()** function. Belt Assembly 3D uses Matrix Force as a connector, so use Matrix Force to connect between bodies.

```
private IForceMatrix createMatrixForce(string matrixName, IBNPAssembly assembly, IBody ActionBody, IBody
BaseBody, IReferenceFrame refFrame, IMarker ActionMarker, IMarker BaseMarker)
{
        double BusingForceMarkerX;
        double BusingForceMarkerY;
        double BusingForceMarkerZ;

        EulerAngle eulerType;
        double[] eulerAngles = { 0, 0, 0 };
```

```
    IBNPAssemblyConnectingForceParameter assConForce = assembly.ConnectingForceParameter;
    IForceMatrix matrixForce = sub.CreateForceMatrix(matrixName, BaseBody, ActionBody, refFrame,
refFrame);
    BaseMarker.RefFrame.GetOrigin(out BusingForceMarkerX, out BusingForceMarkerY, out
BusingForceMarkerZ);
    BaseMarker.RefFrame.GetEulerAngleDegree(out eulerType, out eulerAngles[0], out eulerAngles[1], out
eulerAngles[2]);
    matrixForce.BaseMarker.RefFrame.SetOrigin(BusingForceMarkerX, BusingForceMarkerY,
BusingForceMarkerZ);
    matrixForce.BaseMarker.RefFrame.SetEulerAngleDegree(eulerType, eulerAngles[0], eulerAngles[1],
eulerAngles[2]);

    ActionMarker.RefFrame.GetOrigin(out BusingForceMarkerX, out BusingForceMarkerY, out
BusingForceMarkerZ);
    ActionMarker.RefFrame.GetEulerAngleDegree(out eulerType, out eulerAngles[0], out eulerAngles[1], out
eulerAngles[2]);
    matrixForce.ActionMarker.RefFrame.SetOrigin(BusingForceMarkerX, BusingForceMarkerY,
BusingForceMarkerZ);
    matrixForce.ActionMarker.RefFrame.SetEulerAngleDegree(eulerType, eulerAngles[0], eulerAngles[1],
eulerAngles[2]);
    for (int iCount = 0; iCount < 6; iCount++)
    {
        for (int jCount = 0; jCount < 6; jCount++)
        {
            if (assConForce.StiffnessMatrix(iCount, jCount).ParametricValue == null)
            {
                matrixForce.StiffnessMatrix(iCount, jCount).Value =
assConForce.StiffnessMatrix(iCount, jCount).Value;
            }
            else
            {
                matrixForce.StiffnessMatrix(iCount, jCount).ParametricValue =
assConForce.StiffnessMatrix(iCount, jCount).ParametricValue;
            }
        }
    } for (int iCount = 0; iCount < 6; iCount++)
    {
        if (assConForce.Preload(iCount).ParametricValue == null)
        {
            matrixForce.Preload(iCount).Value = assConForce.Preload(iCount).Value;
        }
        else
        {
            matrixForce.Preload(iCount).ParametricValue =
assConForce.Preload(iCount).ParametricValue;
        }
    }
    for (int iCount = 0; iCount < 6; iCount++)
    {
        if (assConForce.ReferenceLength(iCount).ParametricValue == null)
        {
            matrixForce.ReferenceLength(iCount).Value =
assConForce.ReferenceLength(iCount).Value;
        }
        else
        {
            matrixForce.ReferenceLength(iCount).ParametricValue =
assConForce.ReferenceLength(iCount).ParametricValue;
        }
    }

    return matrixForce;
}
```

2. Create the **create2DBeltBusingForce()** function. Belt Assembly 2D uses Bushing Force as a connector, so use Bushing Force to connect between bodies.

```
private IForceBushing create2DBeltBusingForce(string BusingName, IBNPAssembly2D assembly, IBody ActionBody,
IBody BaseBody, IReferenceFrame refFrame, IMarker ActionMarker, IMarker BaseMarker)
{
        double BusingForceMarkerX;
        double BusingForceMarkerY;
        double BusingForceMarkerZ;

        EulerAngle eulerType;
        double[] eulerAngles = { 0, 0, 0 };

        IBNPAssembly2DBushingForceParameter assemblyBushingForce = assembly.BusingForceParameter;

        IForceBushing BusingForce = sub.CreateForceBushing(BusingName, BaseBody, ActionBody, refFrame);
        BaseMarker.RefFrame.GetOrigin(out BusingForceMarkerX, out BusingForceMarkerY, out
BusingForceMarkerZ);
        BaseMarker.RefFrame.GetEulerAngleDegree(out eulerType, out eulerAngles[0], out eulerAngles[1], out
eulerAngles[2]);
        BusingForce.BaseMarker.RefFrame.SetOrigin(BusingForceMarkerX, BusingForceMarkerY,
BusingForceMarkerZ);
        BusingForce.BaseMarker.RefFrame.SetEulerAngleDegree(eulerType, eulerAngles[0], eulerAngles[1],
eulerAngles[2]);

        ActionMarker.RefFrame.GetOrigin(out BusingForceMarkerX, out BusingForceMarkerY, out
BusingForceMarkerZ);
        ActionMarker.RefFrame.GetEulerAngleDegree(out eulerType, out eulerAngles[0], out eulerAngles[1], out
eulerAngles[2]);
        BusingForce.ActionMarker.RefFrame.SetOrigin(BusingForceMarkerX, BusingForceMarkerY,
BusingForceMarkerZ);
        BusingForce.ActionMarker.RefFrame.SetEulerAngleDegree(eulerType, eulerAngles[0], eulerAngles[1],
eulerAngles[2]);

        BusingForce.UseRadial = true;

        BusingForce.RotationalDampingZ.Coefficient.Value =
assemblyBushingForce.RotationalDampingZ.Coefficient.Value;
        BusingForce.RotationalStiffnessZ.Coefficient.Value =
assemblyBushingForce.RotationalStiffnessZ.Coefficient.Value;
        BusingForce.RotationalPreloadZ.Value = assemblyBushingForce.RotationalPreloadZ.Value;

        if (assemblyBushingForce.RotationalDampingZ.UseExponentValue)
        {
                BusingForce.RotationalDampingZ.UseExponentValue = true;
                BusingForce.RotationalDampingZ.ExponentValue.Value =
assemblyBushingForce.RotationalDampingZ.ExponentValue.Value;
        }

        if (assemblyBushingForce.RotationalStiffnessZ.UseExponentValue)
        {
                BusingForce.RotationalStiffnessZ.UseExponentValue = true;
                BusingForce.RotationalStiffnessZ.ExponentValue.Value =
assemblyBushingForce.RotationalStiffnessZ.ExponentValue.Value;
        }


        BusingForce.TranslationalDampingX.Coefficient.Value =
assemblyBushingForce.TranslationalDampingX.Coefficient.Value;
        BusingForce.TranslationalDampingY.Coefficient.Value =
assemblyBushingForce.TranslationalDampingY.Coefficient.Value;

        BusingForce.TranslationalStiffnessX.Coefficient.Value =
assemblyBushingForce.TranslationalStiffnessX.Coefficient.Value;
        BusingForce.TranslationalStiffnessY.Coefficient.Value =
assemblyBushingForce.TranslationalStiffnessY.Coefficient.Value;

        return BusingForce;
}
```

3.

Chapter

4

Create Contact Code

Task Objectives

In **ProcessNet**, learn how to create a dialog window, a function that calls a dialog window, and the code to create a Contact.

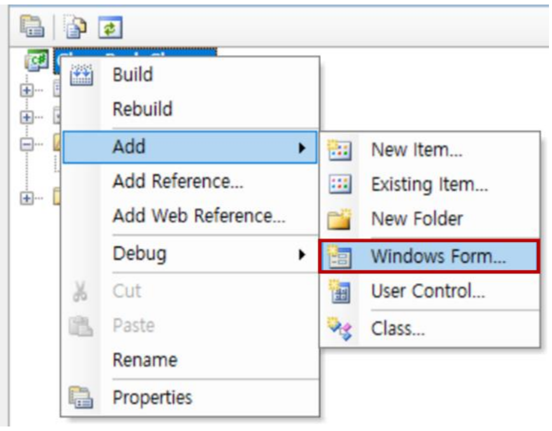
Estimated Time to Complete

20 minutes

Creating a Dialog Window

Once you define a Contact between a link body and a pulley, you will learn how to create a dialog window that creates a Contact between the body group and the pulley you created in Chapter 3 based on the Contact information you defined.

To create a CreateContact dialog window:

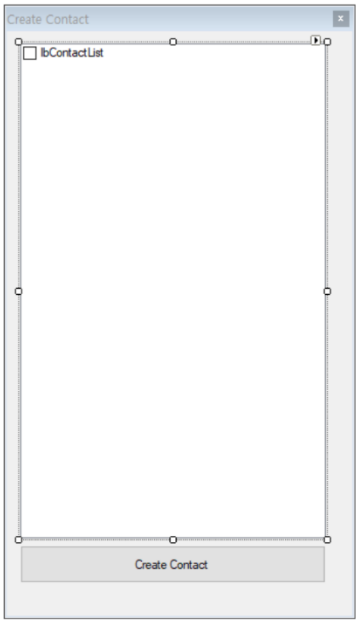


1. In **ProcessNet IDE**, in the Project Explorer window, **right-click SimpleBeltSystem**.
2. Click **Add – Windows Form**.
3. When the **Add New Item** dialog window appears, click the **Windows Form** icon, and then type **CreateContact** in the Name field.
4. Click **Add**.
5. **CreateContact.cs[Design],** a design window for Windows Form, appears in the **IDE Project Editor** window.
6. Click the **CreateContact** dialog window in the top left corner of the screen.
7. In the bottom right corner of the screen, in Properties Window, the information about **ChangeGeneralBody** appears, where you need to set the size to 341, 544.
8. Also, set **FormBorderStype** to **FixedToolWindow**.
9. Move the cursor over the **ToolBox** in the top left corner of the screen. You will see the menu with which you can add a dialog window, button, and other control functions.
10. For **CheckedListBox1, Button1**, and **TextBox1**, modify the values of **Text** and **Name** by referring to the following table.

Dialog Element	Text	Name	Location	Size
CheckedListBox1		lbContactList	12, 12	299, 424
Button1	Create Contact	btCreateContact	12, 451	299, 37

11. After setting all of the values described above, the dialog window should resemble the figure on the right.

12. **Click File – Save CreateContact.cs** to **save** the file.



Configuring the Initial Settings of the Dialog Window

So far we have configured the appearance of the dialog window. Now, you must add variables to define what values users can enter in the text box and the event generated when users click a button. Also, you will learn how to use the **ProcessNet** function in the dialog window.

To configure the initial settings of the CreateContact dialog window:

1. In the **Project Explorer**, right-click **CreateContact.cs**. In the context menu, click **View Code** to display the source code of **CreateContact.cs** in the **Edit IDE Project** window.
2. In the edit window, enter the variables to be used in the dialog window.

```
using FunctionBay.RecurDyn.ProcessNet;
using FunctionBay.RecurDyn.ProcessNet.BNP;

namespace SimpleBeltSystem
{
        public partial class CreateContact: Form
        {
                IModelDocument modelDocument;
                ISubSystem sub;
                public CreateContact(IModelDocument modelDoc)
                {
                        InitializeComponent();
                        modelDocument = modelDoc;
                        sub = modelDocument.GetDataStorage() as ISubSystem;
                }
        }
}
```

- **FunctionBay.RecurDyn.ProcessNet** and **FunctionBay.RecurDyn.ProcessNet.BNP** are the references for using the ProcessNet functions.

3. In the **Project Explorer**, right-click **CreateContact.cs**, and then click **View Designer**. In the dialog window, double-click the **Create Contact** button to create the **btCreateContact_Click() function.**
4. Enter the following code in the **btCreateContact_Click()** function.

```
private void btCreateContact_Click(object sender, EventArgs e)
{
          if (this.lbContactList.CheckedItems.Count != 0)
          {
                    for (int assemblyCount = 0; assemblyCount <= this.lbContactList.Items.Count - 1;
assemblyCount++)
                    {
                              if (this.lbContactList.GetItemCheckState(assemblyCount) == CheckState.Checked)
                              {
                                        string[] assembliesName =
this.lbContactList.Items[assemblyCount].ToString().Split('.');
                                        createContact(assembliesName[0], assembliesName[1]);
                                        this.lbContactList.SetItemChecked(assemblyCount, false);
                              }
                    }
          }
}
```

5. Once again, in the **Project Explorer**, right-click the **CreateContact.cs** file, and click **View Designer**.
6. Double-click an empty area in the dialog window.
7. The **CreateContact_Load()** function is created as shown below. Enter the following code in the function.

```
private void CreateContact_Load(object sender, EventArgs e)
{
          ISubSystem sub = modelDocument.GetDataStorage() as ISubSystem;
          IContactCollection contacts = sub.ContactCollection as IContactCollection;
          foreach (IContact contact in contacts)
          {
                    string contactType = Microsoft.VisualBasic.Information.TypeName(contact);
                    if (contactType == "IContactGeoSurface" || contactType == "IContactGeoCurve")
                    {
                              string groupName = findContact(contact);
                              if (groupName != "")
                              {
                                        this.lbContactList.Items.Add(groupName + "." + contact.Name, true);
                              }
                    }
          }
}
```

- This function retrieves the Contact connected to the group body when the dialog window opens.

8. To find the Contact connected to the group body, create a **findContact()** function.

```
public string findContact(IContact contact)
{
        string groupName = "";
        IGeometry actionGeometry = contact.ActionEntity;
        IGeometry baseGeometry = contact.BaseEntity;

        string[] actiondummy = actionGeometry.FullName.Split('.');
        string[] basedummy = baseGeometry.FullName.Split('.');

        string actionName = actiondummy[0];
        string baseName = basedummy[0];

        ISubSystem sub = modelDocument.GetDataStorage() as ISubSystem;
        IGroupGeneralCollection groupGenerals = sub.GroupGeneralCollection as IGroupGeneralCollection;
        if (groupGenerals.Count > 1)
        {
                foreach (IGroupGeneral groupGeneral in groupGenerals)
                {
                        IGenericCollection generics = groupGeneral.GroupEntities();
                        if (generics.Count > 1)
                        {
                                foreach (IGeneric generic in generics)
                                {
                                        if (generic.FullName.Contains(actionName) ||
                                generic.FullName.Contains(baseName))
                                        {
                                                groupName = groupGeneral.Name;
                                                break;
                                        }
                                }
                                if (groupName !="")
                                        break;
                        }
                }
        }
        return groupName;
}
```

- If the geometry of the group composed of general bodies matches the geometry of the Contact, it is judged that there is a Contact.

Creating a Contact

In case of belt body and connector, the related information could be brought from the assembly using ProcessNet, but in case of Contact, it was not the case. So, once you define one General Contact directly, you can use the defined Contact and ProcessNet to create a Contact between a pulley and an assembly. In this lesson you will learn how to create a Contact using Geo Surface Contact and Geo Curve Contact.

To create a Contact:

1. Create the **createContact()** function. Copy the following code and insert it into the function.
    - This function creates a Contact between a general body and a pulley. For Contact, use **Geo Surface Contact** and **Geo Curve Contact** functions.

```
public void createContact(string generalGroupName, string contactName)
{
        IContact contact = sub.GetEntity(contactName) as IContact;
        IGroupGeneral groupGeneral = sub.GetEntity(generalGroupName) as IGroupGeneral;
        IGenericCollection generics = groupGeneral.GroupEntities();
```

```
    IContactGeoSurface geoSurface = contact as IContactGeoSurface;
    IContactGeoCurve geoCurve = contact as IContactGeoCurve;
    IGeometry actionGeometry = contact.ActionEntity as IGeometry;
    IGeometry baseGeometry = contact.BaseEntity as IGeometry;

    string actionBodyName = actionGeometry.FullName.Split('.')[0];
    string baseBodyName = baseGeometry.FullName.Split('.')[0];
    bool contactActionbody = false;
    foreach (IGeneric linkbody in generics)
    {
        if (linkbody.Name == actionBodyName)
        {
            contactActionbody = true;
            break;
        }
    }
    IGroupGeneral contactgroup = sub.CreateGroupGeneral("Contact_" + generalGroupName + "_" +
contactName, new object[] { });
    if (geoSurface != null)
    {
        foreach (IBody body in generics)
        {
            if (contactActionbody)
            {
                string[] dummy = actionGeometry.FullName.Split('.');
                dummy[0] = body.Name;
                string linkGeometryName = null;
                for (int count = 0; count < dummy.Length; count++)
                {
                    if (count == dummy.Length - 1)
                    {
                        linkGeometryName += dummy[count];
                    }
                    else
                    {
                        linkGeometryName += dummy[count] + ".";
                    }
                }
                actionGeometry =
modelDocument.GetEntityFromFullName(linkGeometryName) as IGeometry;

            }
            else
            {
                string[] dummy = baseGeometry.FullName.Split('.');
                dummy[0] = body.Name;
                string linkGeometryName = null;
                for (int count = 0; count < dummy.Length; count++)
                {
                    if (count == dummy.Length - 1)
                    {
                        linkGeometryName += dummy[count];
                    }
                    else
                    {
                        linkGeometryName += dummy[count] + ".";
                    }
                }
            }
            geoSurface = sub.CreateContactGeoSurface(contactName + "_" + body.Name,
baseGeometry, actionGeometry);
            contactgroup.AddEntities(new object[] { geoSurface });
        }
    }
    else if (geoCurve != null)
    {
        foreach (IBody body in generics)
        {
            if (contactActionbody)
            {
                string[] dummy = actionGeometry.FullName.Split('.');
                dummy[0] = body.Name;
                string linkGeometryName = null;
                for (int count = 0; count < dummy.Length; count++)
                {
                    if (count == dummy.Length - 1)
                    {
                        linkGeometryName += dummy[count];
                    }
                    else
                    {
                        linkGeometryName += dummy[count] + ".";
                    }
                }
                actionGeometry =
modelDocument.GetEntityFromFullName(linkGeometryName) as IGeometry;

            }
            else
            {
                string[] dummy = baseGeometry.FullName.Split('.');
                dummy[0] = body.Name;
                string linkGeometryName = null;
                for (int count = 0; count < dummy.Length; count++)
                {
                    if (count == dummy.Length - 1)
                    {
                        linkGeometryName += dummy[count];
                    }
                    else
                    {
                        linkGeometryName += dummy[count] + ".";
                    }
                }
                baseGeometry =
modelDocument.GetEntityFromFullName(linkGeometryName) as IGeometry;

            }
            geoCurve = sub.CreateContactGeoCurve(contactName + "_" + body.Name,
baseGeometry, actionGeometry);
            contactgroup.AddEntities(new object[] { geoCurve });
        }

    }

    IGenericCollection contacts = contactgroup.GroupEntities();

    foreach (IContact cpoyContact in contacts)
    {
        copyContactProf(cpoyContact, contact);
    }
    modelDocument.DeleteEntity(contact);
    modelDocument.SetUndoHistory("Create Contact");
}
```

A description of the **createContact()** function is provided below.

```
IContact contact = sub.GetEntity(contactName) as IContact;
IGroupGeneral groupGeneral = sub.GetEntity(generalGroupName) as IGroupGeneral;
IGenericCollection generics = groupGeneral.GroupEntities();

IContactGeoSurface geoSurface = contact as IContactGeoSurface;
IContactGeoCurve geoCurve = contact as IContactGeoCurve;
IGeometry actionGeometry = contact.ActionEntity as IGeometry;
IGeometry baseGeometry = contact.BaseEntity as IGeometry;

string actionBodyName = actionGeometry.FullName.Split('.')[0];
string baseBodyName = baseGeometry.FullName.Split('.')[0];
```

The code above declares a variable to use in the **createContact()** function.

- **Contact**: A Contact between a general body and a pulley
- **groupGeneral**: A group general which comprises general bodies converted from clone bodies
- **actionBodyName**: Name of an action body
- **baseBodyName**: Name of a base body

```
bool contactActionbody = false;
foreach (IGeneric linkbody in generics)
{
        if (linkbody.Name == actionBodyName)
        {
                contactActionbody = true;
                break;
        }
}
```

The above code adds a conditional statement to judge whether the action entity of a Contact is a general body or a pulley. If the action entity is a pulley, a Contact is created through loop statement using the base entity value. If it is opposite, a Contact is created through loop statement using the action entity value.

```
IGroupGeneral contactgroup = sub.CreateGroupGeneral("Contact_" + generalGroupName + "_" + contactName,
new object[] { });
```

The code above creates a group general to add the created Contact to.

```
if (geoSurface != null)
{
    foreach (IBody body in generics)
    {
        if (contactActionbody)
        {
            string[] dummy = actionGeometry.FullName.Split('.');
            dummy[0] = body.Name;
            string linkGeometryName = null;
            for (int count = 0; count < dummy.Length; count++)
            {
                if (count == dummy.Length - 1)
                {
                    linkGeometryName += dummy[count];
                }
                else
                {
                    linkGeometryName += dummy[count] + ".";
                }
            }
            actionGeometry = modelDocument.GetEntityFromFullName(linkGeometryName)
        as IGeometry;

        }
        else
        {
            string[] dummy = baseGeometry.FullName.Split('.');
            dummy[0] = body.Name;
            string linkGeometryName = null;
            for (int count = 0; count < dummy.Length; count++)
            {
                if (count == dummy.Length - 1)
                {
                    linkGeometryName += dummy[count];
                }
                else
                {
                    linkGeometryName += dummy[count] + ".";
                }
            }
       }
       geoSurface = sub.CreateContactGeoSurface(contactName + "_" + body.Name,
baseGeometry, actionGeometry);
       contactgroup.AddEntities(new object[] { geoSurface });
    }
}
```

The code above creates a Geo Surface Contact between a pulley and a general body. It creates a Geo Surface Contact between a pulley and a body included in the generics collection. It executes the loop statement after judging whether the link body is an action entity or a base entity.

- If you run the FullName method, it returns in the form of **BodyName.GeometryName.FaceName@Subsystem**. So you can use the Split function to sever the body name at the beginning and change it to the name of another body to create a Contact.

```
else if (geoCurve != null)
{
        foreach (IBody body in generics)
        {
                if (contactActionbody)
                {
                        string[] dummy = actionGeometry.FullName.Split('.');
                        dummy[0] = body.Name;
                        string linkGeometryName = null;
                        for (int count = 0; count < dummy.Length; count++)
                        {
                                if (count == dummy.Length - 1)
                                {
                                        linkGeometryName += dummy[count];
                                }
                                else
                                {
                                        linkGeometryName += dummy[count] + ".";
                                }
                        }
                        actionGeometry = modelDocument.GetEntityFromFullName(linkGeometryName)
                as IGeometry;

                }
                else
                {
                        string[] dummy = baseGeometry.FullName.Split('.');
                        dummy[0] = body.Name;
                        string linkGeometryName = null;
                        for (int count = 0; count < dummy.Length; count++)
                        {
                                if (count == dummy.Length - 1)
                                {
                                        linkGeometryName += dummy[count];
                                }
                                else
                                {
                                        linkGeometryName += dummy[count] + ".";
                                }
                        }
                        baseGeometry = modelDocument.GetEntityFromFullName(linkGeometryName)
                as IGeometry;

                }
                geoCurve = sub.CreateContactGeoCurve(contactName + "_" + body.Name, baseGeometry,
        actionGeometry);
                contactgroup.AddEntities(new object[] { geoCurve });
                }

}
```

The code above creates a Geo Curve Contact between a pulley and a general body. Perform the same procedure as the Geo Surface Contact.

```
IGenericCollection contacts = contactgroup.GroupEntities();

foreach (IContact cpoyContact in contacts)
{
        copyContactProf(cpoyContact, contact);
}
modelDocument.DeleteEntity(contact);
modelDocument.SetUndoHistory("Create Contact");
```

The above code copies the Contact information and adds the Undo history.

2. Create the **copyContactProf()** function that copies the Contact Property value.

```
private void copyContactProf(IContact copyContact, IContact originalContact)
{
            string contactType = Microsoft.VisualBasic.Information.TypeName(originalContact);
            if (contactType == "IContactGeoSurface")
            {
                IContactGeoSurface copyGeoSurface = copyContact as IContactGeoSurface;
                IContactGeoSurface originalGeoSurface = originalContact as IContactGeoSurface;

                copyGeoSurface.ActionPatchOption.SurfaceType =
originalGeoSurface.ActionPatchOption.SurfaceType;
                if (originalGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue == null)
                {
                    copyGeoSurface.ActionPatchOption.BoundingBufferLength.Value =
originalGeoSurface.ActionPatchOption.BoundingBufferLength.Value;

                }
                else
                {
                    copyGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue =
originalGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue;

                }
                copyGeoSurface.ActionPatchOption.UsePlaneToleranceFactor =
originalGeoSurface.ActionPatchOption.UsePlaneToleranceFactor;
                if (originalGeoSurface.ActionPatchOption.UsePlaneToleranceFactor)
                {
                    if (originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue == null)
                    {
                        copyGeoSurface.ActionPatchOption.PlaneToleranceFactor.Value =
originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.Value;

                    }
                    else
                    {
                        copyGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue =
originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue;

                    }
                }

                copyGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor =
originalGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor;
                if (originalGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor)
                {
                    if (originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue == null)
                    {
                        copyGeoSurface.ActionPatchOption.MaxFacetSizeFactor.Value =
originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.Value;

                    }
                    else
                    {
                        copyGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue =
originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue;

                    }
                }
                copyGeoSurface.ActionPatchOption.UseCubicCell =
originalGeoSurface.ActionPatchOption.UseCubicCell;
                if (originalGeoSurface.ActionPatchOption.UseCubicCell)
                {
                    copyGeoSurface.ActionPatchOption.CubicCell = originalGeoSurface.ActionPatchOption.CubicCell;
                }
                copyGeoSurface.BasePatchOption.SurfaceType = originalGeoSurface.BasePatchOption.SurfaceType;
                if (originalGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue == null)
                {
                    copyGeoSurface.ActionPatchOption.BoundingBufferLength.Value =
originalGeoSurface.ActionPatchOption.BoundingBufferLength.Value;

                }
                else
                {
                    copyGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue =
originalGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue;

                }
                copyGeoSurface.ActionPatchOption.UsePlaneToleranceFactor =
originalGeoSurface.ActionPatchOption.UsePlaneToleranceFactor;
                if (originalGeoSurface.ActionPatchOption.UsePlaneToleranceFactor)
                {
                    if (originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue == null)
                    {
                        copyGeoSurface.ActionPatchOption.PlaneToleranceFactor.Value =
originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.Value;

                    }
                    else
                    {
                        copyGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue =
originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue;

                    }
                }

                copyGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor =
originalGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor;
                if (originalGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor)
                {
                    if (originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue == null)
                    {
                        copyGeoSurface.ActionPatchOption.MaxFacetSizeFactor.Value =
originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.Value;

                    }
                    else
                    {
                        copyGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue =
originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue;

                    }
                }
                copyGeoSurface.ActionPatchOption.UseCubicCell =
originalGeoSurface.ActionPatchOption.UseCubicCell;
                if (originalGeoSurface.ActionPatchOption.UseCubicCell)
                {
                    copyGeoSurface.ActionPatchOption.CubicCell = originalGeoSurface.ActionPatchOption.CubicCell;
                }

                copyGeoSurface.BasePatchOption.SurfaceType = originalGeoSurface.BasePatchOption.SurfaceType;
                if (originalGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue == null)
                {
                    copyGeoSurface.BasePatchOption.BoundingBufferLength.Value =
originalGeoSurface.BasePatchOption.BoundingBufferLength.Value;

                }
                else
                {
                    copyGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue =
originalGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue;

                }
```

```
            copyGeoSurface.BasePatchOption.UsePlaneToleranceFactor =
originalGeoSurface.BasePatchOption.UsePlaneToleranceFactor;
            if (originalGeoSurface.BasePatchOption.UsePlaneToleranceFactor)
            {
                if (originalGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue == null)
                {
                    copyGeoSurface.BasePatchOption.PlaneToleranceFactor.Value =
originalGeoSurface.BasePatchOption.PlaneToleranceFactor.Value;

                }
                else
                {
                    copyGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue =
originalGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue;

                }
            }

            copyGeoSurface.BasePatchOption.UseMaxFacetSizeFactor =
originalGeoSurface.BasePatchOption.UseMaxFacetSizeFactor;
            if (originalGeoSurface.BasePatchOption.UseMaxFacetSizeFactor)
            {
                if (originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue == null)
                {
                    copyGeoSurface.BasePatchOption.MaxFacetSizeFactor.Value =
originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.Value;

                }
                else
                {
                    copyGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue =
originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue;

                }
            }
            copyGeoSurface.BasePatchOption.UseCubicCell = originalGeoSurface.BasePatchOption.UseCubicCell;
            if (originalGeoSurface.BasePatchOption.UseCubicCell)
            {
                copyGeoSurface.BasePatchOption.CubicCell = originalGeoSurface.BasePatchOption.CubicCell;
            }

            copyGeoSurface.BasePatchOption.SurfaceType = originalGeoSurface.BasePatchOption.SurfaceType;
            if (originalGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue == null)
            {
                copyGeoSurface.BasePatchOption.BoundingBufferLength.Value =
originalGeoSurface.BasePatchOption.BoundingBufferLength.Value;

            }
            else
            {
                copyGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue =
originalGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue;

            }
            copyGeoSurface.BasePatchOption.UsePlaneToleranceFactor =
originalGeoSurface.BasePatchOption.UsePlaneToleranceFactor;
            if (originalGeoSurface.BasePatchOption.UsePlaneToleranceFactor)
            {
                if (originalGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue == null)
                {
                    copyGeoSurface.BasePatchOption.PlaneToleranceFactor.Value =
originalGeoSurface.BasePatchOption.PlaneToleranceFactor.Value;

                }
                else
                {
                    copyGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue =
originalGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue;
```

```
                    }
                }

                copyGeoSurface.BasePatchOption.UseMaxFacetSizeFactor =
originalGeoSurface.BasePatchOption.UseMaxFacetSizeFactor;
                if (originalGeoSurface.BasePatchOption.UseMaxFacetSizeFactor)
                {
                    if (originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue == null)
                    {
                        copyGeoSurface.BasePatchOption.MaxFacetSizeFactor.Value =
originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.Value;

                    }
                    else
                    {
                        copyGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue =
originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue;

                    }
                }
                copyGeoSurface.BasePatchOption.UseCubicCell = originalGeoSurface.BasePatchOption.UseCubicCell;
                if (originalGeoSurface.BasePatchOption.UseCubicCell)
                {
                    copyGeoSurface.BasePatchOption.CubicCell = originalGeoSurface.BasePatchOption.CubicCell;
                }
                copyGeoSurface.ActionUpDirection = originalGeoSurface.ActionUpDirection;
                copyGeoSurface.ActionNodeContact = originalGeoSurface.ActionNodeContact;

                copyGeoSurface.BaseUpDirection = originalGeoSurface.BaseUpDirection;
                copyGeoSurface.BaseNodeContact = originalGeoSurface.BaseNodeContact;

                copyGeoSurface.EdgeContact = originalGeoSurface.EdgeContact;

                copyGeoSurface.UseCPM = originalGeoSurface.UseCPM;
                copyGeoSurface.SmoothNodeContact = originalGeoSurface.SmoothNodeContact;

                if (originalGeoSurface.ContactProperty.UseStiffnessSpline)
                {
                    copyGeoSurface.ContactProperty.StiffnessSpline =
originalGeoSurface.ContactProperty.StiffnessSpline;
                }
                else
                {
                    if (originalGeoSurface.ContactProperty.StiffnessCoefficient.ParametricValue == null)
                    {
                        copyGeoSurface.ContactProperty.StiffnessCoefficient.Value =
originalGeoSurface.ContactProperty.StiffnessCoefficient.Value;
                    }
                    else
                    {
                        copyGeoSurface.ContactProperty.StiffnessCoefficient.ParametricValue =
originalGeoSurface.ContactProperty.StiffnessCoefficient.ParametricValue;
                    }
                }

                if (originalGeoSurface.ContactProperty.UseDampingSpline)
                {
                    copyGeoSurface.ContactProperty.DampingSpline =
originalGeoSurface.ContactProperty.DampingSpline;
                }
                else
                {
                    if (originalGeoSurface.ContactProperty.DampingCoefficient.ParametricValue == null)
                    {
                        copyGeoSurface.ContactProperty.DampingCoefficient.Value =
originalGeoSurface.ContactProperty.DampingCoefficient.Value;
                    }
                    else
                    {
                        copyGeoSurface.ContactProperty.DampingCoefficient.ParametricValue =
originalGeoSurface.ContactProperty.DampingCoefficient.ParametricValue;
                    }
                }

                copyGeoSurface.ContactProperty.Friction.ContactFrictionType =
originalGeoSurface.ContactProperty.Friction.ContactFrictionType;
                if (copyGeoSurface.ContactProperty.Friction.ContactFrictionType ==
ContactFrictionType.CoefficientSpline)
                {
                    copyGeoSurface.ContactProperty.Friction.Spline =
originalGeoSurface.ContactProperty.Friction.Spline;
                }
                else if (copyGeoSurface.ContactProperty.Friction.ContactFrictionType ==
ContactFrictionType.ForceSpline)
                {
                    copyGeoSurface.ContactProperty.Friction.Spline =
originalGeoSurface.ContactProperty.Friction.Spline;
                }
                else
                {
                    if (originalGeoSurface.ContactProperty.Friction.Coefficient.ParametricValue == null)
                    {
                        copyGeoSurface.ContactProperty.Friction.Coefficient.Value =
originalGeoSurface.ContactProperty.Friction.Coefficient.Value;
                    }
                    else
                    {
                        copyGeoSurface.ContactProperty.Friction.Coefficient.ParametricValue =
originalGeoSurface.ContactProperty.Friction.Coefficient.ParametricValue;
                    }

                    if (originalGeoSurface.ContactProperty.Friction.StaticThresholdVelocity.ParametricValue == null)
                    {
                        copyGeoSurface.ContactProperty.Friction.StaticThresholdVelocity.Value =
originalGeoSurface.ContactProperty.Friction.StaticThresholdVelocity.Value;
                    }
                    else
                    {
                        copyGeoSurface.ContactProperty.Friction.StaticThresholdVelocity.ParametricValue =
originalGeoSurface.ContactProperty.Friction.StaticThresholdVelocity.ParametricValue;

                    }

                    if (originalGeoSurface.ContactProperty.Friction.DynamicThresholdVelocity.ParametricValue ==
null)
                    {
                        copyGeoSurface.ContactProperty.Friction.DynamicThresholdVelocity.Value =
originalGeoSurface.ContactProperty.Friction.DynamicThresholdVelocity.Value;
                    }
                    else
                    {
                        copyGeoSurface.ContactProperty.Friction.DynamicThresholdVelocity.ParametricValue =
originalGeoSurface.ContactProperty.Friction.DynamicThresholdVelocity.ParametricValue;

                    }
                    if (originalGeoSurface.ContactProperty.Friction.StaticCoefficient.ParametricValue == null)
                    {
                        copyGeoSurface.ContactProperty.Friction.StaticCoefficient.Value =
originalGeoSurface.ContactProperty.Friction.StaticCoefficient.Value;
                    }
                    else
                    {
                        copyGeoSurface.ContactProperty.Friction.StaticCoefficient.ParametricValue =
originalGeoSurface.ContactProperty.Friction.StaticCoefficient.ParametricValue;
                    }
                    if (originalGeoSurface.ContactProperty.Friction.MaximumForce.ParametricValue == null)
                    {
                        copyGeoSurface.ContactProperty.Friction.MaximumForce.Value = 
originalGeoSurface.ContactProperty.Friction.MaximumForce.Value;
                    }
                    else
                    {
                        copyGeoSurface.ContactProperty.Friction.MaximumForce.ParametricValue = 
originalGeoSurface.ContactProperty.Friction.MaximumForce.ParametricValue;
                    }
                    copyGeoSurface.ContactProperty.Friction.UseMaximumForce = 
private void copyContactProf(IContact copyContact, IContact originalContact)
{
        string contactType = Microsoft.VisualBasic.Information.TypeName(originalContact);
        if (contactType == "IContactGeoSurface")
        {
                    IContactGeoSurface copyGeoSurface = copyContact as IContactGeoSurface;
                    IContactGeoSurface originalGeoSurface = originalContact as IContactGeoSurface;

                    copyGeoSurface.ActionPatchOption.SurfaceType = 
originalGeoSurface.ActionPatchOption.SurfaceType;
                    if (originalGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue == null)
                    {
                            copyGeoSurface.ActionPatchOption.BoundingBufferLength.Value = 
originalGeoSurface.ActionPatchOption.BoundingBufferLength.Value;

                    }
                    else
                    {
                            copyGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue = 
originalGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue;

                    }
                    copyGeoSurface.ActionPatchOption.UsePlaneToleranceFactor = 
originalGeoSurface.ActionPatchOption.UsePlaneToleranceFactor;
                    if (originalGeoSurface.ActionPatchOption.UsePlaneToleranceFactor)
                    {
                            if (originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue == 
null)
                            {
                                    copyGeoSurface.ActionPatchOption.PlaneToleranceFactor.Value = 
originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.Value;

                            }
                            else
                            {

        copyGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue = 
originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue;

                            }
                    }

                    copyGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor = 
originalGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor;
                    if (originalGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor)
                    {
                            if (originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue == 
null)
                            {
                                    copyGeoSurface.ActionPatchOption.MaxFacetSizeFactor.Value = 
originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.Value;

                            }
                            else
                            {
                                    copyGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue 
= originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue;
```

```
            }
        }
        copyGeoSurface.ActionPatchOption.UseCubicCell =
originalGeoSurface.ActionPatchOption.UseCubicCell;
        if (originalGeoSurface.ActionPatchOption.UseCubicCell)
        {
            copyGeoSurface.ActionPatchOption.CubicCell =
originalGeoSurface.ActionPatchOption.CubicCell;
        }

        copyGeoSurface.BasePatchOption.SurfaceType =
originalGeoSurface.BasePatchOption.SurfaceType;
        if (originalGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue == null)
        {
            copyGeoSurface.ActionPatchOption.BoundingBufferLength.Value =
originalGeoSurface.ActionPatchOption.BoundingBufferLength.Value;

        }
        else
        {
            copyGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue =
originalGeoSurface.ActionPatchOption.BoundingBufferLength.ParametricValue;

        }
        copyGeoSurface.ActionPatchOption.UsePlaneToleranceFactor =
originalGeoSurface.ActionPatchOption.UsePlaneToleranceFactor;
        if (originalGeoSurface.ActionPatchOption.UsePlaneToleranceFactor)
        {
            if (originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue ==
null)
            {
                copyGeoSurface.ActionPatchOption.PlaneToleranceFactor.Value =
originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.Value;

            }
            else
            {

    copyGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue =
originalGeoSurface.ActionPatchOption.PlaneToleranceFactor.ParametricValue;

            }
        }

        copyGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor =
originalGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor;
        if (originalGeoSurface.ActionPatchOption.UseMaxFacetSizeFactor)
        {
            if (originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue ==
null)
            {
                copyGeoSurface.ActionPatchOption.MaxFacetSizeFactor.Value =
originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.Value;

            }
            else
            {
                copyGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue
= originalGeoSurface.ActionPatchOption.MaxFacetSizeFactor.ParametricValue;

            }
        }
        copyGeoSurface.ActionPatchOption.UseCubicCell =
originalGeoSurface.ActionPatchOption.UseCubicCell;
        if (originalGeoSurface.ActionPatchOption.UseCubicCell)
        {
            copyGeoSurface.ActionPatchOption.CubicCell = 
originalGeoSurface.ActionPatchOption.CubicCell;
        }

        copyGeoSurface.BasePatchOption.SurfaceType = 
originalGeoSurface.BasePatchOption.SurfaceType;
        if (originalGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue == null)
        {
            copyGeoSurface.BasePatchOption.BoundingBufferLength.Value = 
originalGeoSurface.BasePatchOption.BoundingBufferLength.Value;

        }
        else
        {
            copyGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue = 
originalGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue;

        }
        copyGeoSurface.BasePatchOption.UsePlaneToleranceFactor = 
originalGeoSurface.BasePatchOption.UsePlaneToleranceFactor;
        if (originalGeoSurface.BasePatchOption.UsePlaneToleranceFactor)
        {
            if (originalGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue == 
null)
            {
                copyGeoSurface.BasePatchOption.PlaneToleranceFactor.Value = 
originalGeoSurface.BasePatchOption.PlaneToleranceFactor.Value;

            }
            else
            {
                copyGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue 
= originalGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue;

            }
        }

        copyGeoSurface.BasePatchOption.UseMaxFacetSizeFactor = 
originalGeoSurface.BasePatchOption.UseMaxFacetSizeFactor;
        if (originalGeoSurface.BasePatchOption.UseMaxFacetSizeFactor)
        {
            if (originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue == 
null)
            {
                copyGeoSurface.BasePatchOption.MaxFacetSizeFactor.Value = 
originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.Value;

            }
            else
            {
                copyGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue = 
originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue;

            }
        }
        copyGeoSurface.BasePatchOption.UseCubicCell = 
originalGeoSurface.BasePatchOption.UseCubicCell;
        if (originalGeoSurface.BasePatchOption.UseCubicCell)
        {
            copyGeoSurface.BasePatchOption.CubicCell = 
originalGeoSurface.BasePatchOption.CubicCell;
        }

        copyGeoSurface.BasePatchOption.SurfaceType = 
originalGeoSurface.BasePatchOption.SurfaceType;
        if (originalGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue == null)
        {
```

```
                        copyGeoSurface.BasePatchOption.BoundingBufferLength.Value =
originalGeoSurface.BasePatchOption.BoundingBufferLength.Value;

                }
                else
                {
                        copyGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue =
originalGeoSurface.BasePatchOption.BoundingBufferLength.ParametricValue;

                }
                copyGeoSurface.BasePatchOption.UsePlaneToleranceFactor =
originalGeoSurface.BasePatchOption.UsePlaneToleranceFactor;
                if (originalGeoSurface.BasePatchOption.UsePlaneToleranceFactor)
                {
                        if (originalGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue ==
null)
                        {
                                copyGeoSurface.BasePatchOption.PlaneToleranceFactor.Value =
originalGeoSurface.BasePatchOption.PlaneToleranceFactor.Value;

                        }
                        else
                        {
                                copyGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue
= originalGeoSurface.BasePatchOption.PlaneToleranceFactor.ParametricValue;

                        }
                }

                copyGeoSurface.BasePatchOption.UseMaxFacetSizeFactor =
originalGeoSurface.BasePatchOption.UseMaxFacetSizeFactor;
                if (originalGeoSurface.BasePatchOption.UseMaxFacetSizeFactor)
                {
                        if (originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue ==
null)
                        {
                                copyGeoSurface.BasePatchOption.MaxFacetSizeFactor.Value =
originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.Value;

                        }
                        else
                        {
                                copyGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue =
originalGeoSurface.BasePatchOption.MaxFacetSizeFactor.ParametricValue;

                        }
                }
                copyGeoSurface.BasePatchOption.UseCubicCell =
originalGeoSurface.BasePatchOption.UseCubicCell;
                if (originalGeoSurface.BasePatchOption.UseCubicCell)
                {
                        copyGeoSurface.BasePatchOption.CubicCell =
originalGeoSurface.BasePatchOption.CubicCell;
                }
                copyGeoSurface.ActionUpDirection = originalGeoSurface.ActionUpDirection;
                copyGeoSurface.ActionNodeContact = originalGeoSurface.ActionNodeContact;

                copyGeoSurface.BaseUpDirection = originalGeoSurface.BaseUpDirection;
                copyGeoSurface.BaseNodeContact = originalGeoSurface.BaseNodeContact;

                copyGeoSurface.EdgeContact = originalGeoSurface.EdgeContact;

                copyGeoSurface.UseCPM = originalGeoSurface.UseCPM;
                copyGeoSurface.SmoothNodeContact = originalGeoSurface.SmoothNodeContact;

                if (originalGeoSurface.ContactProperty.UseStiffnessSpline)
                {
```

```
            copyGeoSurface.ContactProperty.StiffnessSpline =
originalGeoSurface.ContactProperty.StiffnessSpline;
        }
        else
        {
            if (originalGeoSurface.ContactProperty.StiffnessCoefficient.ParametricValue == null)
            {
                copyGeoSurface.ContactProperty.StiffnessCoefficient.Value =
originalGeoSurface.ContactProperty.StiffnessCoefficient.Value;
            }
            else
            {
                copyGeoSurface.ContactProperty.StiffnessCoefficient.ParametricValue =
originalGeoSurface.ContactProperty.StiffnessCoefficient.ParametricValue;
            }
        }

        if (originalGeoSurface.ContactProperty.UseDampingSpline)
        {
            copyGeoSurface.ContactProperty.DampingSpline =
originalGeoSurface.ContactProperty.DampingSpline;
        }
        else
        {
            if (originalGeoSurface.ContactProperty.DampingCoefficient.ParametricValue ==
null)
            {
                copyGeoSurface.ContactProperty.DampingCoefficient.Value =
originalGeoSurface.ContactProperty.DampingCoefficient.Value;
            }
            else
            {
                copyGeoSurface.ContactProperty.DampingCoefficient.ParametricValue =
originalGeoSurface.ContactProperty.DampingCoefficient.ParametricValue;
            }
        }

        copyGeoSurface.ContactProperty.Friction.ContactFrictionType =
originalGeoSurface.ContactProperty.Friction.ContactFrictionType;
        if (copyGeoSurface.ContactProperty.Friction.ContactFrictionType ==
ContactFrictionType.CoefficientSpline)
        {
            copyGeoSurface.ContactProperty.Friction.Spline =
originalGeoSurface.ContactProperty.Friction.Spline;
        }
        else if (copyGeoSurface.ContactProperty.Friction.ContactFrictionType ==
ContactFrictionType.ForceSpline)
        {
            copyGeoSurface.ContactProperty.Friction.Spline =
originalGeoSurface.ContactProperty.Friction.Spline;
        }
        else
        {
            if (originalGeoSurface.ContactProperty.Friction.Coefficient.ParametricValue == null)
            {
                copyGeoSurface.ContactProperty.Friction.Coefficient.Value =
originalGeoSurface.ContactProperty.Friction.Coefficient.Value;
            }
            else
            {
                copyGeoSurface.ContactProperty.Friction.Coefficient.ParametricValue =
originalGeoSurface.ContactProperty.Friction.Coefficient.ParametricValue;
            }

            if
(originalGeoSurface.ContactProperty.Friction.StaticThresholdVelocity.ParametricValue == null)
            {
                copyGeoSurface.ContactProperty.Friction.StaticThresholdVelocity.Value =
originalGeoSurface.ContactProperty.Friction.StaticThresholdVelocity.Value;
            }
            else
            {

    copyGeoSurface.ContactProperty.Friction.StaticThresholdVelocity.ParametricValue =
originalGeoSurface.ContactProperty.Friction.StaticThresholdVelocity.ParametricValue;

            }

            if
(originalGeoSurface.ContactProperty.Friction.DynamicThresholdVelocity.ParametricValue == null)
            {

    copyGeoSurface.ContactProperty.Friction.DynamicThresholdVelocity.Value =
originalGeoSurface.ContactProperty.Friction.DynamicThresholdVelocity.Value;
            }
            else
            {

    copyGeoSurface.ContactProperty.Friction.DynamicThresholdVelocity.ParametricValue =
originalGeoSurface.ContactProperty.Friction.DynamicThresholdVelocity.ParametricValue;

            }
            if (originalGeoSurface.ContactProperty.Friction.StaticCoefficient.ParametricValue ==
null)
            {
                copyGeoSurface.ContactProperty.Friction.StaticCoefficient.Value =
originalGeoSurface.ContactProperty.Friction.StaticCoefficient.Value;
            }
            else
            {

    copyGeoSurface.ContactProperty.Friction.StaticCoefficient.ParametricValue =
originalGeoSurface.ContactProperty.Friction.StaticCoefficient.ParametricValue;
            }
            if (originalGeoSurface.ContactProperty.Friction.MaximumForce.ParametricValue ==
null)
            {
                copyGeoSurface.ContactProperty.Friction.MaximumForce.Value =
originalGeoSurface.ContactProperty.Friction.MaximumForce.Value;
            }
            else
            {

    copyGeoSurface.ContactProperty.Friction.MaximumForce.ParametricValue =
originalGeoSurface.ContactProperty.Friction.MaximumForce.ParametricValue;
            }
            copyGeoSurface.ContactProperty.Friction.UseMaximumForce =
originalGeoSurface.ContactProperty.Friction.UseMaximumForce;


        }

        if (originalGeoSurface.ContactProperty.StiffnessExponent.ParametricValue == null)
        {
            copyGeoSurface.ContactProperty.StiffnessExponent.Value =
originalGeoSurface.ContactProperty.StiffnessExponent.Value;
        }
        else
        {
            copyGeoSurface.ContactProperty.StiffnessExponent.ParametricValue =
originalGeoSurface.ContactProperty.StiffnessExponent.ParametricValue;
        }
```

```
                    if (originalGeoSurface.ContactPropertyAdditional.ReboundDampingFactor.ParametricValue ==
null)
                    {
                              copyGeoSurface.ContactPropertyAdditional.ReboundDampingFactor.Value =
originalGeoSurface.ContactPropertyAdditional.ReboundDampingFactor.Value;
                    }
                    else
                    {

          copyGeoSurface.ContactPropertyAdditional.ReboundDampingFactor.ParametricValue =
originalGeoSurface.ContactPropertyAdditional.ReboundDampingFactor.ParametricValue;
                    }

                    if (originalGeoSurface.ContactPropertyAdditional.GlobalMaxPenetration.ParametricValue ==
null)
                    {
                              copyGeoSurface.ContactPropertyAdditional.GlobalMaxPenetration.Value =
originalGeoSurface.ContactPropertyAdditional.GlobalMaxPenetration.Value;
                    }
                    else
                    {
                              copyGeoSurface.ContactPropertyAdditional.GlobalMaxPenetration.ParametricValue
= originalGeoSurface.ContactPropertyAdditional.GlobalMaxPenetration.ParametricValue;
                    }


                    if (copyGeoSurface.ContactPropertyAdditional.ContactForceType ==
ContactForceType.BoundaryPenetration)
                    {
                              if
(originalGeoSurface.ContactPropertyAdditional.BoundaryPenetration.ParametricValue == null)
                              {
                                        copyGeoSurface.ContactPropertyAdditional.BoundaryPenetration.Value =
originalGeoSurface.ContactPropertyAdditional.BoundaryPenetration.Value;
                              }
                              else
                              {

          copyGeoSurface.ContactPropertyAdditional.BoundaryPenetration.ParametricValue =
originalGeoSurface.ContactPropertyAdditional.BoundaryPenetration.ParametricValue;
                              }
                    }
                    else
                    {
                              if (originalGeoSurface.ContactProperty.UseDampingExponent)
                              {
                                        if
(originalGeoSurface.ContactProperty.DampingExponent.ParametricValue == null)
                                        {
                                                  copyGeoSurface.ContactProperty.DampingExponent.Value =
originalGeoSurface.ContactProperty.DampingExponent.Value;
                                        }
                                        else
                                        {

          copyGeoSurface.ContactProperty.DampingExponent.ParametricValue =
originalGeoSurface.ContactProperty.DampingExponent.ParametricValue;
                                        }
                              }
                              if (originalGeoSurface.ContactProperty.UseIndentationExponent)
                              {
                                        if
(originalGeoSurface.ContactProperty.IndentationExponent.ParametricValue == null)
                                        {
                                                  copyGeoSurface.ContactProperty.IndentationExponent.Value =
originalGeoSurface.ContactProperty.IndentationExponent.Value;
                                        }
                                        else
                                        {

          copyGeoSurface.ContactProperty.IndentationExponent.ParametricValue =
originalGeoSurface.ContactProperty.IndentationExponent.ParametricValue;
                                        }
                              }
                    }

                    if (originalGeoSurface.MaxStepSizeFactor.ParametricValue == null)
                    {
                              copyGeoSurface.MaxStepSizeFactor.Value =
originalGeoSurface.MaxStepSizeFactor.Value;
                    }
                    else
                    {
                              copyGeoSurface.MaxStepSizeFactor.ParametricValue =
originalGeoSurface.MaxStepSizeFactor.ParametricValue;
                    }
          }

          else if (contactType == "IContactGeoCurve")
          {

                    IContactGeoCurve copyGeoCurve = copyContact as IContactGeoCurve;
                    IContactGeoCurve originalGeoCurve = originalContact as IContactGeoCurve;

                    copyGeoCurve.ActionCurveSegmentOption.Segment =
originalGeoCurve.ActionCurveSegmentOption.Segment;
                    copyGeoCurve.ActionCurveSegmentOption.UseTotalSegment =
originalGeoCurve.ActionCurveSegmentOption.UseTotalSegment;

                    if (originalGeoCurve.ActionCurveSegmentOption.BoundingBufferLength.ParametricValue ==
null)
                    {
                              copyGeoCurve.ActionCurveSegmentOption.BoundingBufferLength.Value =
originalGeoCurve.ActionCurveSegmentOption.BoundingBufferLength.Value;

                    }
                    else
                    {
                              copyGeoCurve.ActionCurveSegmentOption.BoundingBufferLength.ParametricValue
= originalGeoCurve.ActionCurveSegmentOption.BoundingBufferLength.ParametricValue;

                    }

                    copyGeoCurve.ActionCurveSegmentOption.UseCubicCell =
originalGeoCurve.ActionCurveSegmentOption.UseCubicCell;
                    if (originalGeoCurve.ActionCurveSegmentOption.UseCubicCell)
                    {
                              copyGeoCurve.ActionCurveSegmentOption.CubicCell =
originalGeoCurve.ActionCurveSegmentOption.CubicCell;
                    }

                    copyGeoCurve.BaseCurveSegmentOption.Segment =
originalGeoCurve.BaseCurveSegmentOption.Segment;
                    copyGeoCurve.BaseCurveSegmentOption.UseTotalSegment =
originalGeoCurve.BaseCurveSegmentOption.UseTotalSegment;

                    if (originalGeoCurve.BaseCurveSegmentOption.BoundingBufferLength.ParametricValue ==
null)
                    {
                              copyGeoCurve.BaseCurveSegmentOption.BoundingBufferLength.Value =
originalGeoCurve.BaseCurveSegmentOption.BoundingBufferLength.Value;

                    }
                    else
                    {
```

```
            copyGeoCurve.BaseCurveSegmentOption.BoundingBufferLength.ParametricValue = 
originalGeoCurve.BaseCurveSegmentOption.BoundingBufferLength.ParametricValue;

        }

        copyGeoCurve.BaseCurveSegmentOption.UseCubicCell = 
originalGeoCurve.BaseCurveSegmentOption.UseCubicCell;
        if (originalGeoCurve.BaseCurveSegmentOption.UseCubicCell)
        {
            copyGeoCurve.BaseCurveSegmentOption.CubicCell = 
originalGeoCurve.BaseCurveSegmentOption.CubicCell;
        }

        //contact plane normal ÀÌ ¾ø´Ù.

        copyGeoCurve.ActionUpDirection = originalGeoCurve.ActionUpDirection;
        copyGeoCurve.ActionNodeContact = originalGeoCurve.ActionNodeContact;

        copyGeoCurve.BaseUpDirection = originalGeoCurve.BaseUpDirection;
        copyGeoCurve.BaseNodeContact = originalGeoCurve.BaseNodeContact;

        copyGeoCurve.UseCPM = originalGeoCurve.UseCPM;
        copyGeoCurve.SmoothNodeContact = originalGeoCurve.SmoothNodeContact;

        if (originalGeoCurve.ContactProperty.UseStiffnessSpline)
        {
            copyGeoCurve.ContactProperty.StiffnessSpline = 
originalGeoCurve.ContactProperty.StiffnessSpline;
        }
        else
        {
            if (originalGeoCurve.ContactProperty.StiffnessCoefficient.ParametricValue == null)
            {
                copyGeoCurve.ContactProperty.StiffnessCoefficient.ParametricValue = 
originalGeoCurve.ContactProperty.StiffnessCoefficient.ParametricValue;
            }
            else
            {
                copyGeoCurve.ContactProperty.StiffnessCoefficient.Value = 
originalGeoCurve.ContactProperty.StiffnessCoefficient.Value;
            }
        }

        if (originalGeoCurve.ContactProperty.UseDampingSpline)
        {
            copyGeoCurve.ContactProperty.DampingSpline = 
originalGeoCurve.ContactProperty.DampingSpline;
        }
        else
        {
            if (originalGeoCurve.ContactProperty.DampingCoefficient.ParametricValue == null)
            {
                copyGeoCurve.ContactProperty.DampingCoefficient.ParametricValue = 
originalGeoCurve.ContactProperty.DampingCoefficient.ParametricValue;
            }
            else
            {
                copyGeoCurve.ContactProperty.DampingCoefficient.Value = 
originalGeoCurve.ContactProperty.DampingCoefficient.Value;
            }
        }

        copyGeoCurve.ContactProperty.Friction.ContactFrictionType = 
originalGeoCurve.ContactProperty.Friction.ContactFrictionType;
        if (copyGeoCurve.ContactProperty.Friction.ContactFrictionType == 
ContactFrictionType.CoefficientSpline)
        {
            copyGeoCurve.ContactProperty.Friction.Spline =
originalGeoCurve.ContactProperty.Friction.Spline;
        }
        else if (copyGeoCurve.ContactProperty.Friction.ContactFrictionType ==
ContactFrictionType.ForceSpline)
        {
            copyGeoCurve.ContactProperty.Friction.Spline =
originalGeoCurve.ContactProperty.Friction.Spline;
        }
        else
        {
            if (originalGeoCurve.ContactProperty.Friction.Coefficient.ParametricValue == null)
            {
                copyGeoCurve.ContactProperty.Friction.Coefficient.Value =
originalGeoCurve.ContactProperty.Friction.Coefficient.Value;
            }
            else
            {
                copyGeoCurve.ContactProperty.Friction.Coefficient.ParametricValue =
originalGeoCurve.ContactProperty.Friction.Coefficient.ParametricValue;
            }

            if
(originalGeoCurve.ContactProperty.Friction.StaticThresholdVelocity.ParametricValue == null)
            {
                copyGeoCurve.ContactProperty.Friction.StaticThresholdVelocity.Value =
originalGeoCurve.ContactProperty.Friction.StaticThresholdVelocity.Value;
            }
            else
            {

    copyGeoCurve.ContactProperty.Friction.StaticThresholdVelocity.ParametricValue =
originalGeoCurve.ContactProperty.Friction.StaticThresholdVelocity.ParametricValue;

            }

            if
(originalGeoCurve.ContactProperty.Friction.DynamicThresholdVelocity.ParametricValue == null)
            {
                copyGeoCurve.ContactProperty.Friction.DynamicThresholdVelocity.Value
= originalGeoCurve.ContactProperty.Friction.DynamicThresholdVelocity.Value;
            }
            else
            {

    copyGeoCurve.ContactProperty.Friction.DynamicThresholdVelocity.ParametricValue =
originalGeoCurve.ContactProperty.Friction.DynamicThresholdVelocity.ParametricValue;

            }
            if (originalGeoCurve.ContactProperty.Friction.StaticCoefficient.ParametricValue ==
null)
            {
                copyGeoCurve.ContactProperty.Friction.StaticCoefficient.Value =
originalGeoCurve.ContactProperty.Friction.StaticCoefficient.Value;
            }
            else
            {
                copyGeoCurve.ContactProperty.Friction.StaticCoefficient.ParametricValue
= originalGeoCurve.ContactProperty.Friction.StaticCoefficient.ParametricValue;
            }
            if (originalGeoCurve.ContactProperty.Friction.MaximumForce.ParametricValue ==
null)
            {
                copyGeoCurve.ContactProperty.Friction.MaximumForce.Value =
originalGeoCurve.ContactProperty.Friction.MaximumForce.Value;
            }
            else
            {
```

```
                                        copyGeoCurve.ContactProperty.Friction.MaximumForce.ParametricValue
= originalGeoCurve.ContactProperty.Friction.MaximumForce.ParametricValue;
                                }
                                copyGeoCurve.ContactProperty.Friction.UseMaximumForce =
originalGeoCurve.ContactProperty.Friction.UseMaximumForce;


                        }

                        if (originalGeoCurve.ContactProperty.StiffnessExponent.ParametricValue == null)
                        {
                                copyGeoCurve.ContactProperty.StiffnessExponent.Value =
originalGeoCurve.ContactProperty.StiffnessExponent.Value;
                        }
                        else
                        {
                                copyGeoCurve.ContactProperty.StiffnessExponent.ParametricValue =
originalGeoCurve.ContactProperty.StiffnessExponent.ParametricValue;
                        }


                        if (originalGeoCurve.ContactPropertyAdditional.ReboundDampingFactor.ParametricValue ==
null)
                        {
                                copyGeoCurve.ContactPropertyAdditional.ReboundDampingFactor.Value =
originalGeoCurve.ContactPropertyAdditional.ReboundDampingFactor.Value;
                        }
                        else
                        {
                                copyGeoCurve.ContactPropertyAdditional.ReboundDampingFactor.ParametricValue
= originalGeoCurve.ContactPropertyAdditional.ReboundDampingFactor.ParametricValue;
                        }

                        if (originalGeoCurve.ContactPropertyAdditional.GlobalMaxPenetration.ParametricValue ==
null)
                        {
                                copyGeoCurve.ContactPropertyAdditional.GlobalMaxPenetration.Value =
originalGeoCurve.ContactPropertyAdditional.GlobalMaxPenetration.Value;
                        }
                        else
                        {
                                copyGeoCurve.ContactPropertyAdditional.GlobalMaxPenetration.ParametricValue =
originalGeoCurve.ContactPropertyAdditional.GlobalMaxPenetration.ParametricValue;
                        }


                        if (copyGeoCurve.ContactPropertyAdditional.ContactForceType ==
ContactForceType.BoundaryPenetration)
                        {
                                if
(originalGeoCurve.ContactPropertyAdditional.BoundaryPenetration.ParametricValue == null)
                                {
                                        copyGeoCurve.ContactPropertyAdditional.BoundaryPenetration.Value =
originalGeoCurve.ContactPropertyAdditional.BoundaryPenetration.Value;
                                }
                                else
                                {

                copyGeoCurve.ContactPropertyAdditional.BoundaryPenetration.ParametricValue =
originalGeoCurve.ContactPropertyAdditional.BoundaryPenetration.ParametricValue;
                                }
                        }
                        else
                        {
                                if (originalGeoCurve.ContactProperty.UseDampingExponent)
                                {
                                        if (originalGeoCurve.ContactProperty.DampingExponent.ParametricValue
== null)
```

```
                                        {
                                                copyGeoCurve.ContactProperty.DampingExponent.Value = 
originalGeoCurve.ContactProperty.DampingExponent.Value;
                                        }
                                        else
                                        {

        copyGeoCurve.ContactProperty.DampingExponent.ParametricValue = 
originalGeoCurve.ContactProperty.DampingExponent.ParametricValue;
                                        }
                                }
                                if (originalGeoCurve.ContactProperty.UseIndentationExponent)
                                {
                                        if 
(originalGeoCurve.ContactProperty.IndentationExponent.ParametricValue == null)
                                        {
                                                copyGeoCurve.ContactProperty.IndentationExponent.Value = 
originalGeoCurve.ContactProperty.IndentationExponent.Value;
                                        }
                                        else
                                        {

        copyGeoCurve.ContactProperty.IndentationExponent.ParametricValue = 
originalGeoCurve.ContactProperty.IndentationExponent.ParametricValue;
                                        }
                                }
                        }

                        if (originalGeoCurve.MaxStepSizeFactor.ParametricValue == null)
                        {
                                copyGeoCurve.MaxStepSizeFactor.Value = 
originalGeoCurve.MaxStepSizeFactor.Value;
                        }
                        else
                        {
                                copyGeoCurve.MaxStepSizeFactor.ParametricValue = 
originalGeoCurve.MaxStepSizeFactor.ParametricValue;
                        }

                }
}
```

3.

Chapter

5

Register DLL

Task Objectives

In this course, learn how to add ConvertCloneBody and CreateContact dialog windows to the RecurDyn ribbon menu using the Register DLL.

Estimated Time to Complete

5 minutes

To display a dialog window when a user runs the application:

You will learn how to display a dialog window when running a ProcessNet application in RecurDyn and how to make that dialog window dependent on RecurDyn.

To display a dialog window when a user runs the application:

1. In the **Project Explorer**, double-click **ThisApplication.cs**.
2. In the **ThisApplication.cs** file, delete the **HelloProcessNet()** and **CreateBodyExample()** functions. (These functions are generated automatically as an example.)

```
~~public void HelloProcessNet()~~
~~{~~
            ~~//application is assigned at Initialize() such as~~
            ~~//application = RecurDynApplication as IApplication;~~
            ~~application.PrintMessage("Hello ProcessNet");~~
            ~~application.PrintMessage(application.ProcessNetVersion);~~
~~}~~

~~public void CreateBodyExample()~~
~~{~~
            ~~refFrame1 = modelDocument.CreateReferenceFrame();~~
            ~~refFrame1.SetOrigin(100, 0, 0);~~

            ~~refFrame2 = modelDocument.CreateReferenceFrame();~~
            ~~refFrame2.SetOrigin(0, 200, 0);~~

            ~~IBody body1 = model.CreateBodyBox("body1", refFrame1, 150, 100, 100);~~
            ~~application.PrintMessage(body1.Name);~~
            ~~IBody body2 = model.CreateBodySphere("body2", refFrame2, 50);~~
            ~~application.PrintMessage(body2.Name);~~
~~}~~
```

3. Create **RunChangeBody()** and **RunCreateContact()** functions as shown below.
   - Pass the value of MainWindow to the **ChangeGeneralBody** and **CreateContact** classes.

Note: The MainWindow value must be delivered to these classes to use ProcessNet methods in WinForms..

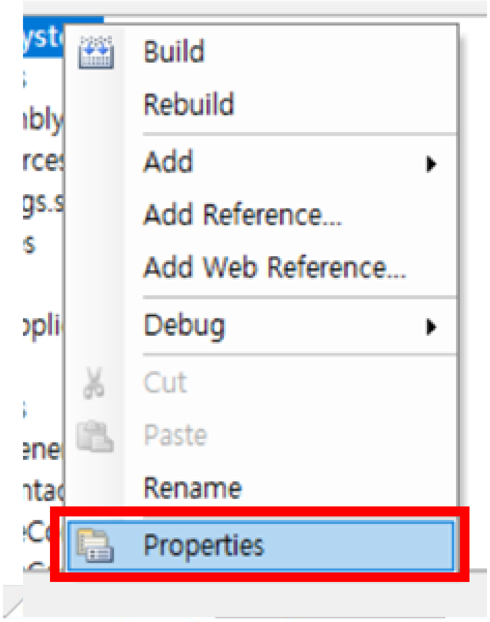
```
public void RunChangeBody()
{
            ChangeGeneralBody Dialog = new ChangeGeneralBody(modelDocument);
            Dialog.Show(MainWindow);
}

public void RunCreateContact()
{
            CreateContact Dialog = new CreateContact(modelDocument);
            Dialog.Show(MainWindow);
}
```

4. In the **File** menu, click **Save ThisApplication.cs** to save the file.

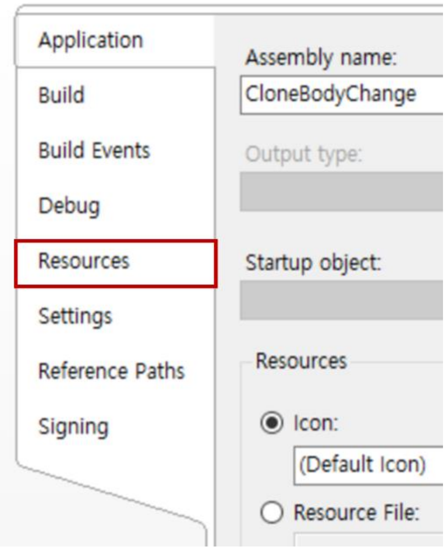
Registering Icons

When you add the **ChangeGeneralBody** and **CreateContact** dialog windows to the ribbon menu, you need to add icons that will be displayed on the ribbon.

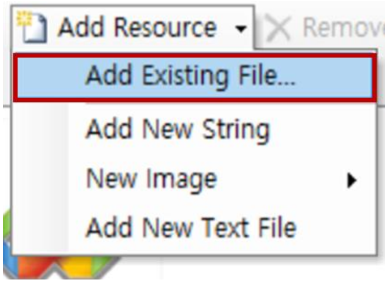
1. In **ProcessNet IDE**, in the Project Explorer window, **right-click SimpleBeltSystem**.
2. Click the **Properties** button.



3. When the **SimpleBeltSystem** Property window appears, click the Resource button.



4. When the **Resources** window opens, click **Add Resource-Add Existing File**.



5. Register **ConvertGeneral.bmp**. (Path of the file:<InstallDir>/Help/Tutorial/ProcessNet/VSTA/SimpleBeltSystem)
6. Go through the steps 4-5 again to register **CreateContact.bmp**.

Creating Functions for Register DLLs

1. Create the **RegisterFunction()** function. Beware that, if the name of the function is not **RegisterFunction**, the **Register DLL** does not work.

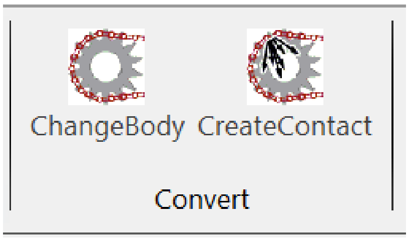
```
public void RegisterFunction()
{
            IRibbonManager ribbonManager = application.RibbonManager;
            IRibbonTab ribbonTab = ribbonManager.FindRibbonTab("Customize");
            IRibbonGroup ribbonGroup = ribbonTab.AddRibbonGroup("Convert");

            //ID for user created processNet function must be between 8000 - 8999 .
            IMenuControl menuControl01 =
ribbonGroup.AddMenuControl(MenuControlType.MenuControlType_Button, 8011);

            // Set example control information.
            IntPtr iIcon01 = SimpleBeltSystem.Properties.Resources.ConvertGeneral.GetHicon();
            menuControl01.SetIcon(iIcon01);
            menuControl01.UseBigIcon = true;

            menuControl01.Caption = "ChangeBody";
            menuControl01.Tooltip = "ChangeBody";
            //menuControl.Description = "MyDescription";
            menuControl01.UseProcessNetFunction = true; //if ID is not between 8000 - 8999, this will set
ID to 8000

            // Get current ProcessNet dll fullpath. 'ProcessNetDllPath' must be absolute path.
            string assemblyname =
System.Reflection.Assembly.GetExecutingAssembly().GetName().Name;
            string ProcessNetDllPath = AppDomain.CurrentDomain.BaseDirectory + @"\" + assemblyname
+ ".dll";

            menuControl01.ProcessNetDllPath = ProcessNetDllPath;
            menuControl01.ProcessNetFunctionName = "RunChangeBody";

            //ID for user created processNet function must be between 8000 - 8999 .
            IMenuControl menuControl02 =
ribbonGroup.AddMenuControl(MenuControlType.MenuControlType_Button, 8012);

            // Set example control information.
            IntPtr iIcon02 = SimpleBeltSystem.Properties.Resources.CreateContact.GetHicon();
            menuControl02.SetIcon(iIcon02);
            menuControl02.UseBigIcon = true;

            menuControl02.Caption = "CreateContact";
            menuControl02.Tooltip = "CreateContact";
            //menuControl.Description = "MyDescription";
            menuControl02.UseProcessNetFunction = true; //if ID is not between 8000 - 8999, this will set
ID to 8000

            menuControl02.ProcessNetDllPath = ProcessNetDllPath;
            menuControl02.ProcessNetFunctionName = "RunCreateContact";
}
```

Testing the Created Application

In this section, you will test whether the application you created works properly.

To run the application:

1. In the **Build** menu, click **Build SimpleBeltSystem**. Check if any errors or warnings appear in the **Error List** pane at the bottom of the **IDE** window. If there are any errors or warnings, correct the problems.
2. In **RecurDyn**, on the **Customize** tab, in the **ProcessNet(VSTA)** group, click **Run**.
3. In the tree in the lower half of the **Run ProcessNet** dialog window, click **RegisterFunction** under **SimpleBeltSystem**.
4. In the **Run ProcessNet** dialog window, click the **Run** button.
5. The icon is registered on the ribbon menu.



Chapter

6

Model Analysis

Task Objectives

In this chapter, use the ProcessNet code you have written to convert a clone link body to a general body and define a General Contact.

Estimated Time to Complete

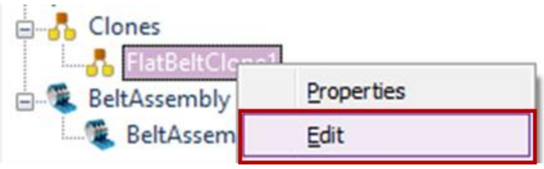
5 minutes

Converting Clone Link Body into General Body

To create FaceSurface in the clone link body:

Create a FaceSurface on the face where you want to create a Contact. Since you can also create a Contact on the face, create only one FaceSurface.

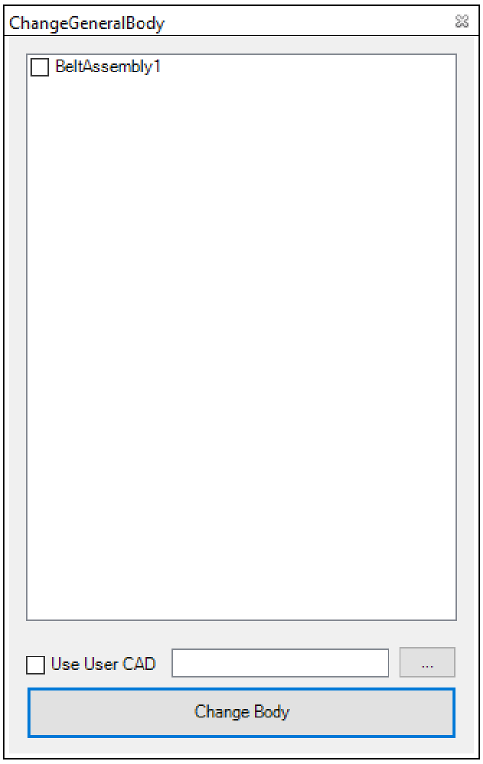
1. Enter the Belt Subsystem Edit Mode.
2. Use the following method to enter the Edit mode of the FlatBeltClone1 body.
   - In Database, right-click **FlatBeltClone1**.
   - When a pop-up menu appears, click **Edit**.



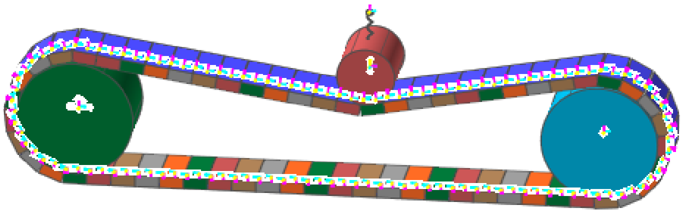
3. On the **Geometry** tab, in the **Surface** group, click **Face**.
4. Set the Creation Method to **Face**.
5. Enter **FlatBelt1.Face2** on the input bar.
6. Right-click on the working window and click **Exit** to exit the Edit mode.

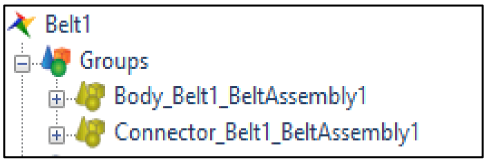
To convert the clone link body:

Convert the clone link of the open belt model to General using the ProcessNet functions you have written so far.



1. On the **Customize** tab, in the **Convert** group, click **ChangeBody**.
2. In the **ChangeGeneralBody** dialog window that opens, select **Belt Assembly1** and click the **ChangeBody** button.
3. You can see that the clone link body has been converted into a general body as shown in the following figure.
4. In the Database window, you can see that **Body_Belt1_BeltAssembly1** and **Connector_Belt1_BeltAssembly1** have been created. Select each group to see the result of creation.



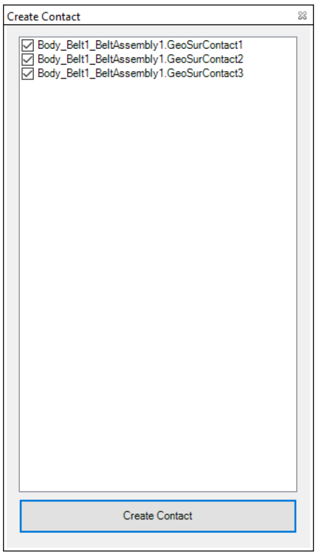


To create a Contact:

1. On the **Professional** tab, in the **Contact** group, click **GeoSur**.
2. Change the Creation Method to **Surface(PatchSet), Surface(PatchSet)**.
3. Create a Geo Surface Contact by typing the values in the table shown below.

Contact	Action	Base
GeoSurContact1	FlatBelt50.FaceSurface1	Roller3.Roller1.Face3
GeoSurContact2	FlatBelt16.FlatBelt1.Face4	Roller1.Roller1.Face3
GeoSurContact3	FlatBelt17.FlatBelt1.Face4	Roller2.Roller1.Face3

4. On the **Customize** tab, in the **Convert** group, click CreateContact.
5. Make sure that the created Contact is checked and click the Create Contact button.
6. Confirm that the Contact has been created.



☑ Body_Belt1_BeltAssembly1.GeoSurContact1
☑ Body_Belt1_BeltAssembly1.GeoSurContact2
☑ Body_Belt1_BeltAssembly1.GeoSurContact3

To run Dynamic / Kinematic analysis:

Run Dynamic/Kinematic analysis of the generated model.

1. In the **Simulation Type** group of the **Analysis** tab, click **Dyn/Kin** to open the Dynamic/Kinematic Analysis dialog window.
2. Set the simulation end time and the number of steps.
   - End Time:1
   - Step:100
   - Plot Multiplier Step Factor:1
3. Click **Simulate** to perform the analysis.
4. Compare the results with the existing **Belt Toolkit model**.

Thanks for participating in this tutorial!