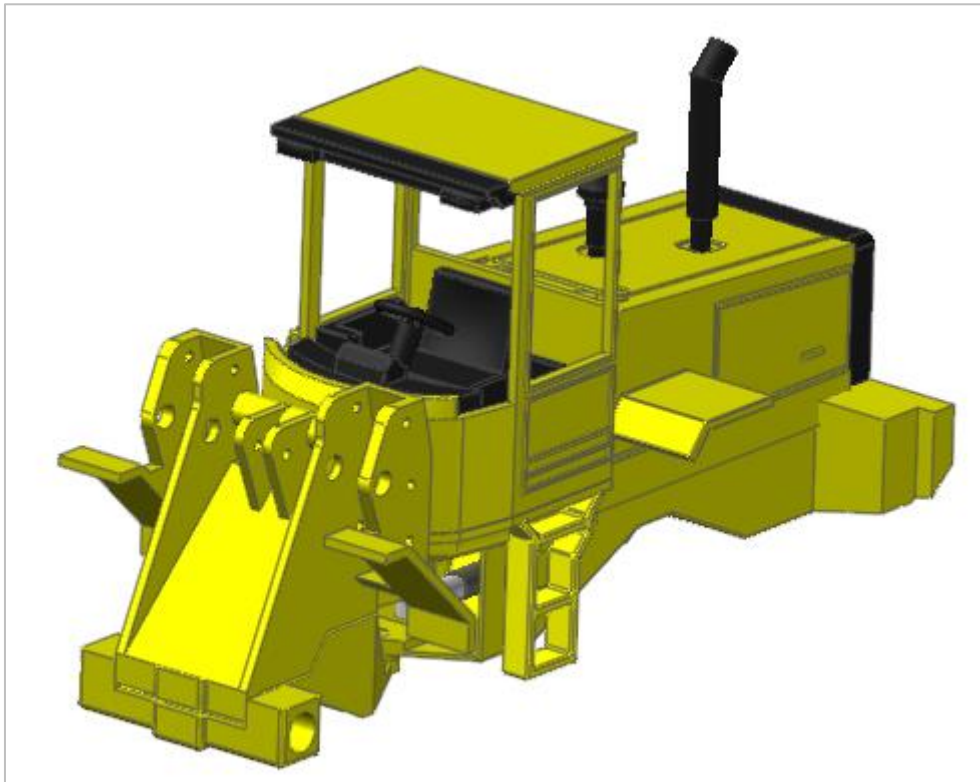




---

## **4WD Loader Tutorial (ProcessNet VSTA)**



**Copyright © 2020 FunctionBay, Inc. All rights reserved.**

User and training documentation from FunctionBay, Inc. is subjected to the copyright laws of the Republic of Korea and other countries and is provided under a license agreement that restricts copying, disclosure, and use of such documentation. FunctionBay, Inc. hereby grants to the licensed user the right to make copies in printed form of this documentation if provided on software media, but only for internal/personal use and in accordance with the license agreement under which the applicable software is licensed. Any copy made shall include the FunctionBay, Inc. copyright notice and any other proprietary notice provided by FunctionBay, Inc. This documentation may not be disclosed, transferred, modified, or reduced to any form, including electronic media, or transmitted or made publicly available by any means without the prior written consent of FunctionBay, Inc. and no authorization is granted to make copies for such purpose.

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.

The software described in this document is provided under written license agreement, contains valuable trade secrets and proprietary information, and is protected by the copyright laws of the Republic of Korea and other countries. UNAUTHORIZED USE OF SOFTWARE OR ITS DOCUMENTATION CAN RESULT IN CIVIL DAMAGES AND CRIMINAL PROSECUTION.

**Registered Trademarks of FunctionBay, Inc. or Subsidiary**

**RecurDyn** is a registered trademark of FunctionBay, Inc.

RecurDyn/Professional, RecurDyn/ProcessNet, RecurDyn/Acoustics, RecurDyn/AutoDesign, RecurDyn/Bearing, RecurDyn/Belt, RecurDyn/Chain, RecurDyn/CoLink, RecurDyn/Control, RecurDyn/Crank, RecurDyn/Durability, RecurDyn/EHD, RecurDyn/Engine, RecurDyn/eTemplate, RecurDyn/FFlex, RecurDyn/Gear, RecurDyn/DriveTrain, RecurDyn/HAT, RecurDyn/Linear, RecurDyn/Mesher, RecurDyn/MTT2D, RecurDyn/MTT3D, RecurDyn/Particleworks I/F, RecurDyn/Piston, RecurDyn/R2R2D, RecurDyn/RFlex, RecurDyn/RFlexGen, RecurDyn/SPI, RecurDyn/Spring, RecurDyn/TimingChain, RecurDyn/Tire, RecurDyn/Track\_HM, RecurDyn/Track\_LM, RecurDyn/TSG, RecurDyn/Valve are trademarks of FunctionBay, Inc.

**Edition Note**

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

# Table of Contents

Getting Started .....	4
Objective .....	4
Approach .....	5
Audience .....	5
Prerequisites.....	5
Procedures .....	5
Estimated Time to Complete .....	5
Opening the Model and Initializing ProcessNet.....	6
Task Objective .....	6
Estimated Time to Complete .....	6
Starting RecurDyn .....	7
Initializing ProcessNet.....	8
Automating Contact Definition.....	10
Task Objective .....	10
Estimated Time to Complete .....	10
Understanding the Contacts to be Created.....	11
Using the ProcessNet Help Examples .....	12
Using the ProcessNet Template .....	13
Creating the Base Application .....	13
Coding Using IntelliSense .....	16
Building and Running the Macro .....	18
Running a Simulation.....	19
Viewing the Results .....	19
Adding Additional Contacts .....	21
Repeating the Build, Simulation, and Viewing Processes .....	22
Adding a Dialog and Message Output .....	23
Task Objective .....	23
Estimated Time to Complete .....	23
Designing a Dialog .....	24
Defining the Behavior of the Dialog .....	28
Displaying the Dialog when Running the Macro .....	29
Test the Dialog Box .....	32
Automating Plot Creation.....	33
Task Objective .....	33
Estimated Time to Complete .....	33
Creating a Dialog .....	34
Plotting the Contact Forces .....	37
Improving the Contact Force Plot .....	40
Improving the Plot Formatting .....	43
Plotting the Total X, Y, and Z Contact Force.....	45

## Chapter

## 1

## Getting Started

### Objective

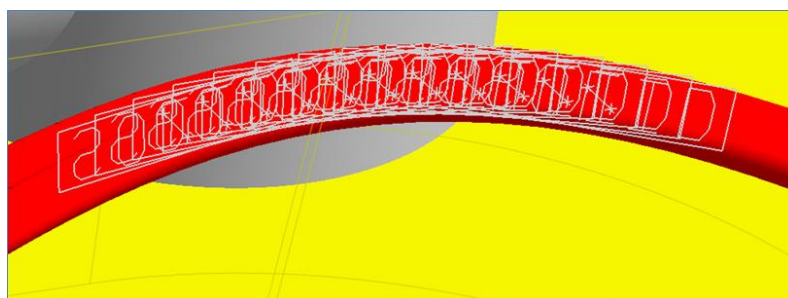
In this tutorial, you will automate three RecurDyn processes using **ProcessNet**. **ProcessNet** is used to develop automated procedures for model building, analysis, and plotting. The three processes are:

- Automate the creation of a series of contacts.
- Create a custom dialog box that provides user control of the contact.
- Automatically check the contact force outputs and only plot the contact forces that have a non-zero output.

You simulate a pair of hoses on a 4WD loader, as shown on the cover of the tutorial. The hoses connect the hydraulic pump at the rear of the vehicle to the cylinders of the loader linkage at the front of the vehicle. The loader steers by articulation. The hoses bend as the two sections of the frame articulate and may contact each other.



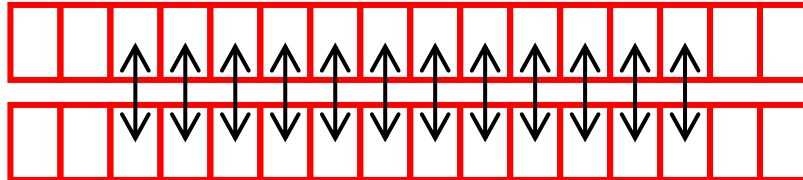
It is important to understand the contact to predict any binding or wear problems. The motion is not trivial—it is hard to predict the motion of the hose and the contact location. You want to know where the contact occurs and the magnitude of the contact force. Therefore, you need to define contacts between many of the segments of the two hoses (see the figure below). It would be a tedious job to do manually and you could make some mistakes. An automated process would be much more efficient



## Approach

You will start with a loader model that includes all of the mechanical components, including the hoses and the motion controls for the model. The hoses were defined using the RFlex Beam Group, using two points and 50 segments. The front attachment points of the hoses are moved backwards during the first 0.18 seconds of the simulation to develop the proper slack in the hoses. Another motion input articulates the frame by lengthening one of the steering cylinders.

The initial set of contacts will be defined between corresponding segments of the two hoses, as shown in the figure below:



## Audience

This tutorial is intended for intermediate users of RecurDyn who previously learned how to create geometry, joints, and force entities. All new tasks are explained carefully.

## Prerequisites

You should first work through the **3D Crank-Slider** and **Engine with Propeller tutorials**, or the equivalent. We assume that you have a basic knowledge of physics.

You will need a license for the RFlex module in order to run this model because the hoses are defined using the RFlex Beam Group.

## Procedures

The tutorial is comprised of the following procedures. The estimated time to complete each procedure is shown in the table.

Procedures	Time (minutes)
Opening model and Initializing ProcessNet	10
Automating Contact Definition	35
Adding a Dialog Box and Message Output	20
Automating Plot Creation	20
<b>Total</b>	<b>85</b>



## Estimated Time to Complete

This tutorial takes approximately 85 minutes to complete.

## Opening the Model and Initializing ProcessNet

### Task Objective

Learn how to open a mechanical model in preparation for using **ProcessNet** and learn how to initialize **ProcessNet**.



### Estimated Time to Complete

10 minutes

## Starting RecurDyn

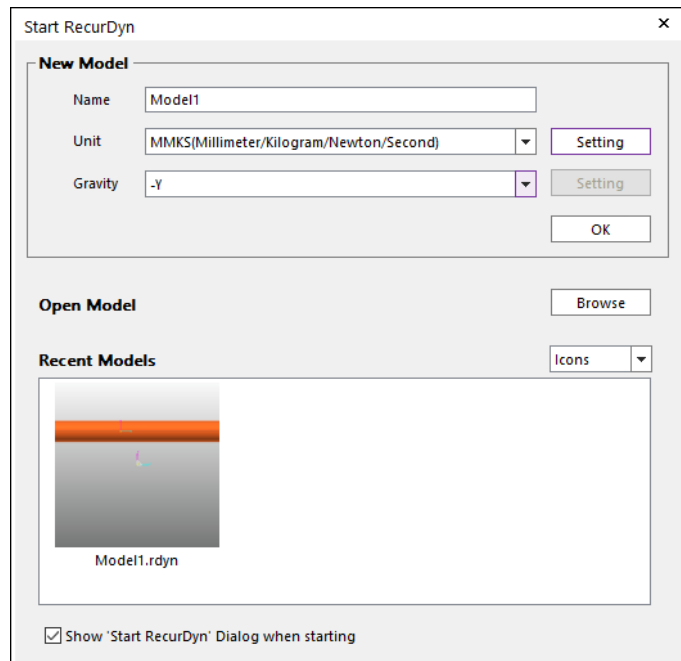
To start RecurDyn and open the initial model:



1. On your Desktop, double-click the **RecurDyn** icon.
2. When the Start RecurDyn dialog box appears, close this because you will not be creating a new model but using an existing one.



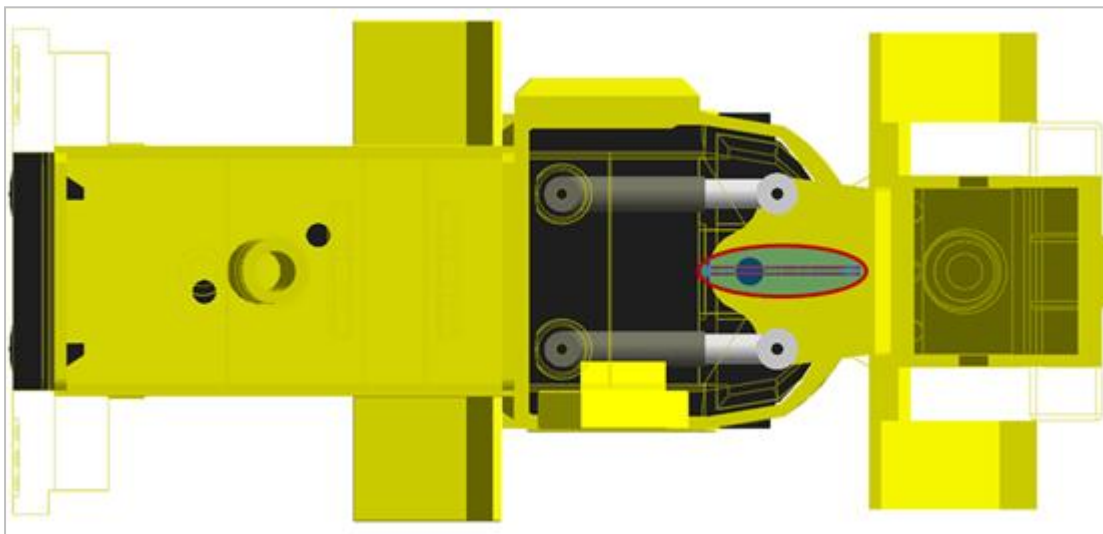
3. From the **Quick Access Toolbar**, click **Open**.
4. Select the file **4WD Loader\_Start.rdyn**.




(Thefilelocation: <InstallDir>/Help/Tutorial/ProcessNet/VSTA/4WDLoader).

5. Click **Open**.

Your model should look like the following.



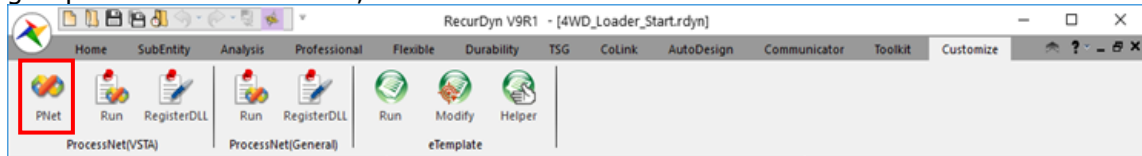
The red lines are the hydraulic hoses that you will study.

**Tip:** Note that this view is shown in the  (**Render Each Object**) viewing mode so that you can see past the components on the bottom of the vehicle that might obscure your view of the hoses. If you have difficulty seeing the hoses, you may want to check if you are in the Shaded viewing mode. If so, change back to the Render Each Object viewing mode.

## Initializing ProcessNet

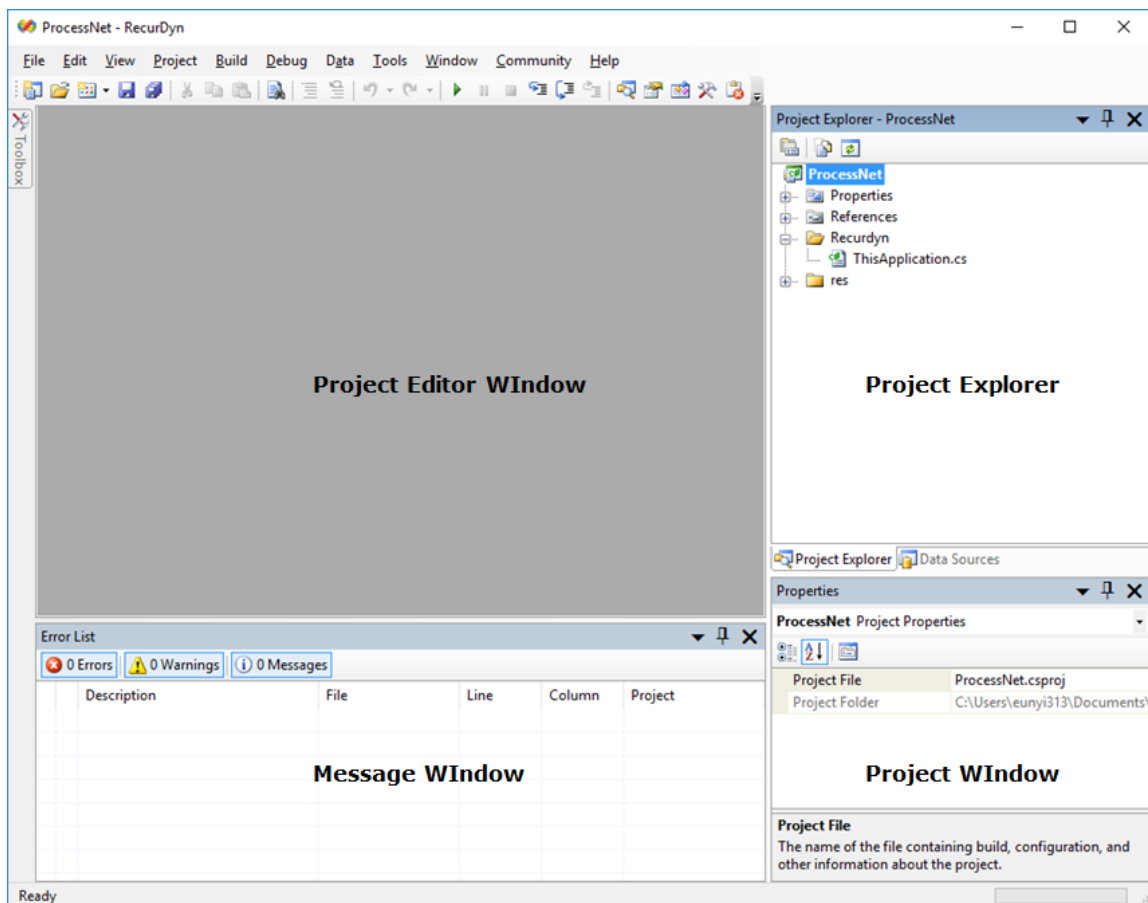
To start ProcessNet and perform initialization:

1. To enter the Integrated Development Environment (IDE), from the **ProcessNet(VSTA)** group of the Customize tab, select **PNet**.



A small window appears that tells you that RecurDyn is **Initializing ProcessNet**. It may take a few minutes to perform the entire initialization process if this is the first time that the **ProcessNet IDE** is used after the RecurDyn installation. Otherwise, the initialization takes 10 to 15 seconds.

When the initialization is complete, a new window appears with the heading **ProcessNet – RecurDyn**, as shown below.



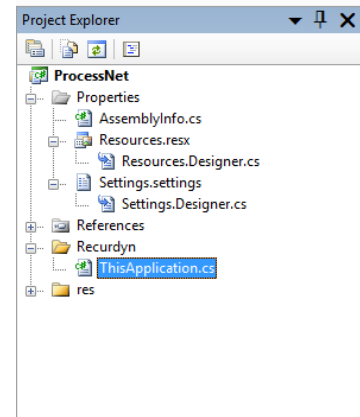
It contains four areas that let you:

- **Project Editor Window** – Write and edit code and design interface objects.
- **Project Explorer** – Get an organized view of your project and its files.
- **Properties window** – View and edit the properties of objects you selected in the Project Editor window or Project Explorer
- **Message window** – View information that is generated as you build and run your code, such as errors in your code.



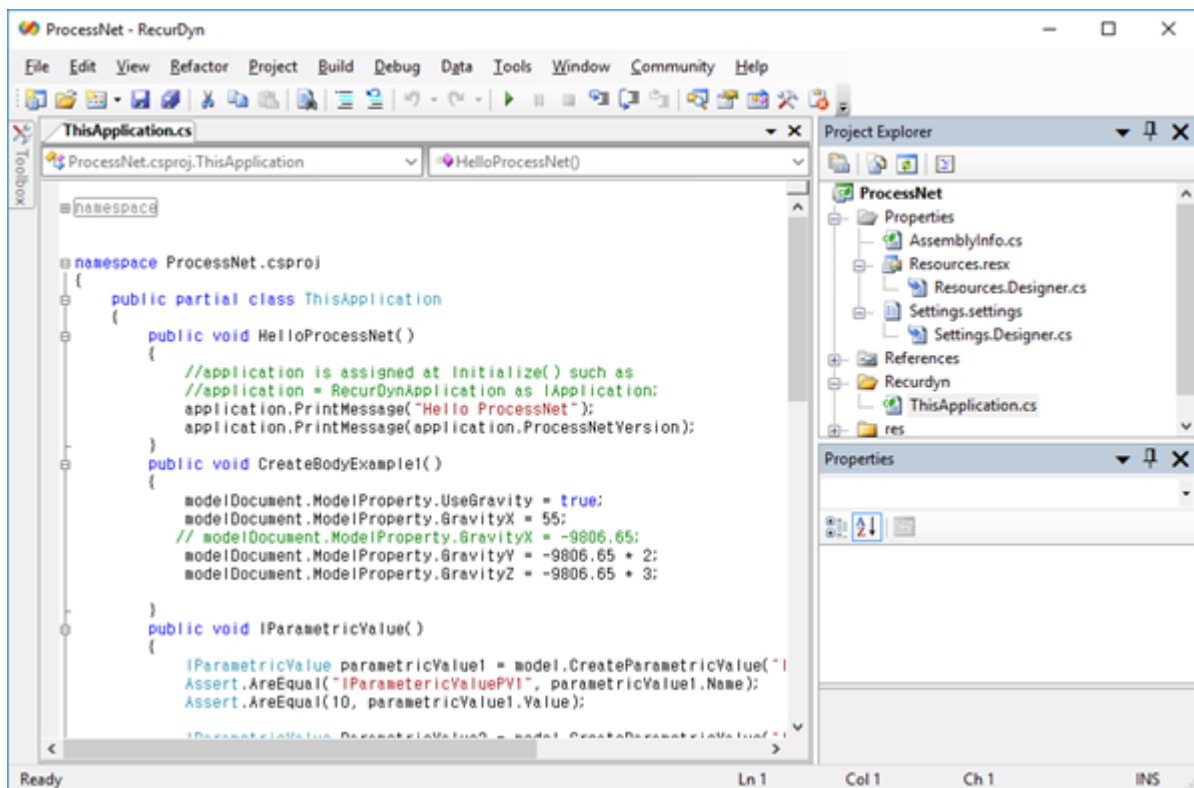
The Project Explorer should contain a **Recurdyn** folder, with **ThisApplication.cs**, as shown in the figure on the right.

- If this is what you see, continue to **Step2** below.
- If the contents of the window are different, it may be because the default Project directory was used by another user. If so, create your own Project space.



**Note:** The location of the default Project directory is in the **ProcessNetProject** directory in the Recurdyn installation directory. The name of the default Project Directory is **ProcessNet**.

- a. Make a copy of the **ProcessNet** directory and place it anywhere on your computer. You can name the directory as you would like.
  - b. Return to the **ProcessNet IDE** application.
  - c. From the **File** menu, select **Close Project**.
  - d. Select Open Project and select the **ProcessNet.csproj** file in your new directory that you created in Step a.
  - e. Delete any items in the Project Explorer window that are not in the figure above.
2. In the **Project Explorer** window, double-click the file **ThisApplication.cs** to open it. The content of the file appears in the Project editor window.



You are now ready to start developing your first **ProcessNet** application.

3. From the **File** menu, Select **Save All** as **4WD\_Loader**.

## Chapter

## 3

## Automating Contact Definition

### Task Objective

In two steps, you will create a **ProcessNet** application that creates a set of contacts between the segments of the hoses. You will learn how to:

- Obtain the necessary information to create a RecurDyn entity.
- Use the **ProcessNet** help in RecurDyn.
- Work in the IDE to develop an application.

You will then run a simulation, view the results, update your application, and rerun the simulation to observe the improved results.

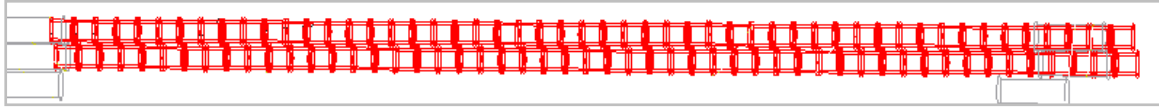


### Estimated Time to Complete

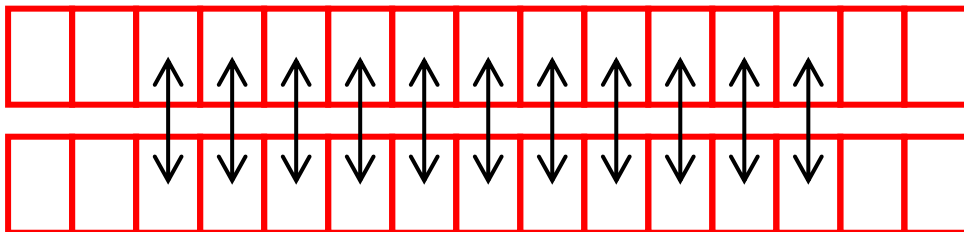
35 minutes

## Understanding the Contacts to be Created

Chapter 1 explained that each of the two hoses have 50 segments, as shown in the figure below. The hoses are separated at the mounting locations at each end. The centerline of the vehicle runs between the hoses. When articulation occurs one hose tends to be stretched more and the other hose stretched less. With the curving of both hoses being different, they can make contact in the middle of the span of the hose.



For your first **ProcessNet** application, you will create 11 contacts in the middle of the span of the hoses as shown in the figure below.



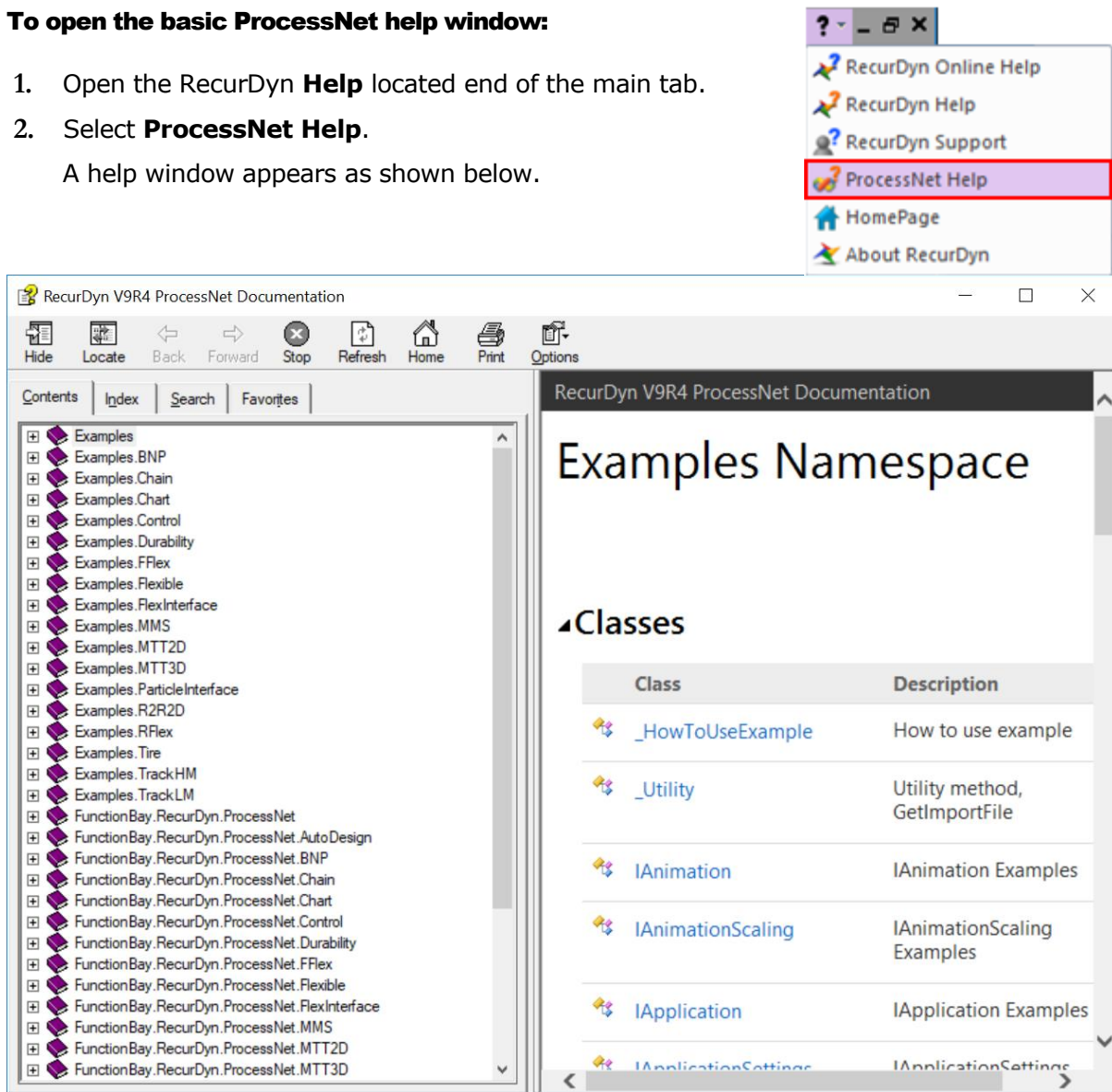
## Using the ProcessNet Help Examples

In this section, you will learn what kind of information is available to help you create your ProcessNet application.

### To open the basic ProcessNet help window:

1. Open the RecurDyn **Help** located end of the main tab.
2. Select **ProcessNet Help**.

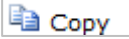
A help window appears as shown below.



Here, you will find various code examples. If you are developing basic RecurDyn macros without using the specialized toolkits, you will find the following namespaces most useful:

- **Examples Namespace** - Contains actual examples of all or a portion of basic **ProcessNet** applications, as well as examples of creating basic **RecurDyn** entities and parameters.
  - **Examples.Plot Namespace** - Contains examples of creating **RecurDyn** plots.
3. See one of the examples, by expanding:  
**Examples Namespace** → **IBody Class** → **IBodyExamples Method** (Tip: You can also select the link with the same name in the window to the right.)

This particular method contains examples of how to create basic RecurDyn bodies such as spheres, boxes, and cylinders, and how to modify their parameters.

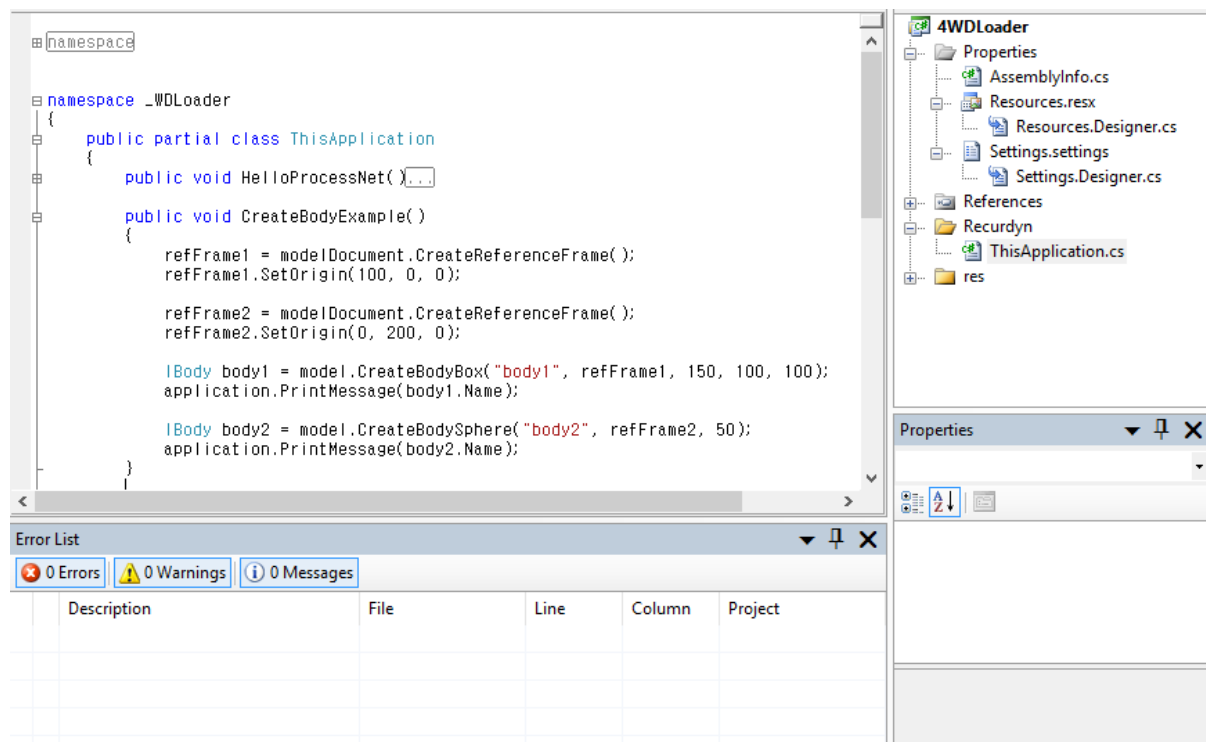
**Tip:** As you develop your own macros, you can use the  tool on the right side of the screen to copy entire sections of example code.

4. Close the ProcessNet Help window.

## Using the ProcessNet Template

Before you start coding, take a look at the existing code which is part of the RecurDyn ProcessNet template for Visual C#. There is a sample macro called **HelloProcessNet()**. This is a simple program which outputs "Hello ProcessNet" to the RecurDyn message window.

Pressing on the + signs on the left side of the screen expands code enclosed in sections. Use this method to expand the RecurDyn generated code, further below the **HelloProcessNet()** macro. You will see that there are several commonly used variables which are declared, and some of them are initiated in the **Initialize()** method. This code makes it convenient for you, as the developer, to easily use these variables in your own code without having to declare and initialize them.



## Creating the Base Application

You will now develop the application to create the hose contacts.

### To create the base application:

1. Copy the following code, and insert it after the **HelloProcessNet()** macro, as shown below.

```
public void ProcessNetTutorialCreateSolidContact()
{
}
```

You have now created a stub for your macro. If you were to compile your code now, the method would show up in the list of available macros to run in RecurDyn, although it would not do anything.

- Next, insert the following variable declarations into the macro stub you just created (below the opening curly brace, and above the closing curly brace):

```
{
int BodyNumStart = 20; // Start creating contacts with body 20
int BodyNumEnd = 30; // Continue until body 30
int BodyInterval = 51; // Interval between body number on hose 1
                        // and corresponding body's number on
                        // hose 2 is 51
}
```

Note that each variable is followed by an explanation, preceded with two forward slashes, `//`. In C#, any characters after these two slashes are ignored by the compiler and can be used as comments in the code, for documentation purposes.

---

**Note:** Comments in C# can be preceded by two slashes, `//`.

---

- After the variable declarations you just added, add the following for loop:

```
public void ProcessNetTutorialCreateSolidContact()
{
    int BodyNumStart = 20; // Start creating contacts with body 20
    int BodyNumEnd = 30; // Continue until body 30
    int BodyInterval = 51; // Interval between body number on hose 1
                        // and corresponding body's number on
                        // hose 2 is 51

    for (int i = BodyNumStart; i <= BodyNumEnd; i++)
    {
    }
}
```

- The code placed inside the for loop will repeat. For the first loop, the index `i` will be equal to **BodyNumStart**. Then `i` will be incremented by 1, and the next loop will execute. This will repeat while the condition `i ≤ BodyNumEnd` is true.
- Note that `i` is declared within the **for** loop statement, meaning that it will be valid only for code within the for loop.
- Also note that, in C#, the syntax `i++` means to increment `i` by 1, or `i = i + 1`, after it has been used in the loop.

4. Within the for loop you just added, insert the following code:

```
public void ProcessNetTutorialCreateSolidContact()
```

```
public void ProcessNetTutorialCreateSolidContact()
{
    int BodyNumStart = 20; // Start creating contacts with body 20
    int BodyNumEnd = 30; // Continue until body 30
    int BodyInterval = 51; // Interval between body number on hose 1
                          // and corresponding body's number on
                          // hose 2 is 51

    for (int i = BodyNumStart; i <= BodyNumEnd; i++)
    {
        int j = i + BodyInterval; // j is the index for the
                                // corresponding bodies on hose #2
                                // Do the contact for corresponding bodies

        IBody baseBody = model.GetEntity("BeamBody" + i.ToString()) as IBody;
        IGeometry baseGeom = baseBody.GetEntity("HollowCircularBeam1") as IGeometry;
        IBody actionBody = model.GetEntity("BeamBody" + j.ToString()) as IBody;
        IGeometry actionGeom = actionBody.GetEntity("HollowCircularBeam1") as IGeometry;
        IContactSolidContact solidContact = model.CreateContactSolidContact("solidContact"
        + i.ToString(), baseGeom, actionGeom);
    }
}
```

An explanation of this code block is as follows:

```
int j = i + BodyInterval; // j is the index for the
                        // corresponding bodies on hose #2
```

In the code above:

- **i** is the index for the bodies in hose 1, and **j** is the index for the bodies in hose 2.
- To make sure that the bodies correspond, the value of **j** will always be equal to **i + bodyInterval**.
- Also, note that **j** is being declared inside the for loop, and therefore will not be valid outside the for loop.

```
IBody baseBody
    = model.GetEntity("BeamBody" + i.ToString()) as IBody;
```

The code above retrieves the body that we want to use as the base body by its name:

- The **+** character is used to join two strings.
- The **ToString()** method is used to convert the integer value of **i** to a string.
- Therefore, the **GetEntity()** method looks for bodies by the names "BeamBody20", "BeamBody21", etc.
- Because the **GetEntity()** method can return any generic entity, the call must be explicitly casted as **IBody** to tell the compiler what type to return.

```
IGeometry baseGeom
    = baseBody.GetEntity("HollowCircularBeam1") as IGeometry;
```

▪

The above code retrieves geometry of the base body by its name, "HollowCircularBeam1". This geometry will be used to create the solid contact.

The previous two steps are repeated for the action body.

```
IContactSolidContact solidContact
    = model.CreateContactSolidContact("solidContact"
    + i.ToString(), baseGeom, actionGeom);
```

Finally, the command above creates the solid contact, naming it "solidContact20" (or "solidContact21", "solidContact22", etc.). The command uses the base and action geometry we defined in the previous steps.

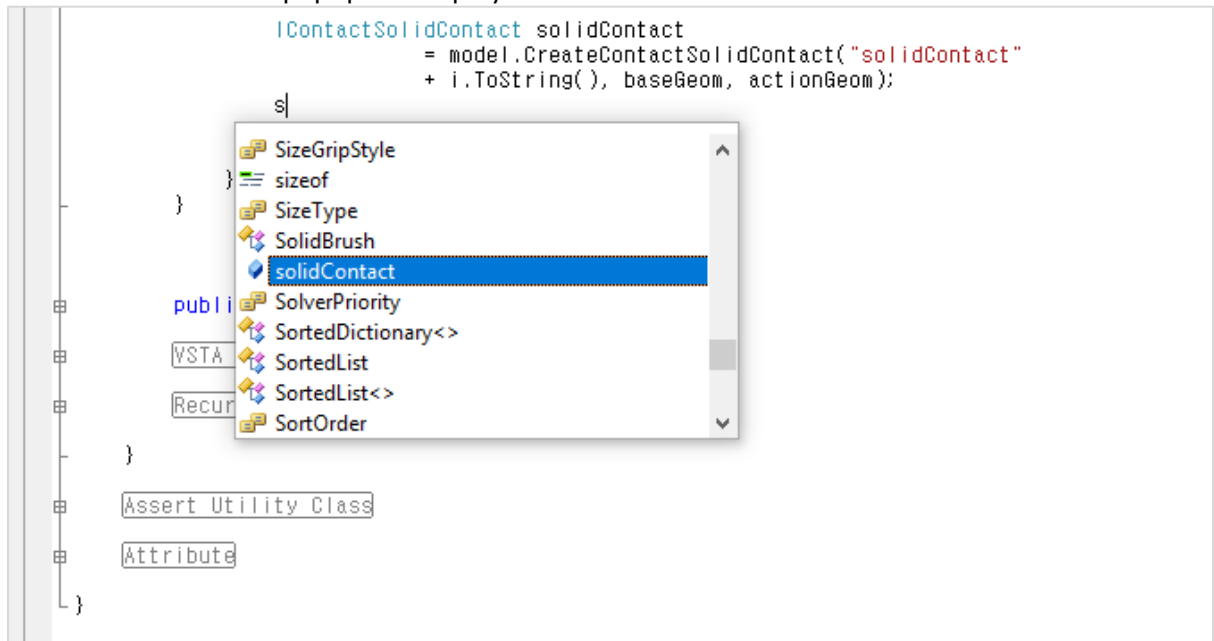
## Coding Using IntelliSense

Up to this point, you may have been cutting and pasting code directly from this tutorial. Once you start writing your own macros, however, you will be manually typing in code. A feature of the ProcessNet IDE which helps you code faster is called IntelliSense, and you will now explore how to take advantage of this while also accomplishing the following task.

If compiled, the macro should now create a contact with the default properties. But the default stiffness and damping values for the Solid Contact are too high. You will now set these contact parameters to lower values.

### To modify the contact parameters using IntelliSense:

1. After the last line of code, you entered in the previous steps, begin typing the character 's'. You should see a popup list display as shown below:



The list contains all of the valid things you could possibly type next, including names of variables accessible at this part of the code, methods you could call, etc.

2. From the IntelliSense list, locate **solidContact**.



3. Select **solidContact** by double-clicking on it or by pressing **Enter**.
4. Next, type the period character (.).
5. Select **ContactProperty** from the IntelliSense list.
6. Continue this procedure until you have typed:

```
solidContact.ContactProperty.StiffnessCoefficient.Value =1000;
```

7. Repeat this procedure to type the next line:

```
solidContact.ContactProperty.DampingCoefficient.Value = 0.1;
```

In the end, the method should appear in its entirety as shown below:

```
public void ProcessNetTutorialCreateSolidContact()
{
    int BodyNumStart = 20; // Start creating contacts with body 20
    int BodyNumEnd = 30; // Continue until body 30
    int BodyInterval = 51; // Interval between body number on hose 1
                        // and corresponding body's number on
                        // hose 2 is 51

    for (int i = BodyNumStart; i <= BodyNumEnd; i++)
    {
        int j = i + BodyInterval; // j is the index for the
                                // corresponding bodies on hose #2
                                //Do the contact for corresponding bodies

        IBody baseBody = model.GetEntity("BeamBody" + i.ToString()) as IBody;
        IGeometry baseGeom = baseBody.GetEntity("HollowCircularBeam1") as IGeometry;
        IBody actionBody = model.GetEntity("BeamBody" + j.ToString()) as IBody;
        IGeometry actionGeom = actionBody.GetEntity("HollowCircularBeam1") as IGeometry;
        IContactSolidContact solidContact = model.CreateContactSolidContact("solidContact"
        + i.ToString(), baseGeom, actionGeom);

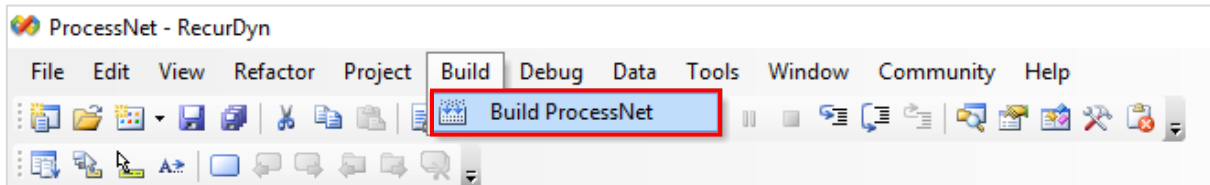
        solidContact.ContactProperty.StiffnessCoefficient.Value =1000;
        solidContact.ContactProperty.DampingCoefficient.Value = 0.1;
    }
}
```

There should be no errors or warnings in the **Error List** window at the bottom of the IDE window.

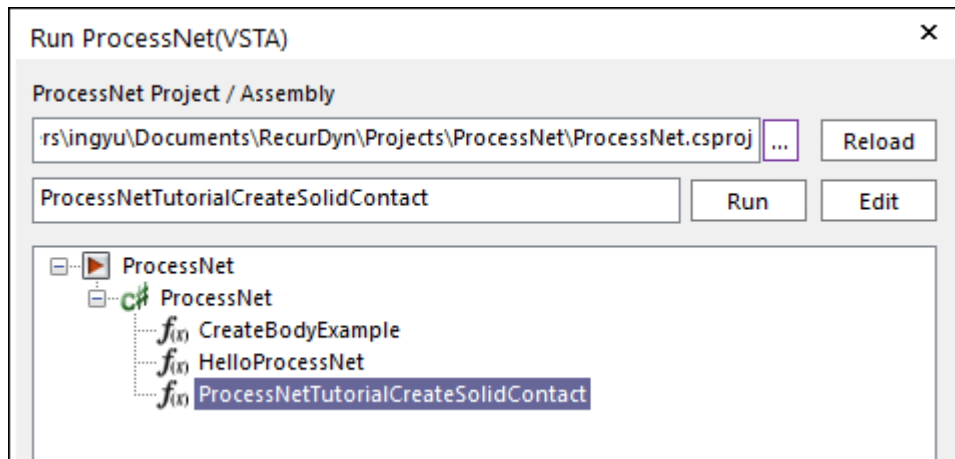
## Building and Running the Macro

To build and run the macro:

1. If there are any errors or warnings, check the listing and make any corrections.
2. From the **Build** menu, select **Build ProcessNet**.



3. Return to RecurDyn, and from the **ProcessNet(VSTA)** group in the **Customize** tab, click **Run..**
4. Select **ProcessNetTutorialCreateSolidContact** from the list of macros. Its name is loaded in the field next to the Run button.
5. Click **Run**.



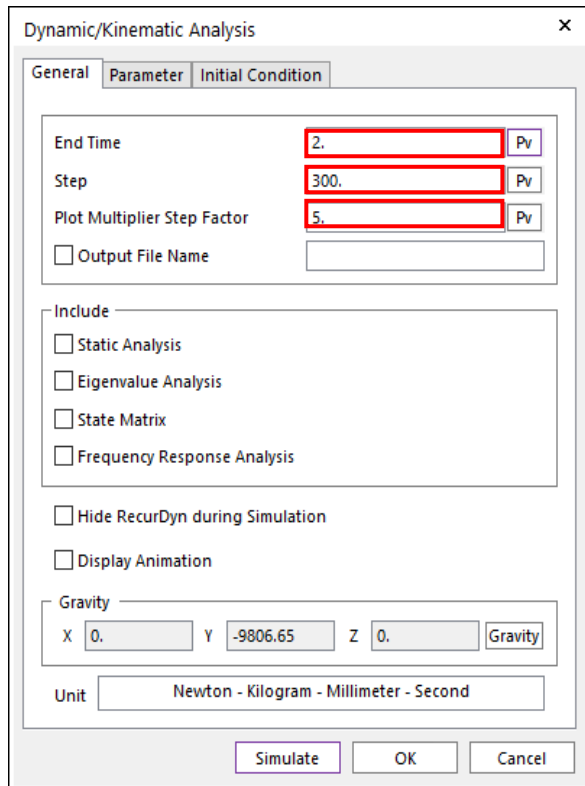
Eleven contacts are added to your model.

## Running a Simulation

You are now ready to run a simulation.

### To run a simulation:

1. From the **Simulation Type** group in the **Analysis** tab, click **Dynamic/Kinematic**.
2. Set the simulation to run for **2.0** seconds with **300** steps and a Plot **Multiplier Step Factor** of **5** as shown in the figure on the right.
3. Click **Simulate**. The simulation will run in 3-4 minutes, depending on the speed of your computer.



## Viewing the Results

### To view the results:

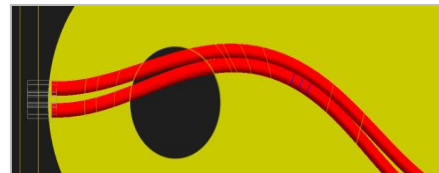
1. Set up the model so you are looking at a bottom view of the vehicle with the Render Each shading mode turned on.



2. In the Simulation toolbar, in the animation controls, click the **Play** button.



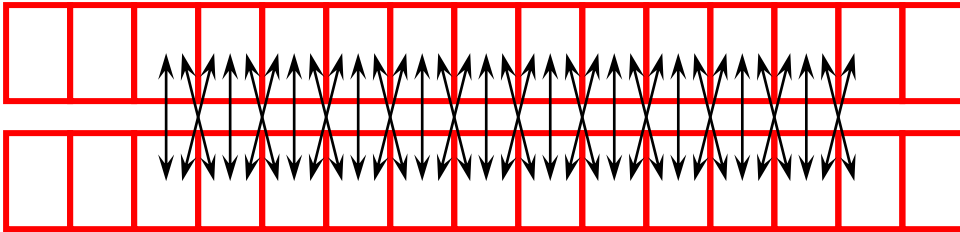
At the end of the simulation the hoses are deformed as shown in the figure to the right.



3. Turn on the action force display of all of the contacts by:
  - Selecting the first Solid contact in the Database Window.
  - While holding down the **Shift** key, select the last Solid contact in the Database Window.
  - Click the right mouse button and select **Property**.
  - In the dialog box that appears, click on the Solid tab. At the bottom of the dialog box use the pulldown menu to set the **Force Display** to **Action**.
  - Click **OK**.

4. Play the animation again and you will see that the forces are slightly erratic.

As you watch the motion you can see that the hoses slide with respect to each other. It may not be enough to have contact between corresponding segments. Contact with neighboring segments may also be required. This is shown in the figure below.



## Adding Additional Contacts

### To add additional contacts:

1. Modify the code as shown below:

```

IContactSolidContact solidContact = model.CreateContactSolidContact("solidContact"
+ i.ToString(), baseGeom, actionGeom);
solidContact.ContactProperty.StiffnessCoefficient.Value = 1000;
solidContact.ContactProperty.DampingCoefficient.Value = 0.1;

// Do the contact for body i+1 and body j
solidContact = model.CreateContactSolidContact("solidContact" + i.ToString() + "a",
(model.GetEntity("BeamBody" + Convert.ToString(i + 1)) as IBody).
GetEntity("HollowCircularBeam1") as IGeometry,
(model.GetEntity("BeamBody" + j.ToString()) as IBody).GetEntity("HollowCircularBeam1")
as IGeometry);
solidContact.ContactProperty.StiffnessCoefficient.Value = 1000;
solidContact.ContactProperty.DampingCoefficient.Value = 0.1;

// Do the contact for body i and body j+1
solidContact = model.CreateContactSolidContact("solidContact" + Convert.ToString(i) + "b",
(model.GetEntity("BeamBody" + i.ToString()) as IBody).GetEntity("HollowCircularBeam1")
as IGeometry, (model.GetEntity("BeamBody" + Convert.ToString(j + 1))
as IBody).GetEntity("HollowCircularBeam1") as IGeometry);
solidContact.ContactProperty.StiffnessCoefficient.Value = 1000;
solidContact.ContactProperty.DampingCoefficient.Value = 0.1;
}
}

```

Here, two more contacts are created for each loop through the code. The contact between the  $i+1^{\text{th}}$  segment on hose 1 and the  $j^{\text{th}}$  segment on hose 2 is named with an "a" suffix (i.e. "solidContact20a"). Similarly, the contact between the between the  $i^{\text{th}}$  segment on hose 1 and the  $j+1^{\text{th}}$  segment on hose 2 is named with a "b" suffix (i.e. "solidContact20b").

---

**Note:** Contact creation will not happen if a contact with the same name already exists.



---

- You might notice that the syntax for creating these additional contacts is in a more compact, efficient form. For example, retrieving the action body and geometry are all done within the **CreateContactSolidContact()** method, whereas earlier separate temporary variables were declared and assigned to store the value of each. This syntax makes it more efficient to code, but you need to make sure that you cast the return object of the methods explicitly with the as clause (i.e. as **IBody** or as **IGeometry**).

## Repeating the Build, Simulation, and Viewing Processes

Repeat the earlier steps to determine the effect on the model from the changes you just made.

### To repeat the processes:

1. Ensure there are no errors or warnings in the Error List window at the bottom of the IDE window. If there are, check the listing and make any corrections. From the **Build** menu, select **Build ProcessNet**.
2. Return to RecurDyn and delete all of the contacts that were defined for the previous run.
3. From the **ProcessNet(VSTA)** group in the **Customize** menu, select **Run**.
4. Click **Run**.  
33 contacts are added to your model.
5. Rerun the simulation with the same parameters as the previous run. With the additional contacts the simulation will run in 4-5 minutes.
6. Turn on the action force display of all of the contacts using the same procedure that was described earlier.
7. In the Simulation toolbar, in the animation controls, click the **Play**  button, again looking at a bottom view of the vehicle with the **Render Each shading**  mode turned on.



You will see that the contact forces are now much smoother.

## Adding a Dialog and Message Output

In this chapter, you will create a dialog window that will allow the user to control which segments to create contacts between. This will include designing the layout of the dialog and adding code to the existing subroutine to call the new dialog.

### Task Objective

Learn how to increase the flexibility of your application by creating a dialog window that will allow the user to control how the hose-to-hose contacts are created. Also, learn how to display a message to the user in the RecurDyn message window.



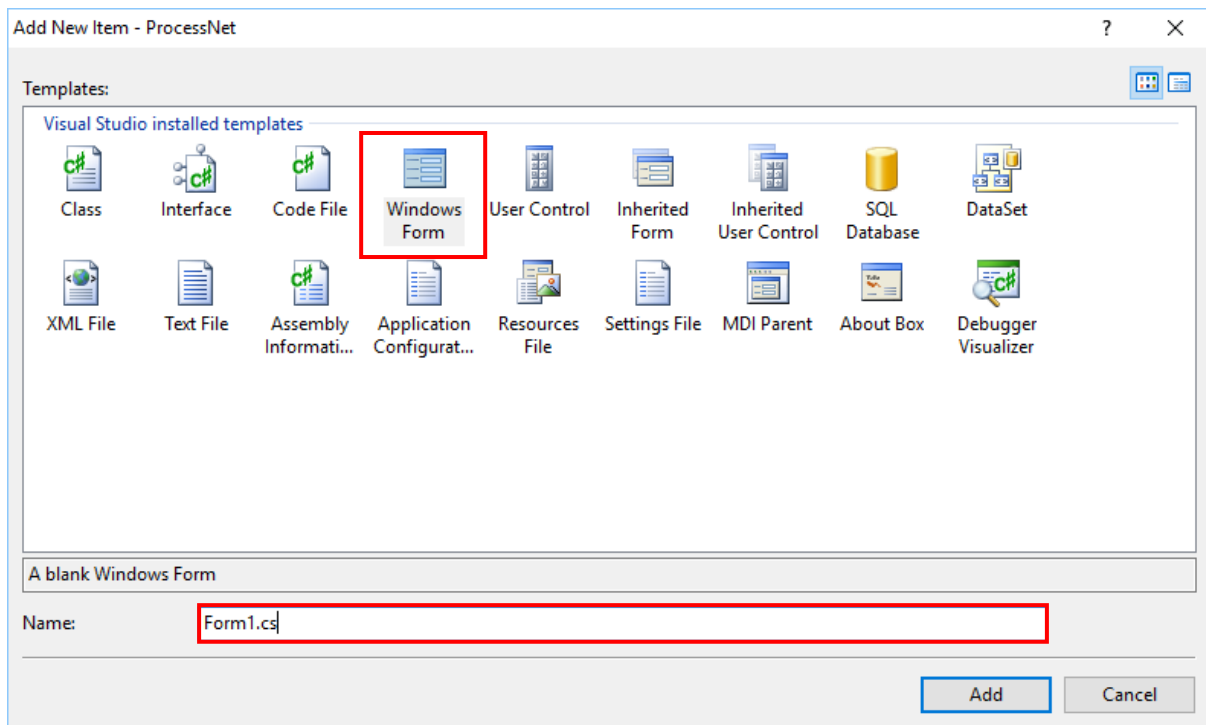
### Estimated Time to Complete

20 minutes

## Designing a Dialog

To design a new dialog window:

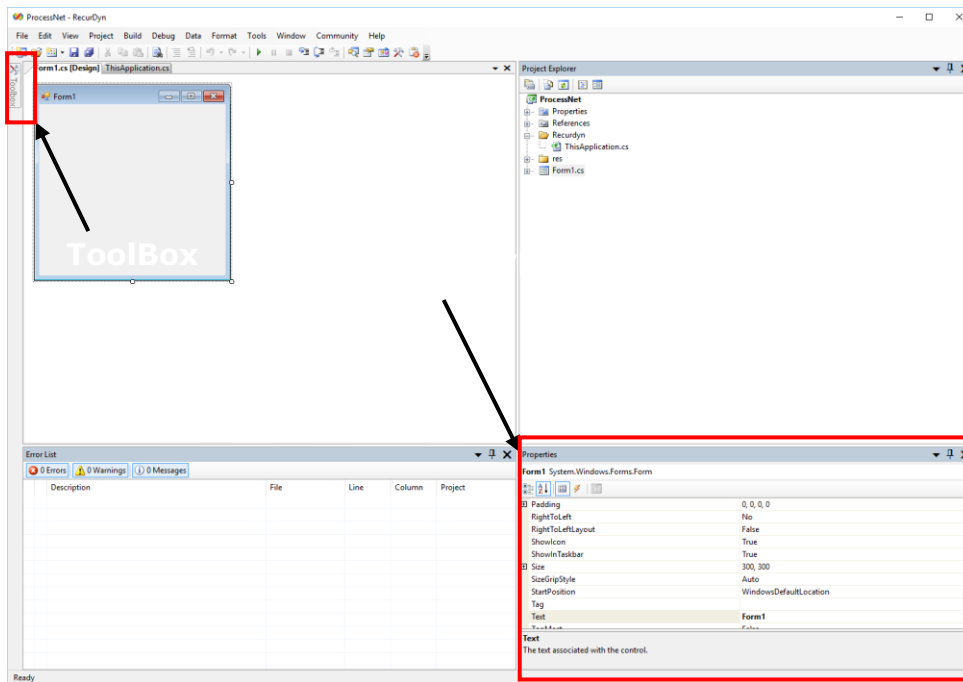
1. Return to **ProcessNet IDE**.
2. From the **Project** menu, select **Add Windows Form**.
3. In the Add New Item dialog box, select **Windows Form**, as shown below.



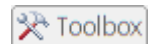
4. Accept the default name **Form1.cs**.
5. Click **Add**.



The design window for Form1, Form1.cs [Design], appears in the IDE Project Editor window.

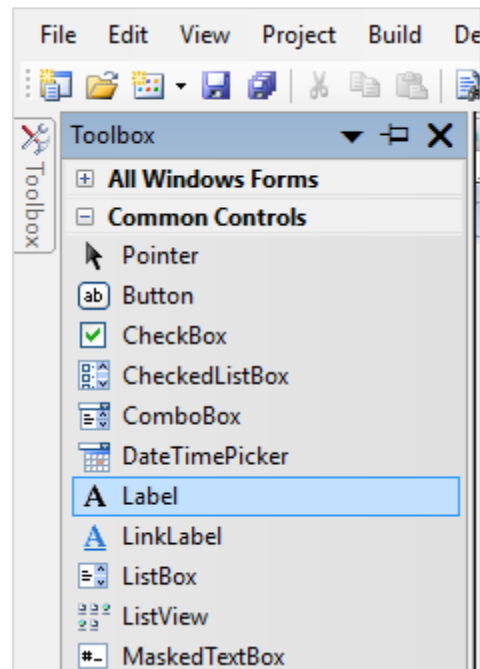


- In the upper left of the screen, move the cursor over the **Toolbox** tool.



A fly-out menu appears, which contains the different elements you could add to dialog windows and other similar controls.

- In the **Common Controls** list, click **Label**, and drag it into the upper left area of the dialog you are designing.



8. Repeat the same step as above, except drag **TextBox** into the dialog, to the right of the **Label**, as shown on the right.

9. Repeat the last two steps, twice, so there are a total of three rows of labels followed by textboxes.

10. From the **Toolbox**, add a **Checkbox** below the other elements.

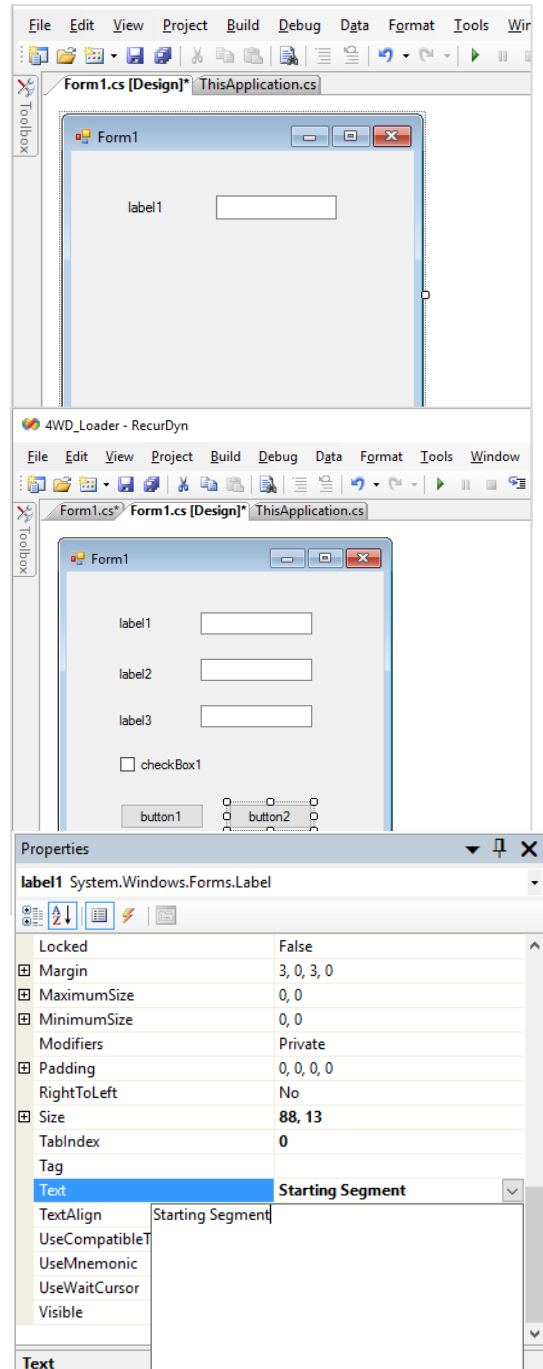
11. From the **Toolbox**, add two **Buttons** below the other elements.

At this point, the dialog should appear as shown on the right.

12. Click **label1** to select it.

13. In the Properties window in the lower right corner, change the value of **Text** to **Starting Segment**.

**Tip:** The edit field for the label is small, but you can edit the name more easily by clicking the dropdown arrow to the right of the field, as shown on the right. You have a larger area to work with.

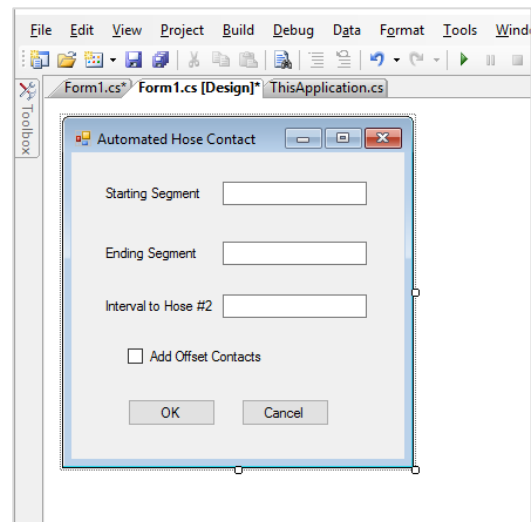


14. Repeat the last step for **Labels 2 and 3**, the **Checkbox, button1** and **button2**, and the **Dialog**, using the following table for the name changes:

Dialog Element	Text
label1	Starting Segment
label2	Ending Segment
label3	Interval to Hose #2
checkBox1	Add Offset Contacts
button1	OK
button2	Cancel
Form1	Automated Hose Contact

**Tip:** Select the dialog itself by clicking anywhere where there are no components.

15. Resize and move the dialog elements so the dialog appears as shown at below.



**Tip:** The **ProcessNet IDE** provides alignment guides to aid in this step.

16. From the **File** menu, select **Save All**. This will save **Form1.cs** as well as the project setup.

## Defining the Behavior of the Dialog

At this point, the appearance of the dialog has been set. You now have to define its behavior, by adding variables that will save the values users input into the textboxes, as well as the state of the checkbox. You will add code that will initialize all these variables and will also define what happens when the user clicks the **OK** button.

### To define the dialog window behavior:

1. In the dialog design window, double-click any part of the dialog that is not another component.

This displays the code of **Form1.cs** in the IDE Project editor window and creates stubs for a subroutine called `Form1_Load`. This subroutine will be called whenever a new copy of the dialog box is created and loaded, so this is an ideal place to put code that will initialize variables.

2. Insert the following block of code, which defines the variables that are used in the dialog box:

```
public partial class Form1 : Form
{
    public int BNumStart;
    public int BNumEnd;
    public int BodyInterval;
    public bool AddOffsetFlag;

    public Form1()
    {
        InitializeComponent();
    }
}
```

3. Insert the following block of code, which sets up the initial values that appear in the dialog box:

```
private void Form1_Load(object sender, EventArgs e)
{
    textBox1.Text = "20";
    BNumStart = 20;
    textBox2.Text = "30";
    BNumEnd = 30;
    textBox3.Text = "51";
    BodyInterval = 51;
    checkBox1.Checked = false;
}
```

4. Return to the dialog design window and double-click on the **OK** button.
5. Within the new method that was just automatically created, insert the following code as shown below:

```
private void button1_Click(object sender, EventArgs e)
{
    BNumStart = Convert.ToInt32(textBox1.Text);
    BNumEnd = Convert.ToInt32(textBox2.Text);
    BodyInterval = Convert.ToInt32(textBox3.Text);
    AddOffsetFlag = checkBox1.Checked;
    DialogResult = DialogResult.OK;
    Close();
}
```

6. Return to the dialog design window again and this time double-click on the Cancel button.
7. Within the new method that was just automatically created, insert the following code as shown below:

```
private void button2_Click(object sender, EventArgs e)
{
    Close();
}
```

8. From the File menu, select Save Form1.cs.

## Displaying the Dialog when Running the Macro

Now you will define when the dialog is displayed when running the macro, by creating a new instance of it within the macro subroutine.

### To display the dialog window from within the macro:

1. Copy the entire **ProcessNetTutorialCreateSolidContact** subroutine that you just created.
2. Paste it into ThisApplication.cs, renaming the sub ProcessNetTutorialCreateSolidContact\_WithDialog.

3. Make the following changes to the subroutine, as indicated below (delete the code indicated with ~~strikethroughs~~). An explanation of each change will follow.

```
public void ProcessNetTutorialCreateSolidContact_WithDialog()
{
    int BodyNumStart = 20; // Start creating contacts with body 20
    int BodyNumEnd = 30; // Continue until body 30
    int BodyInterval = 51; // Interval between body number on hose 1
        // and corresponding body's number on
        // hose 2 is 51

    // Create a Form
    Form1 MyForm = new Form1();

    // Open the Dialog
    MyForm.ShowDialog();

    if (MyForm.DialogResult == System.Windows.Forms.DialogResult.OK)
    {
        int NumContacts = 0;
        int BodyNumStart = MyForm.BNumStart;
        int BodyNumEnd = MyForm.BNumEnd;
        int BodyInterval = MyForm.BodyInterval;

        for (int i = BodyNumStart; i <= BodyNumEnd; i++)
        {
            int j = i + BodyInterval; // j is the index for the
            // corresponding bodies on
            // hose #2
            // Do the contact for corresponding bodies

            solidContact.ContactProperty.DampingCoefficient.Value = 0.1;

            // Increment the number of contacts
            NumContacts = NumContacts + 1;

            if (MyForm.AddOffsetFlag)
            {
                //Do the contact for body i+1 and body j
                //Do the contact for body i and body j+1

                solidContact.ContactProperty.DampingCoefficient.Value = 0.1;
                // Increment the number of contacts
                NumContacts = NumContacts + 2;
            }
        }
        application.PrintMessage(NumContacts.ToString() +
        " contacts were created between the two hoses.");
    }
}
```

- Change 1:

```
int BodyNumStart = 20; // Start creating contacts with body 20
int BodyNumEnd = 30; // Continue until body 30
int BodyInterval = 51; // Interval between body number on hose 1
    // and corresponding body's number on
    // hose 2 is 51
```

Here you deleted code which hard-coded the values which will now be determined by user input into the dialog.

- Change 2:

```
// Create a Form
Form1 MyForm = new Form1();

// Open the Dialog
MyForm.ShowDialog();

if (MyForm.DialogResult == System.Windows.Forms.DialogResult.OK)
{
    int NumContacts = 0;
```

Here you created a new instance of Form1 and displayed it. The if statement tests for the response of the user. If the user clicks **OK**, the code enclosed in the if statement is executed. Also, a variable called **NumContacts** is declared, which keeps track of the number of contacts that have been created. The if statement is closed in Change 5.

- Change 3:

```
int BodyNumStart = MyForm.BNumStart;
int BodyNumEnd = MyForm.BNumEnd;
int BodyInterval = MyForm.BodyInterval;
```

Here you replaced the code where the segment number variables were hard-coded. The new code assigns these variables with the values the user enters in the dialog.

- Change 4:

```
// Increment the number of contacts
NumContacts = NumContacts + 1;

if (MyForm.AddOffsetFlag)
{
    // Increment the number of contacts
    NumContacts = NumContacts + 2;
```

**NumContacts** is updated appropriately. Also, the if statement tests for whether or not the user checked the **Add Offset Contacts** checkbox. If the user did, the code enclosed by the if statement is executed.

- Change 5:

```
application.PrintMessage(NumContacts.ToString() +
    " contacts were created between the two hoses.");
}
```

An output message is displayed to the user within the RecurDyn message window, which confirms how many contacts were created. Also, the outermost if statement is closed (see Change 1).

4. From the **File** menu, select **Save ThisApplication.cs**.

## Test the Dialog Box

Repeat the earlier steps to build the macro and test it with the changes you just made.

### To repeat the processes:

1. Ensure there are no errors or warnings in the Error List window at the bottom of the IDE window. If there are, check the listing and make any corrections. From the **Build** menu, select **Build ProcessNet**.
2. Return to RecurDyn and delete all of the contacts that were defined for the previous run.
3. From the **ProcessNet(VSTA)** group in the **Customize** tab, click **Run**.

You will see that there is a new item in the list, **ProcessNetDialogCreateSolidContact\_WithDailog**. Select that item.

Its name is loaded in the field next to the **Run** button.

4. Click **Run**.

The dialog box shown on the right appears.

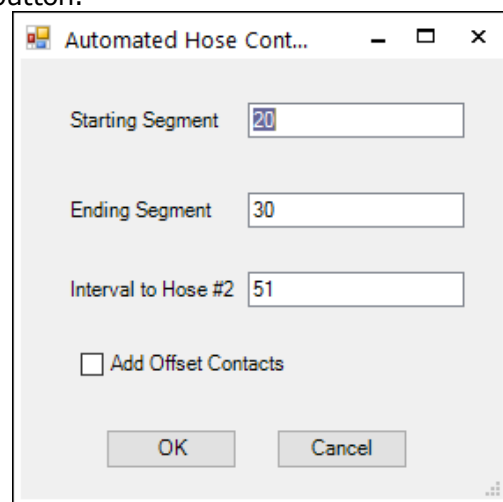
5. Click **OK** to use the default values.

You should now see in the RecurDyn Database window that 11 contacts have been created. A confirmation message in the RecurDyn message output window also states that 11 contacts were created.

6. Delete the contacts that were just created by clicking on the **Undo** arrow in RecurDyn, and run the macro again, but this time select the **Add Offset Contacts** checkbox.

You should see that 33 contacts are added to your model, and again there will be a confirmation message in the RecurDyn message output window.

7. Restore the results from the simulation that was run earlier. Click on the **File** menu and select the **Import** command. Use the pull-down menu to change the file type to **RecurDyn Animation Data File (\*.rad)**. Select the file **4WD\_Loader.rad** and click **Open**.
8. Close the **Run ProcessNet** window.





## Automating Plot Creation

In this chapter, you will make two plots in a multi-window plot:

- The first plot will display the force magnitudes of the individual segment-to-segment contacts. Contacts that do not experience any force will be excluded, so the plot contains only data of interest. The first plot will be enhanced with improved formatting, addition of labels, and scaling of the X-axis to focus on the time of hose contact.
- The second plot will display the total hose-to-hose contact force, broken into X, Y, and Z components. Both plots will be labeled and formatted.

### Task Objective

Learn how to use **ProcessNet** commands to automate plotting. This will include:

- Importing a plot data file.
- Intelligently determining which data to display.
- Processing and then plotting the data on a multi-window plot.
- Formatting the chart based on the data.



### Estimated Time to Complete

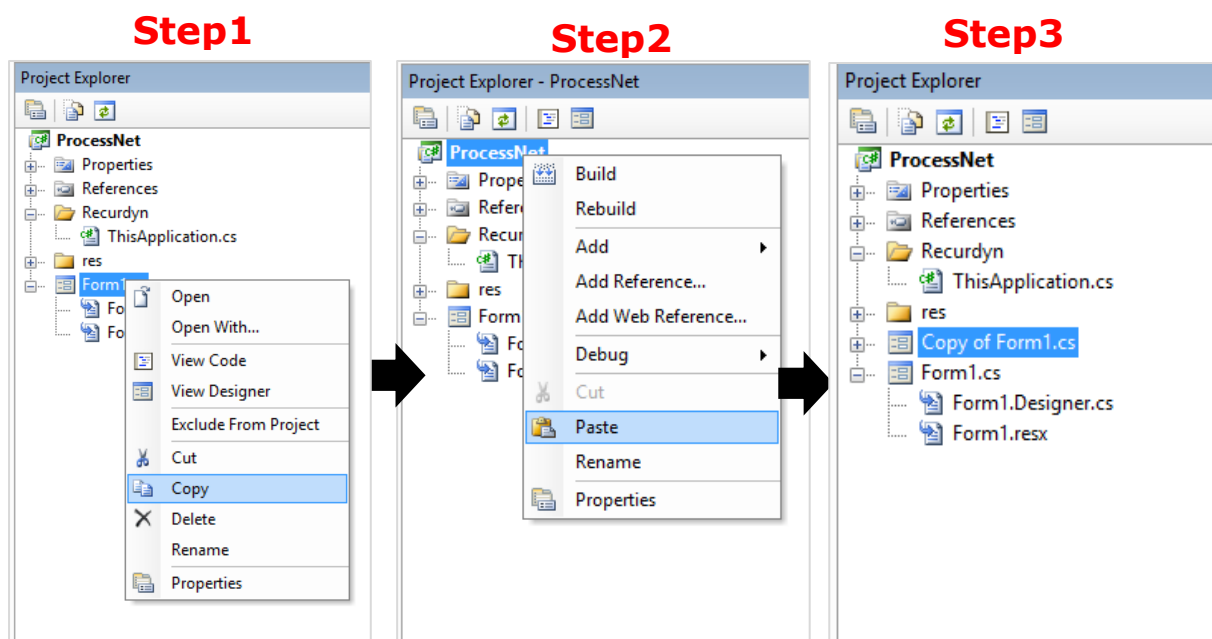
20 minutes

## Creating a Dialog

To determine which data to plot, another dialog window that reads user input will be needed. This dialog will be very similar to the first dialog you created, so start by copying that dialog.

**To create the new dialog window:**

1. In the Project Explorer window on the right, right-click **Form1.cs**, and select **Copy** (refer to the diagram below for the next several steps).
2. Right-click **ProcessNet** and select **Paste**.
3. Click **Copy of Form1.cs** to highlight it, and then click again to edit its name.
4. Change the name to **Form2.cs**.



**To adjust the new dialog window:**

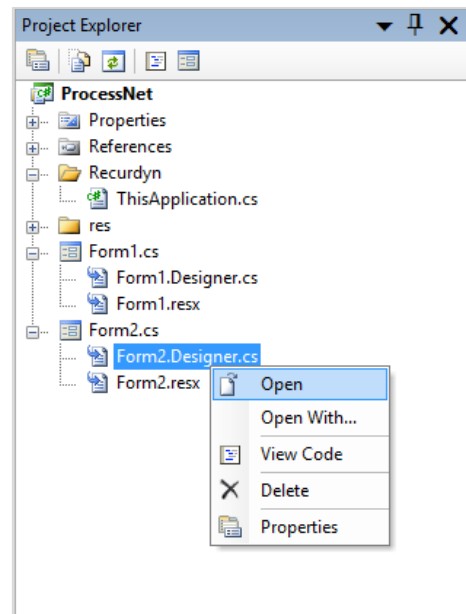
1. Double-click Form2.cs to open it.  
The dialog design window appears.
2. From the View menu, select Code to display the code window.
3. In the code, perform a search for "Form1". Replace all instances with "Form2". There should be 3 instances of this, as shown below:

```
namespace ProcessNet.csproj
{
    public partial class Form1 Form2 : Form
    {
        public int BNumStart;
        public int BNumEnd;
        public int BodyInterval;
        public bool AddOffsetFlag;

        public Form1 Form2()
        {
            InitializeComponent();
        }

        private void Form1_Load Form2_Load(object sender, EventArgs e)
        {
            textBox1.Text = "20";
            BNumStart = 20;
        }
    }
}
```

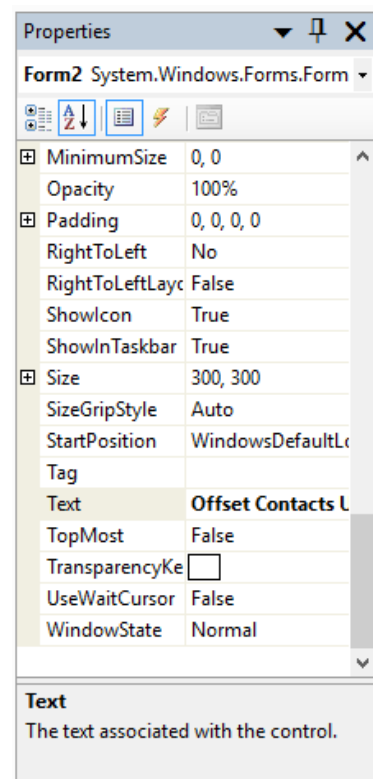
4. In the **Project Explorer** window, right-click on **Form2.Designer.cs** and select **Open**, as shown at right.



- Again, perform a search for "Form1". Replace all instances with "Form2". There should be 2 instances of this, as shown below:

```
partial class Form1 Form2
{
    private void InitializeComponent()
    {
        this.Load += new System.EventHandler(this.Form1_Load Form2_Load);
        this.ResumeLayout(false);
        this.PerformLayout();
    }
}
```

- Above the IDE Project Editor window, **click Form2.cs [Design]** tab.
- Select the dialog window itself (not any of its individual components), by clicking on any part of the dialog which is not a component.
- In the Properties window (lower-right of screen), change the **Text** to **Automated Hose Contact Plotting**.



- In the **IDE Project Editor** window, click and drag the right side of the dialog window to resize it so the new title is fully displayed.
- Click the **Add Offset Contacts** label.
- In the Properties window, change the text to **Offset Contacts Used**.
- From the **File** menu, select **Save All**.

## Plotting the Contact Forces

You will now create a new subroutine in the ThisApplication class.

### To create a new subroutine:

1. Copy the following code block (including code on the next page) and insert it into the **ThisApplication** class. (**Tip:** Make sure that this code is inserted within the class but not within another subroutine.)

```
public void ProcessNetTutorialPlotData()
{
    // Create an auto contact plotting dialog
    Form2 MyForm = new Form2();

    // Open the dialog
    MyForm.ShowDialog();

    // If the user clicked OK:
    if (MyForm.DialogResult == System.Windows.Forms.DialogResult.OK)
    {
        // Get the TIME data
        double[] TIME = plotDocument.GetPlotData("4WD_Loader/TIME");

        // ||||| Plot individual contact forces |||||

        // Active upper-left plot window
        plotDocument.ActivateView(0, 0);

        for (int bodyIndex = MyForm.BNumStart;
            bodyIndex <= MyForm.BNumEnd; bodyIndex++)
        {
            // Load up the contact name number array

            String[] contNum = {bodyIndex.ToString(), "", ""};

            if (MyForm.AddOffsetFlag)
            {
                contNum[1] = bodyIndex.ToString() + "a";
                contNum[2] = bodyIndex.ToString() + "b";
            }

            for (int contNumIndex = 0; contNumIndex < contNum.Length;
                contNumIndex++)
            {
                if (String.Compare(contNum[contNumIndex], "") != 0)
                {
                    // Get the contact data for this segment
                    double[] contact = plotDocument.GetPlotData(
                        "4WD_Loader/Contact/Solid Contact/solidContact"
                        + contNum[contNumIndex] +
                        "/FM_SolidContact");

                    // Plot vs. TIME
                    plotDocument.DrawPlot("Contact"
                        + contNum[contNumIndex], TIME, contact);
                }
            }
        }
    }
}
```

The first two commands create a new Form2 window and display it to the user:

```
// Create an auto contact plotting dialog
Form2 MyForm = new Form2();

// Open the dialog
MyForm.ShowDialog();
```

The following if statement tests whether or not the user clicked the OK button, and executes the enclosed code if the user did:

```
// If the user clicked OK:
if (MyForm.DialogResult == System.Windows.Forms.DialogResult.OK)
{
```

The following code obtains the plot data for TIME from the imported file and saves it to a new array.

```
// Get the TIME data
double[] TIME = plotDocument.GetPlotData("4WD_Loader/TIME");
```

The following command activates the upper left plotting window:

```
// Active upper-left plot window
plotDocument.ActivateView(0, 0);
```

The following code loads up an array of strings which will contain partial names of the data to plot. The array is loaded according to what the user entered into the dialog. For example, if the user set the Starting Contact to 20 and selected the "Offset Contacts Used" checkbox, then during the first loop the array will contain the strings {"20", "20a", "20b"}.

```
for (int bodyIndex = MyForm.BNumStart;
     bodyIndex <= MyForm.BNumEnd; bodyIndex++)
{
    // Load up the contact name number array

    String[] contNum = {bodyIndex.ToString(), "", ""};

    if (MyForm.AddOffsetFlag)
    {
        contNum[1] = bodyIndex.ToString() + "a";
        contNum[2] = bodyIndex.ToString() + "b";
    }

    for (int contNumIndex = 0; contNumIndex < contNum.Length; contNumIndex++)
    {
        if (String.Compare(contNum[contNumIndex], "") != 0)
        {
```

This next command retrieves plot data by the name of the data. The name is the same as that seen from the plotting database window, using the "/" character as a separator:

```
// Get the contact data for this segment
double[] contact = plotDocument.GetPlotData(
    "4WD_Loader/Contact/Solid Contact/solidContact"
    + contNum[contNumIndex] + "/FM_SolidContact");
```

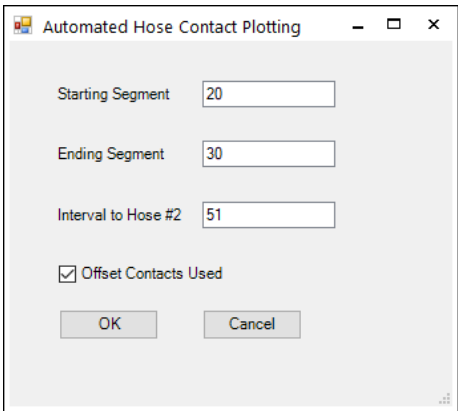
Finally, the next command draws the data in the plot window.

```
// Plot vs. TIME
plotDocument.DrawPlot("Contact"
+ contNum[contNumIndex], TIME, contact);
```

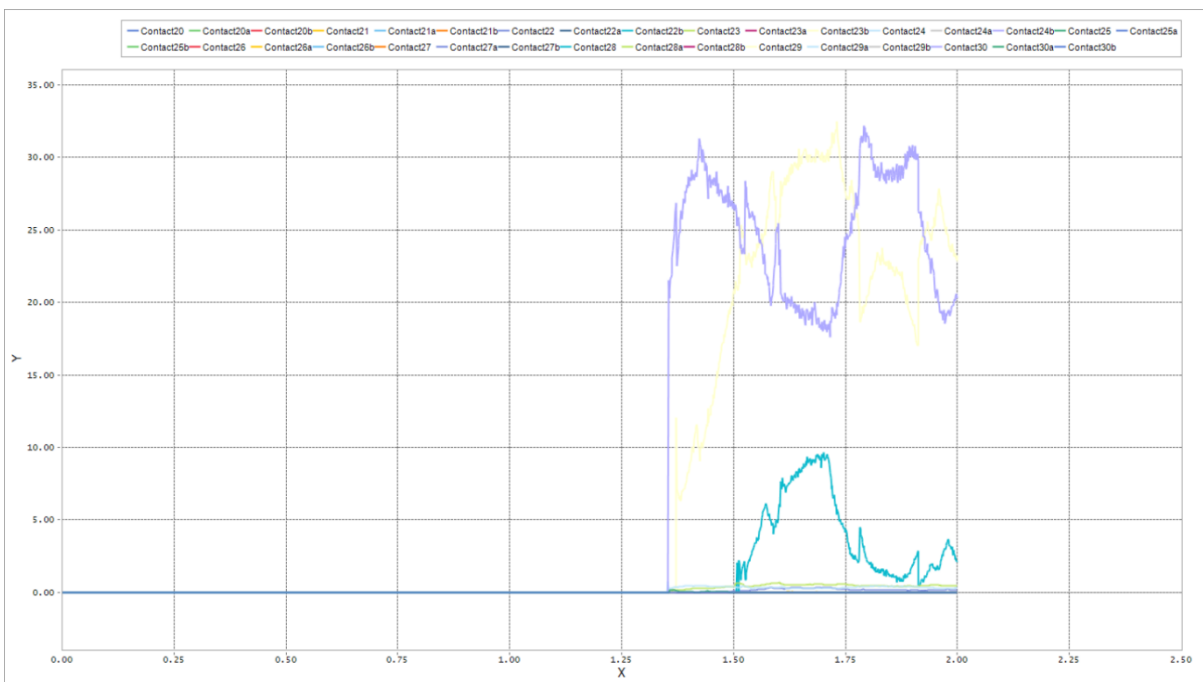
2. Save the file.
3. From the **Build** menu, select **Build ProcessNet**.

**To test the edited subroutine with plotting:**

1. Return to the RecurDyn modeling window with the current loader model loaded and open a **plotting window**.
2. From the **ProcessNet(VSTA)** group in the **Customize** tab, click **Run**.
3. From the list, select **ProcessNetTutorialPlotData**.
4. Select **Run**.
5. In the Dialog2 dialog window that appears, accept the default segment numbers, and check **Offset Contacts Used**.



You should see the following plot (as adjusted to the size of your RecurDyn Plot Window):



Above, all 33 contacts have been plotted, but only a fraction of the curves contain non-zero values because the contact between the hoses is restricted to several segments on each hose. Also, all the contact occurs at the last third of the simulation, but the plot shows the entire timeline of the simulation.

## Improving the Contact Force Plot

The plot can be improved so it focuses on the interesting behavior, by:

- Only plotting non-zero contacts.
- Reducing the timeline (or range of the X-axis) so it only contains non-zero contact behavior.

You will start by only plotting non-zero contacts. Some of the logic for reducing the timeline will be introduced but formatting the X-axis will come later.

### To plot only non-zero contact forces:

1. Make the following changes to the code:

```
public void ProcessNetTutorialPlotData()
{
    // Create an auto contact plotting dialog
    Form2 MyForm = new Form2();

    // Open the dialog
    MyForm.ShowDialog();

    // If the user clicked OK:
    if (MyForm.DialogResult == System.Windows.Forms.DialogResult.OK)
    {
        // Get the TIME data
        double[] TIME = plotDocument.GetPlotData("4WD_Loader/TIME");

        // Initialize variables for X-axis limits
        double timeAtFirstContact = TIME[TIME.Length - 1];
        double timeAtLastContact = 0;

        // ||||| Plot individual contact forces |||||

        // Active upper-left plot window
        plotDocument.ActivateView(0, 0);

        for (int bodyIndex = MyForm.BNumStart;
            bodyIndex <= MyForm.BNumEnd; bodyIndex++)
        {
            // Load up the contact name number array
            String[] contNum = {bodyIndex.ToString(), "", ""};

            if (MyForm.AddOffsetFlag)
            {
                contNum[1] = bodyIndex.ToString() + "a";
                contNum[2] = bodyIndex.ToString() + "b";
            }

            for (int contNumIndex = 0; contNumIndex < contNum.Length;
                contNumIndex++)
```

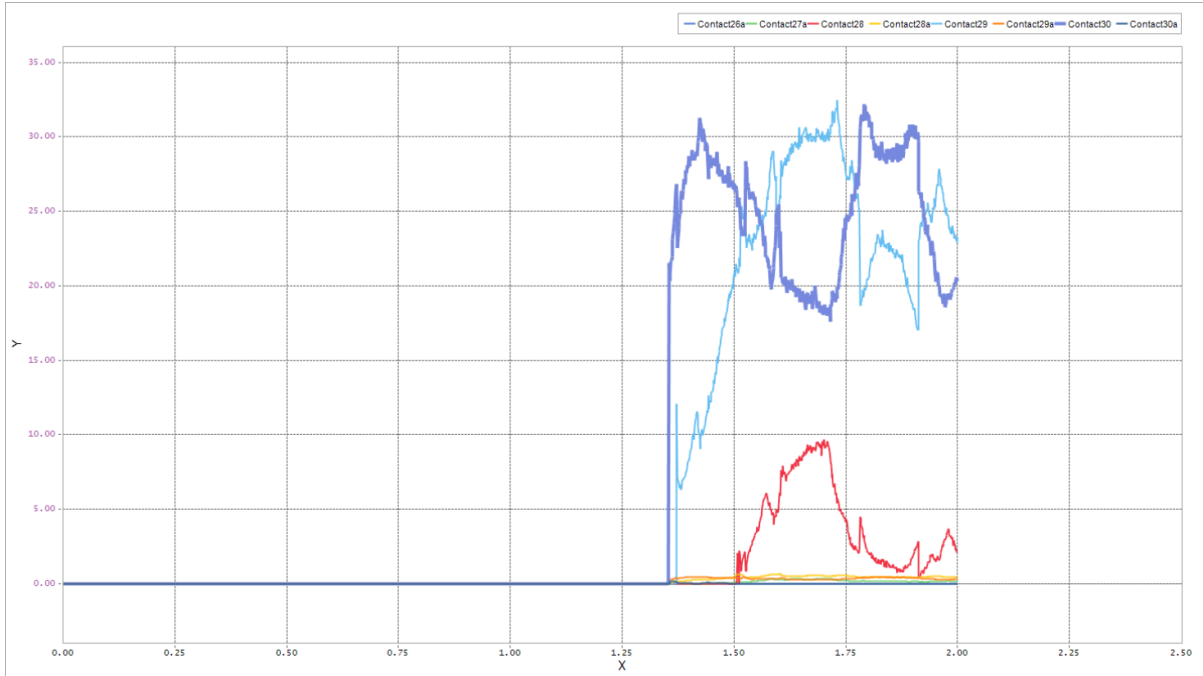




**To test the non-zero contact force plot:**

Follow the same steps as before to test the new **ProcessNet** macro in a new Plot Window (open a new Plot Window to begin).

You should see the following plot:



You can now see which contacts are non-zero throughout the simulation. Only 8 of the 33 contacts are non-zero.

## Improving the Plot Formatting

The last plot completed communicates more clearly than the previous versions, but there is still room for improvement. The plot title and axis labels and titles are not descriptive, and the timescale is still not focused on the interesting part of the simulation. You will now perform some plot formatting using **ProcessNet** functions.

### To improve the plot formatting:

1. Make the following code changes:

```
#region namespace
using System;
using Microsoft.VisualBasic;
using System.Windows.Forms; //IWin32Window
using System.IO;

using FunctionBay.RecurDyn.ProcessNet;
//For C#
using FunctionBay.RecurDyn.ProcessNet.Chart;
//using FunctionBay.RecurDyn.ProcessNet.MTT2D;
//using FunctionBay.RecurDyn.ProcessNet.FFlex;
//using FunctionBay.RecurDyn.ProcessNet.RFlex;
//using FunctionBay.RecurDyn.ProcessNet.Tire;
```

```

        // If non-zero contact data found, then plot vs.
        // TIME
        if (madeContact) plotDocument.DrawPlot("Contact"
+ contNum[contNumIndex], TIME, contact);
    }
}

// Get Chart object and format

IChart Chart1 = plotDocument.ActiveChartControl;
Chart1.Title.Text = "Hose Contact Force";
Chart1.AxisX.Title.Text = "Time (sec)";
Chart1.AxisX.Min = 0.1 * Math.Truncate(timeAtFirstContact * 10);
Chart1.AxisX.Max = 0.1 * Math.Ceiling(timeAtLastContact * 10);
Chart1.AxisX.LabelsFormat.Decimals = 1;
Chart1.AxisY.LabelsFormat.Decimals = 1;
Chart1.AxisY.Title.Text = "Contact Force (N)";
Chart1.LegendBox.Alignment = LegendBoxAlignment.LegendBoxAlignment_Far;
}
}
```

Note that the definition of the X-axis limits **Chart1.AxisX.Min** and **Chart1.AxisX.Max** use the variables **timeAtFirstContact** and **timeAtLastContact** that were calculated in the earlier section.

2. Save the file.
3. From the **Build** menu, select **Build ProcessNet**.

### To test the formatting of the plot:

Follow the same steps as before to test the new **ProcessNet** macro in a new **Plot Window** (open a new **Plot Window** to begin).

You should see the following plot:



Now the plot has been fully formatted, and the X-axis has been scaled so the non-zero contact can be seen in detail.

## Plotting the Total X, Y, and Z Contact Force

Suppose that you now want to see what the total contact force between the two hoses is, divided into its X, Y, and Z components. The data for this plot will need to be calculated using the existing data in the plot file. In **ProcessNet**, because the plot data can be stored in arrays, you can perform mathematical operations on it. For this next plot, you will add array data together to create the sums of the X, Y, and Z components of all the contacts.

Because you have already learned from the previous plot most of the commands to create the new one, a single large block of code will be copied and inserted for this part of the tutorial. Note, however, the additional data processing that sums the arrays together.

### To add the second plot:

1. Insert the following code block into the location shown below, near the end of the subroutine but within the If statement:

```
// Get Chart object and format
.
.
.
Chart1.AxisY.Title.Text = "Contact Force (N)";
Chart1.LegendBox.DockedPosition = DockedPositionType.DockedPositionType_Right;

// ||||| Plot sums of X, Y, and Z components |||||

// Activate the lower-left plot window
plotDocument.ActivateView(1, 0);

double[] xSum = new double[TIME.Length];
double[] ySum = new double[TIME.Length];
double[] zSum = new double[TIME.Length];

for (int bodyIndex = MyForm.BNumStart; bodyIndex <= MyForm.BNumEnd; bodyIndex++)
{
    // Load up the contact name number array

    String[] contNum = {bodyIndex.ToString(), "", ""};
    if (MyForm.AddOffsetFlag)
    {
        contNum[1] = bodyIndex.ToString() + "a";
        contNum[2] = bodyIndex.ToString() + "b";
    }
    for (int contNumIndex = 0; contNumIndex < contNum.Length; contNumIndex++)
    {
        if (String.Compare(contNum[contNumIndex], "") != 0)
        {
            // Get the contact data for this segment
            double[] contactX = plotDocument.GetPlotData(
                "4WD_Loader/Contact/Solid Contact/solidContact"
                + contNum[contNumIndex] + "/FX_SolidContact");
            double[] contactY = plotDocument.GetPlotData(
                "4WD_Loader/Contact/Solid Contact/solidContact"
                + contNum[contNumIndex] + "/FY_SolidContact");
            double[] contactZ = plotDocument.GetPlotData(
                "4WD_Loader/Contact/Solid Contact/solidContact"
                + contNum[contNumIndex] + "/FZ_SolidContact");

            if (contNumIndex == MyForm.BNumStart)
            {
                // If looping through the first contact,
                // initialize the sum arrays
                xSum = contactX;
                ySum = contactY;
                zSum = contactZ;
            }
        }
    }
}
```

```

        else
        {
            // Else, add this contact's array data to the
            // running total
            for (int j = 0; j < TIME.Length; j++)
            {
                xSum[j] = xSum[j] + contactX[j];
                ySum[j] = ySum[j] + contactY[j];
                zSum[j] = zSum[j] + contactZ[j];
            }
        }
    }
}

// Plot vs. TIME
plotDocument.DrawPlot("X Sum", TIME, xSum);
plotDocument.DrawPlot("Y Sum", TIME, ySum);
plotDocument.DrawPlot("Z Sum", TIME, zSum);

// Get Chart object and format
IChart Chart2 = plotDocument.ActiveChartControl;
Chart2.Title.Text = "Total Hose-to-Hose Contact Force";
Chart2.AxisX.Title.Text = "Time (sec)";
Chart2.AxisX.Min = 0.1 * Math.Truncate(timeAtFirstContact * 10);
Chart2.AxisX.Max = 0.1 * Math.Ceiling(timeAtLastContact * 10);
Chart2.AxisX.LabelsFormat.Decimals = 1;
Chart2.AxisY.LabelsFormat.Decimals = 1;
Chart2.AxisY.Title.Text = "Contact Force (N)";
Chart2.LegendBox.Alignment = LegendBoxAlignment.LegendBoxAlignment_Far;

}
}

```

2. Save the file.
3. From the **Build** menu, select **Build ProcessNet**.

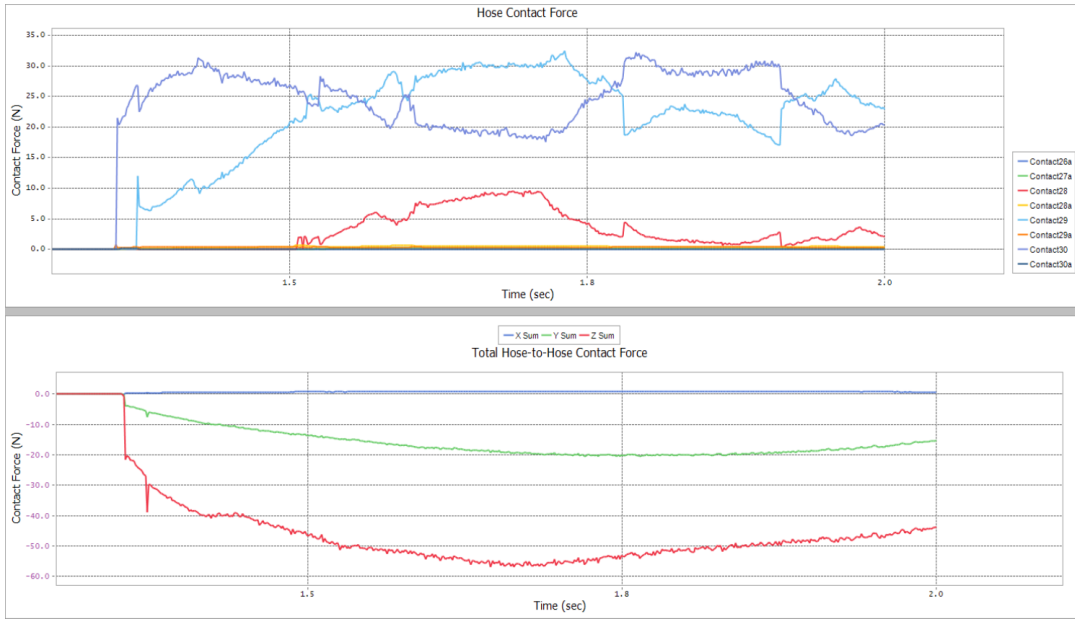
### To test the second plot:

The second plot will be displayed in the lower-left window (in contrast, the first plot was displayed in the upper right window), so ensure that both windows are displayed to avoid errors.

Follow the same steps as before to test the new **ProcessNet** macro in a new **Plot Window** except this time, from the Windows group of the Home tab, select Show Left Windows.

- You should see the following two plots:

# PROCESSNET 4WD LOADER TUTORIAL (VSTA)



*Thank you for participating in this tutorial!*