IPC Quick Start Guide

DOI:
10.3927/59340663

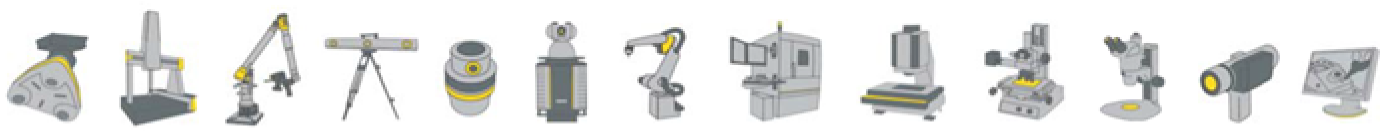
X-Tek X-ray and CT Inspection

IPC Quick Start Guide

IPC programming for Inspect-X

XTM0499-A1

Parmesh Gajjar, Henry Moseley X-ray Imaging Facility,
The University of Manchester



Legal information

About this manual

This manual applies to the software type and version given on the front cover. It contains the ORIGINAL Instructions for the safe use of this software. Nikon Metrology reserves the right to revise and improve its products as it sees fit. This publication describes the state of this product at the time of publication, and may not reflect the product at all times in the future.

Liability for calibration

Nikon Metrology will only accept liability for re-calibration if carried out in our own factory, and for the verification process at the customer's site carried out by Nikon Metrology staff. We also accept liability if the re-calibration or verification process is carried out by the customer if these processes are explicitly agreed between the customer and Nikon Metrology. We will not accept liability if the customer performs the re-calibration, or takes over responsibility for the verification process from Nikon Metrology without prior authorisation, even if the recommended Nikon Metrology process is followed.

Author information

Parmesh Gajjar,
The Henry Moseley X-ray Imaging Facility, The University of Manchester, Oxford Road, Manchester M13 9PL. Email: ***parmesh.gajjar@alumni.manchester.ac.uk***.

Manufacturer's contact details

Nikon Metrology Tring (X-Tek Systems Ltd),
Tring Business Centre, Icknield Way, Tring, Hertfordshire HP23 4JX. United Kingdom.

Tel. +44 1442 828700 Fax: +44 1442 828118

Website: http://www.nikonmetrology.com

Registered in England No. 01981536, VAT No. GB433 079 460

Sources of additional information

Contact ***service.nm-tring@nikon.com*** to:

- Get technical support (customers and distributors)
- Request quotations and order service parts

Current application software, drivers and manuals can be downloaded from:

http://extranet.nikonmetrology.com/

Access is restricted and a user account is required. Send an application for an account to ***service.nm-tring@nikon.com***.

Documentation feedback, suggestions for new or missing content or other comments can be sent to: ***EngDoc.NM-Derby@nikon.com***.

Contents

1 Introduction

1.1 About this guide

The Nikon Inspect-X software used for controlling Nikon X-ray Computed Tomography (CT) systems allows users to perform a wide variety of tasks, including inspections, 2D and 3D CT scans and batch scheduling. Whilst the Inspect-X software fulfils the requirements of the majority of users, there may, however, be cases where users want to use their X-ray CT system to do something that cannot (easily) be done with Inspect-X. For example, users may wish to sequentially acquire and manipulate radiographs, perform CT scans at certain predetermined times or allow communication between the X-ray CT system and an external module or experimental rig.

One of the lesser known features within the Nikon Inspect-X software is that it allows users to control and manipulate their Nikon X-ray CT system from a custom-built application through Inter Process Communication (IPC). Inspect-X exposes a number of interfaces which we can access by programming using the Microsoft .NET platform. Basic operations can be accessed, for example switching X-rays on, moving the manipulator and acquiring images from the detector, as well as higher level operations such as performing a CT scan and allowing users to develop highly complex and customised applications.

The IPC interface replaces the Visual Basic for Applications (VBA) programming that was available in older versions of Inspect-X (Versions 2.2 and older). All versions of Inspect-X from version 3 onwards use the IPC interface. The new IPC interface provides a much greater potential for creating custom applications for both industrial and academic environments; however, initially getting started with IPC can be confusing and overwhelming.

The aim of this Quick Start Guide is to help you set up and start programming using IPC by writing simple applications:

- A walk-through tutorial is provided in ***Tutorial 1: Writing your first IPC program - Switching X-rays on and off*** (on page 5) for a simple program to switch X-rays on and off. This also explains some basics of the IPC communications.
- Two further tutorial tasks can be found in ***Tutorial 2: Moving the manipulator*** (on page 27) and ***Tutorial 3: Acquiring an image*** (on page 31), which will help bring familiarity with controlling the manipulator and image-processing channels.
- Sample code is provided in ***Appendix: Sample code*** (on page 35).

This Quick Start Guide is not exhaustive, and there is a lot which is not covered. For instance, we might want to develop complex applications with complex user interfaces, created through the Designer. There are many online tutorials and books for this, and the Quick Start Guide only uses basic features of form design. The Quick Start Guide concentrates more on explaining the basics of IPC communication with Inspect-X, and it is hoped that these sections will provide sufficient experience of the IPC interface to allow you to develop your own custom applications.

Note: This Quick Start Guide was written using Inspect-X version 5.1.4 and Visual Studio 2012 Professional. Other versions may look slightly different.

1.2 Prerequisites

1. A working Nikon X-ray CT system with Inspect-X software (version 4 or later) installed.
2. Microsoft Visual Studio 2010 (or later) with Microsoft Visual C# compilers.
3. Basic previous programming experience.

1.3 Getting started with C#

If you have some prior experience in programming, then you should be able to successfully follow the tutorials in this guide to build some simple IPC applications in C# that perform basic operations with the X-ray CT system. If you wish to learn more about C#, then an online tutorial can be found at ***https://www.tutorialspoint.com/csharp/index.htm***.

1.4 Acknowledging this work

I hope this Guide helps you get to grips with the basics of IPC programming, helping you create custom research and industrial applications. If you can, I would be grateful if you can cite this work as:

Gajjar, P. (2017), 'IPC Quick Start Guide', Technical report produced by The University of Manchester and Nikon Metrology; Contact address: Nikon Metrology, Tring, HP23 4JX, UK.
DOI: 10.3927/59340663

1.5 Disclaimer

The use of the IPC Programming Interface is taken solely at the discretion of you and your company together with your Nikon point of contact.

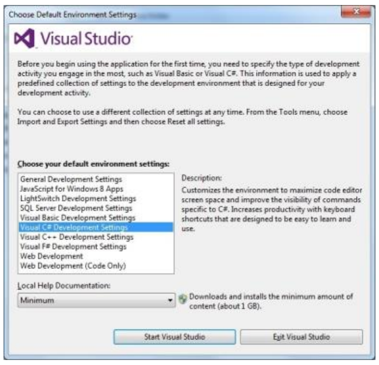
The author of this Quick Start Guide takes no responsibility for any business losses, including without limitation loss of or damage to profits, income, revenue, use, production, anticipated savings, business, contracts, commercial opportunities or goodwill that arises from using IPC Programming or following the material in this Guide.

1.6 Important information

It is illegal (against the Ionising Radiations Regulations, 1999) to write code which automatically switches X-rays on without any user interaction. Your application must include an 'OK/Cancel' button to control the generation of X-rays. This means that if the X-rays switch off because an interlock has been broken, they cannot be restarted automatically by the IPC code. If an interlock has not been broken, then a user may switch the X-rays back on again using an 'OK' button.

2 Setting Up

Microsoft Visual Studio should be installed on the acquisition computer. The .NET programming in this Quick Start Guide will be done in Visual C#, so ensure that these options are selected on installation and first start up.



Choose Default Environment Settings

Visual Studio

Before you begin using the application for the first time, you need to specify the type of development activity you engage in the most, such as Visual Basic or Visual C#. This information is used to apply a predefined collection of settings to the development environment that is designed for your development activity.

You can choose to use a different collection of settings at any time. From the Tools menu, choose Import and Export Settings and then choose Reset all settings.

Choose your default environment settings:

- General Development Settings
- JavaScript for Windows 8 Apps
- LightSwitch Development Settings
- SQL Server Development Settings
- Visual Basic Development Settings
- Visual C# Development Settings
- Visual C++ Development Settings
- Visual F# Development Settings
- Web Development
- Web Development (Code Only)

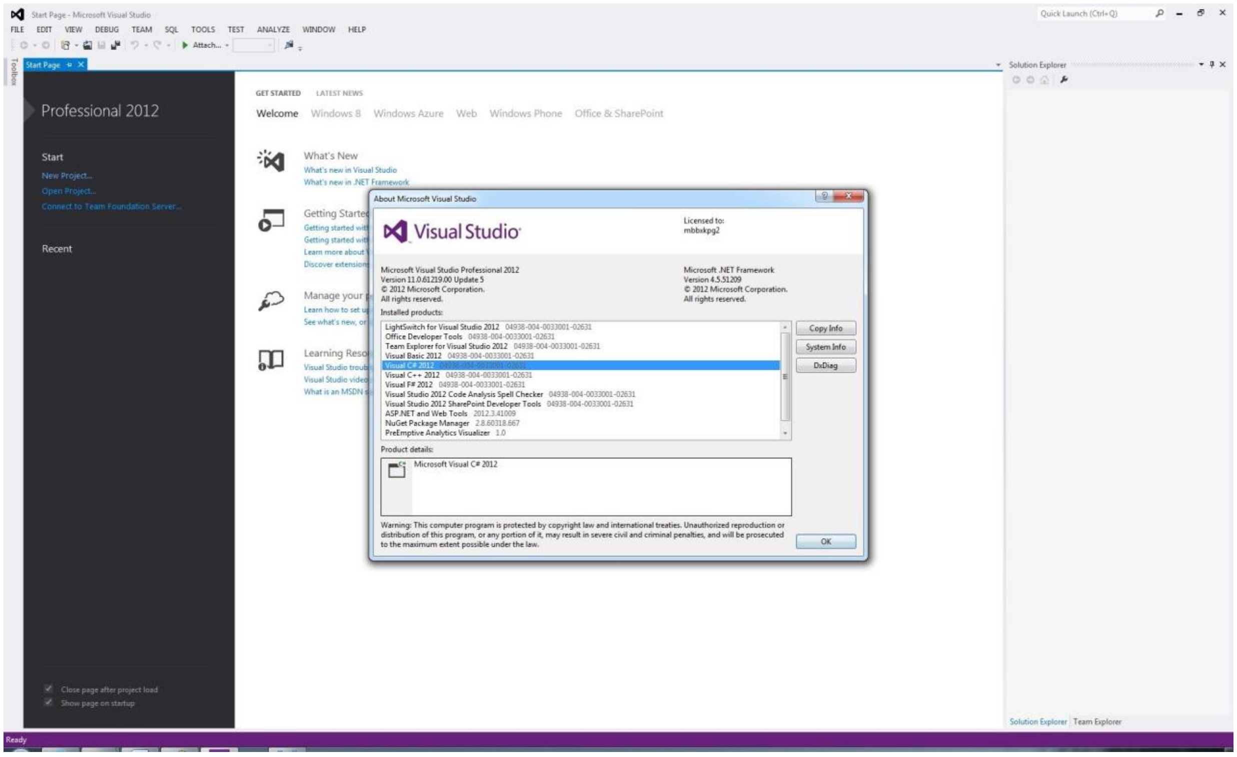
Description:
Customizes the environment to maximize code editor screen space and improve the visibility of commands specific to C#. Increases productivity with keyboard shortcuts that are designed to be easy to learn and use.

Local Help Documentation: Minimum

Downloads and installs the minimum amount of content (about 1 GB).

Figure 2-1: Visual Studio start-up screen

It is possible to check whether Visual C# is installed by navigating to **Help** > **About Microsoft Visual Studio** and checking the list of installed products.



About Microsoft Visual Studio

Visual Studio

Licensed to:
mbbxkpg2

Microsoft Visual Studio Professional 2012
Version 11.0.61219.00 Update 5
© 2012 Microsoft Corporation.
All rights reserved.

Microsoft .NET Framework
Version 4.5.51209
© 2012 Microsoft Corporation.
All rights reserved.

Installed products:

```
LightSwitch for Visual Studio 2012   04938-004-0033001-02631
Office Developer Tools   04938-004-0033001-02631
Team Explorer for Visual Studio 2012   04938-004-0033001-02631
Visual Basic 2012   04938-004-0033001-02631
Visual C# 2012   04938-004-0033001-02631
Visual C++ 2012   04938-004-0033001-02631
Visual F# 2012   04938-004-0033001-02631
Visual Studio 2012 Code Analysis Spell Checker   04938-004-0033001-02631
Visual Studio 2012 SharePoint Developer Tools   04938-004-0033001-02631
ASP.NET and Web Tools   2012.3.41009
NuGet Package Manager   2.8.60318.667
PreEmptive Analytics Visualizer   1.0
```

Product details:

Microsoft Visual C# 2012

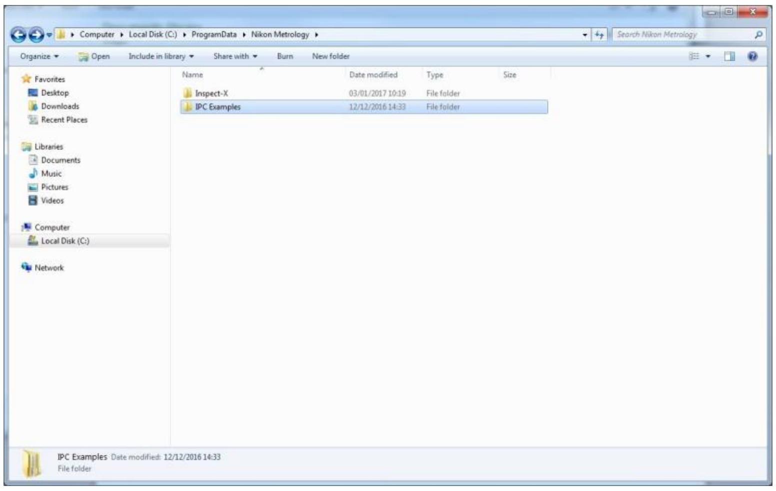
Warning: This computer program is protected by copyright law and international treaties. Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under the law.

Figure 2-2: Checking whether Visual C# is installed in Visual Studio

The next step is to copy the sample IPC programs to our documents directory (or another folder of our choice). This leaves the originals intact, in case the client-contract interfaces are erroneously damaged when programming.

- Navigate to **<OS Drive>\ProgramData\Nikon Metrology**

Note: It may be necessary to show hidden files and folders to locate the ProgramData folder.



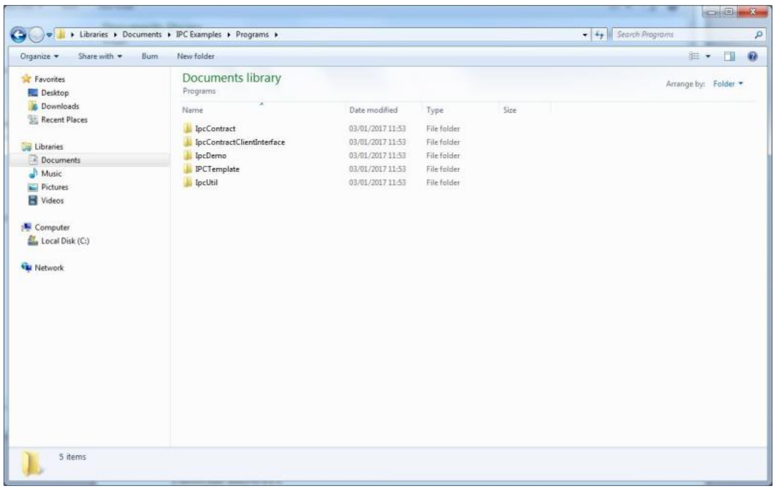
Name	Date modified	Type	Size
Inspect-X	03/01/2017 10:19	File folder	
IPC Examples	12/12/2016 14:33	File folder	

IPC Examples Date modified: 12/12/2016 14:33
File folder

Figure 2-3: Locating the IPC Examples folder

- Copy the entire “IPC Examples” folder to a destination folder of your choice (for example My Documents, namely Libraries\Documents).
- Navigate to the new “IPC Examples” just created (C:\User\User1\Documents\IPC Examples), and open the “Programs” folder.

  This should contain five folders named “IpcContract”, “IpcContractClientInterface”, “IpcDemo”, “IPCTemplate” and “IpcUtil”.



Name	Date modified	Type	Size
IpcContract	03/01/2017 11:53	File folder	
IpcContractClientInterface	03/01/2017 11:53	File folder	
IpcDemo	03/01/2017 11:53	File folder	
IPCTemplate	03/01/2017 11:53	File folder	
IpcUtil	03/01/2017 11:53	File folder	

5 items

Figure 2-4: The contents of the IPC Examples folder

The first three folders contain code that governs the communication between Inspect-X and the custom application.

The “IpcDemo” folder contains a sample working IPC application that performs a simple circular CT scan. Whilst this sample application may initially appear quite daunting, it should hopefully be more understandable after working through this Quick Start Guide.

Finally, the “IPCTemplate” folder contains a template for developing new applications. We will base our new applications on this template by copying the template folder for each new application.

3 Tutorial 1: Writing your first IPC program - Switching X-rays on and off

This tutorial is the IPC equivalent of a “Hello World” program in basic programming courses, which will give familiarity with different function calls in IPC, and yields a program that switches X-rays on, waits for five seconds after X-rays have stabilised, then switches X-rays off again.

Creating and setting up the Visual Studio project (on page 5) and ***The front-end user form and back-end code*** (on page 10) set up the Visual Studio project. The renaming allows each new application to contain appropriate titles, and although it may take a short amount of time in the first few instances, experience will allow you to rename everything very quickly.

An overview of the code (on page 14) provides an overview of the code and explains how the IPC interfaces work.

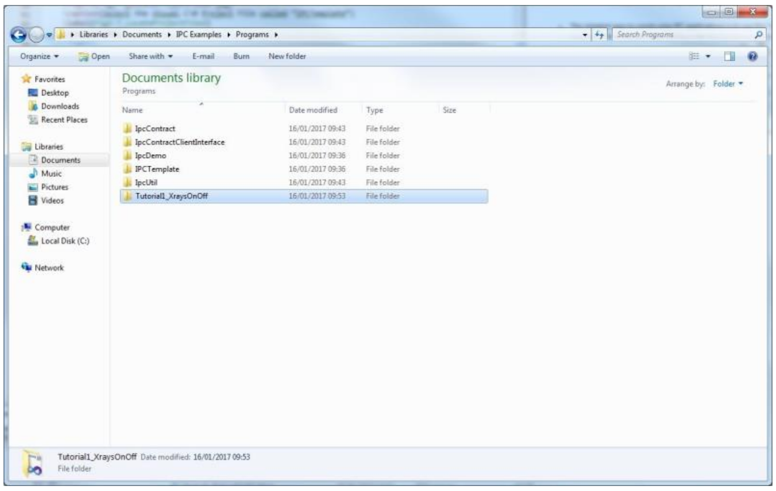
Channel connections (on page 17) through to ***Running the application*** (on page 24) constructs our first application in small segments.

If you have problems, refer to ***Troubleshooting*** (on page 24).

3.1 Creating and setting up the Visual Studio project

The simplest way to create new IPC applications is to use the “IPCTemplate” as a starting point, since a large amount of the complex code dealing with the communication with Inspect-X is already included.

- Create a copy of the “IPCTemplate” folder and rename it to “Tutorial1_XraysOnOff”.



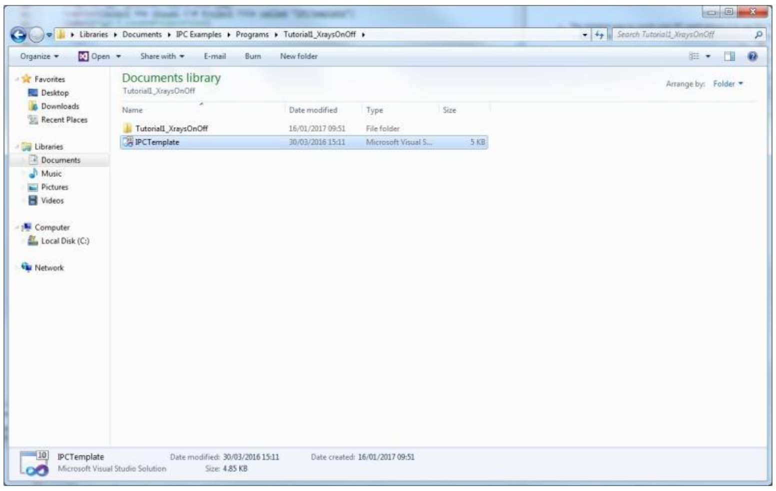
Name	Date modified	Type	Size
IpcContract	16/01/2017 09:43	File folder	
IpcContractClientInterface	16/01/2017 09:43	File folder	
IpcDemo	16/01/2017 09:36	File folder	
IPCTemplate	16/01/2017 09:36	File folder	
IpcUtil	16/01/2017 09:43	File folder	
Tutorial1_XraysOnOff	16/01/2017 09:53	File folder	

Tutorial1_XraysOnOff Date modified: 16/01/2017 09:53
File folder

Figure 3-1: Creating the “Tutorial1_XraysOnOff” folder

- Open the folder, which will contain a folder named “IPCTemplate” and a Microsoft Visual Studio Solution.

  Rename the folder again to “Tutorial1_XraysOnOff”, but do not rename the Microsoft Visual Studio Solution.

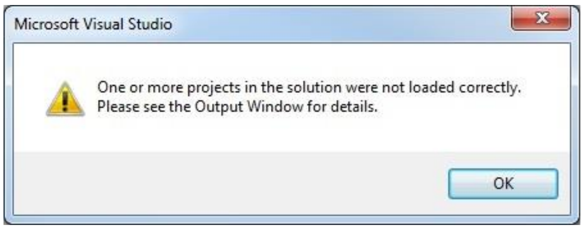


Name	Date modified	Type	Size
Tutorial1_XraysOnOff	16/01/2017 09:51	File folder	
IPCTemplate	30/03/2016 15:11	Microsoft Visual S...	5 KB

IPCTemplate Date modified: 30/03/2016 15:11 Date created: 16/01/2017 09:51
Microsoft Visual Studio Solution Size: 4.85 KB

Figure 3-2: Rename the folder but not the Microsoft Visual Studio Solution

- Double-click the Microsoft Visual Studio Solution to open it in Visual Studio, ignoring the following warning:



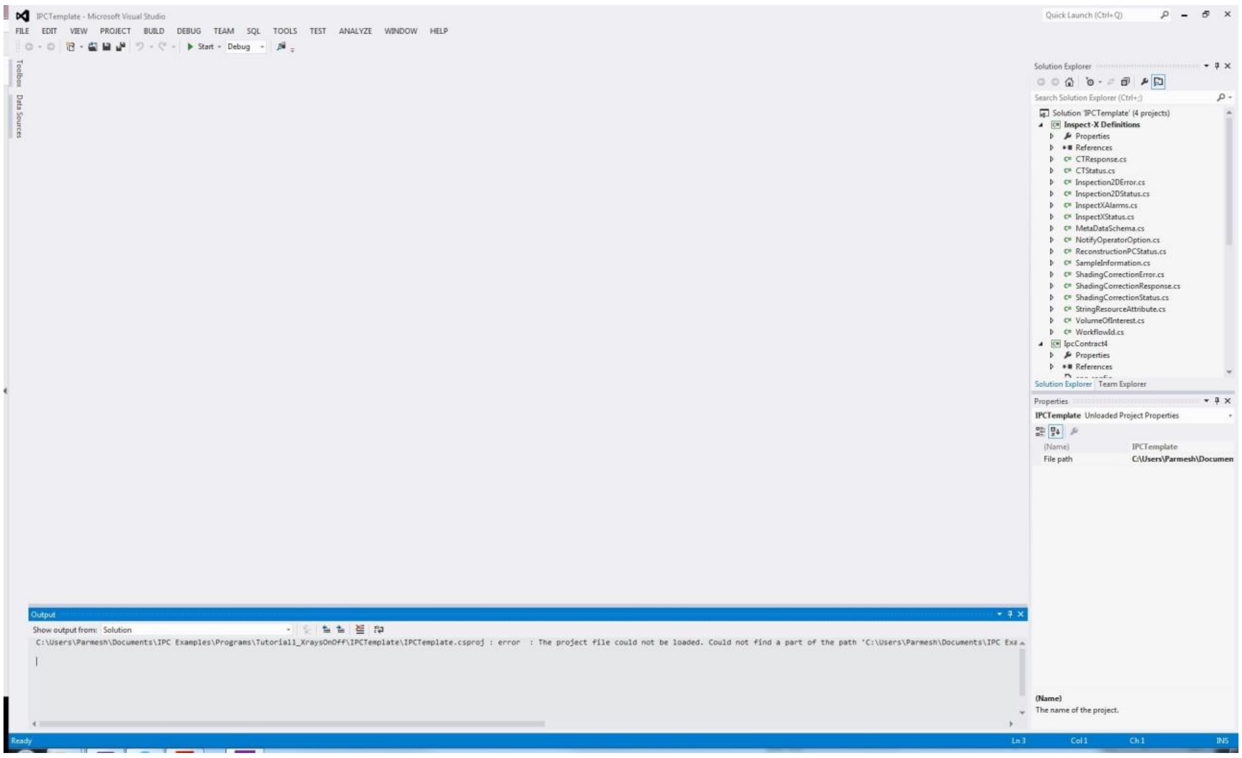
Microsoft Visual Studio

One or more projects in the solution were not loaded correctly.
Please see the Output Window for details.

OK

Figure 3-3: Ignore the above warning that arises when loading the project. This will be corrected shortly.

The warning arises because we renamed one of the folders, and we will correct this shortly. You should be presented with a screen as follows:

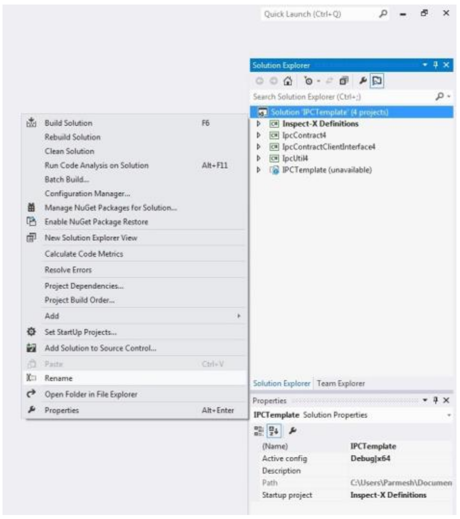


```
C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\IPCTemplate\IPCTemplate.csproj : error  : The project file could not be loaded. Could not find a part of the path 'C:\Users\Parmesh\Documents\IPC Exa
```

Figure 3-4: Initial project screen in Visual Studio

There are a few small 'fiddly' things that we will set up and rename from with Visual Studio:

- Firstly, find "Solution 'IPCTemplate (4 Projects)" at the top of the Solution Explorer (usually on the right-hand edge of the screen).
- Right click on this (or press **F2**), and rename it to "Tutorial1_XraysOnOff".



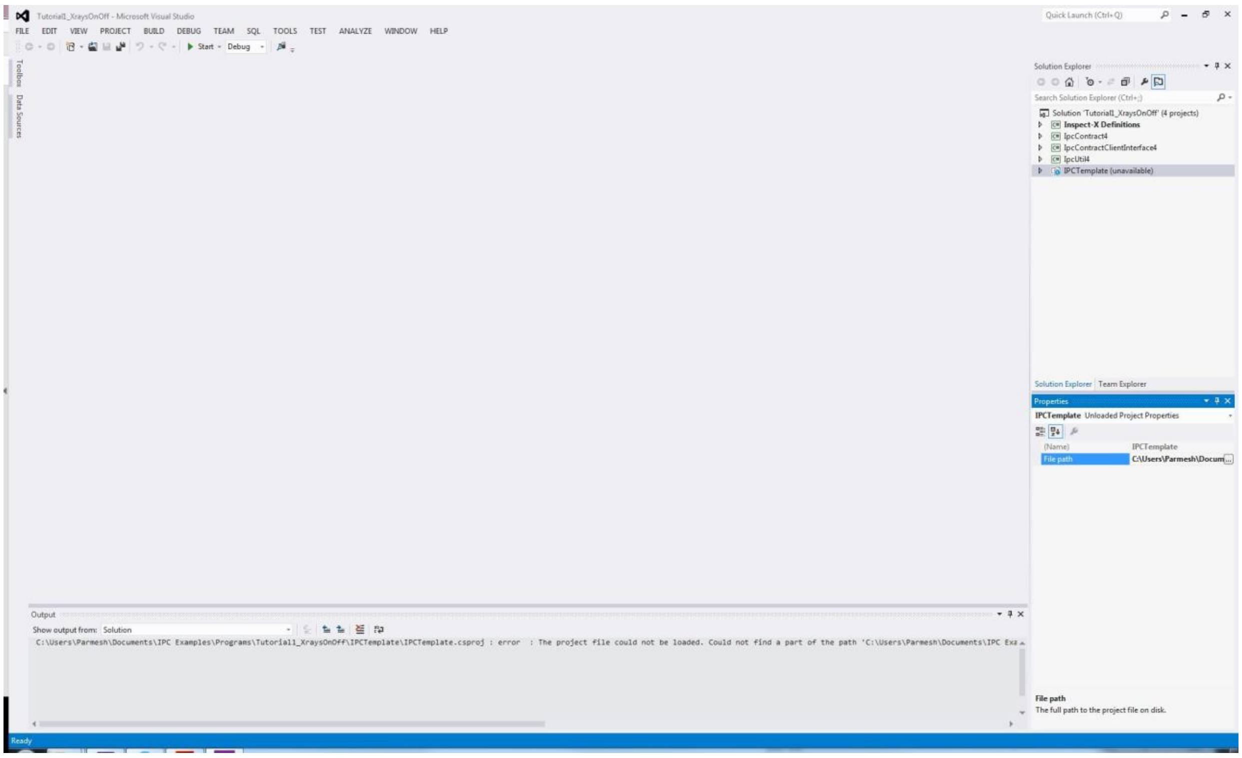
IPCTemplate Solution Properties

(Name)	IPCTemplate
Active config	Debug|x64
Description	
Path	C:\Users\Parmesh\Documen
Startup project	Inspect-X Definitions

Figure 3-5: Rename the entire Project Solution

- To correct the error when loading the solution, find the "IPCTemplate" project (next to a blue rectangular box containing an exclamation mark) and select it. It will have "unavailable" written next to it.

  In the **Properties** pane (usually below the Solution Explorer), find "File path".


Tutorial1_XraysOnOff - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW HELP
Start
Debug
Quick Launch (Ctrl+Q)
Toolbox
Data Sources
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Tutorial1_XraysOnOff' (4 projects)
Inspect-X Definitions
IpcContract4
IpcContractClientInterface4
IpcUtil4
IPCTemplate (unavailable)
Solution Explorer
Team Explorer
Properties
Output
Show output from: Solution
Ready

IPCTemplate Unloaded Project Properties

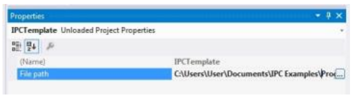
(Name)	IPCTemplate
File path	C:\Users\Parmesh\Docum

File path
The full path to the project file on disk.

```
C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\IPCTemplate\IPCTemplate.csproj : error  : The project file could not be loaded. Could not find a part of the path 'C:\Users\Parmesh\Documents\IPC Exa
```

Figure 3-6: Find the File path property for the IPCTemplate in the Properties pane

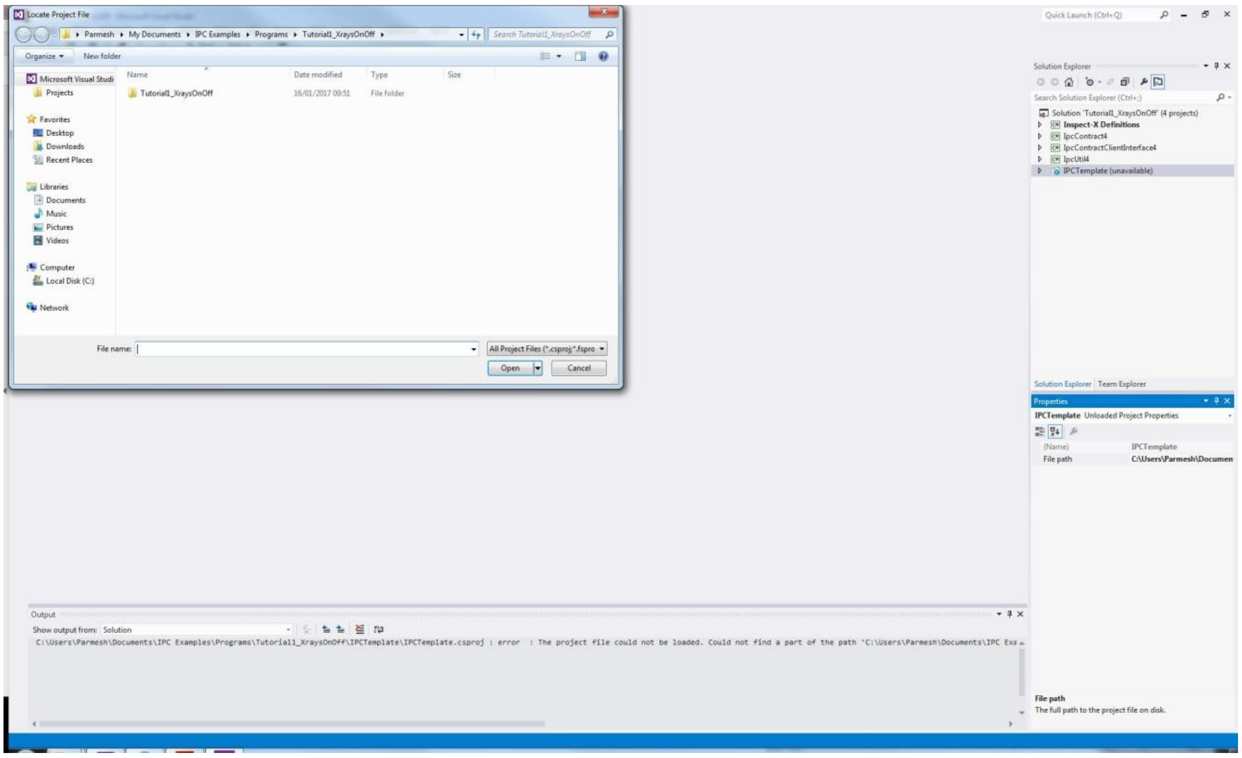
Click the “File path” and then click the ellipsis (…) on the right-hand side.



(Name)	IPCTemplate
File path	C:\Users\User\Documents\IPC Examples\Pro…

Figure 3-7: Clicking on the file path shows the ellipsis to locate the project file

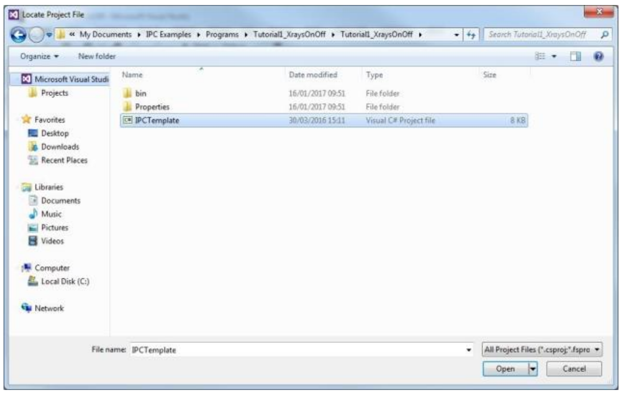
This will bring up a dialog box in which there is one folder showing, namely “Tutorial1_XraysOnOff”.


Locate Project File
Parmesh ▸ My Documents ▸ IPC Examples ▸ Programs ▸ Tutorial1_XraysOnOff ▸
Organize ▾
New folder
Name
Date modified
Type
Size
Tutorial1_XraysOnOff
16/01/2017 09:51
File folder
File name:
All Project Files (*.csproj;*.fspro
Open
Cancel
Solution Explorer
Solution 'Tutorial1_XraysOnOff' (4 projects)
Inspect-X Definitions
IpcContract4
IpcContractClientInterface4
IpcUtil4
IPCTemplate (unavailable)
Properties
IPCTemplate Unloaded Project Properties
(Name) IPCTemplate
File path C:\Users\Parmesh\Documen
File path
The full path to the project file on disk.
Output
Show output from: Solution

```
C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\IPCTemplate\IPCTemplate.csproj : error  : The project file could not be loaded. Could not find a part of the path 'C:\Users\Parmesh\Documents\IPC Exa
```

Figure 3-8: The first dialog should contain one folder called “Tutorial1_XraysOnOff”

Enter this folder, and select the Visual C# Project File called “IPCTemplate”.


Locate Project File
« My Documents ▸ IPC Examples ▸ Programs ▸ Tutorial1_XraysOnOff ▸ Tutorial1_XraysOnOff ▸
Organize ▾
New folder
File name: IPCTemplate
All Project Files (*.csproj;*.fspro
Open
Cancel

Name	Date modified	Type	Size
bin	16/01/2017 09:51	File folder	
Properties	16/01/2017 09:51	File folder	
IPCTemplate	30/03/2016 15:11	Visual C# Project file	8 KB

Figure 3-9: Select the Visual C# Project File called “IPCTemplate”

Then right-click on the IPCTemplate project in the Solution Explorer (saying unavailable next to it), and click **Reload Project**.

The unavailable should now disappear, and the blue box should be replaced by a C#.

- Rename this IPCTemplate project (by right-clicking or pressing F2) to “Tutorial1_XraysOnOff”.

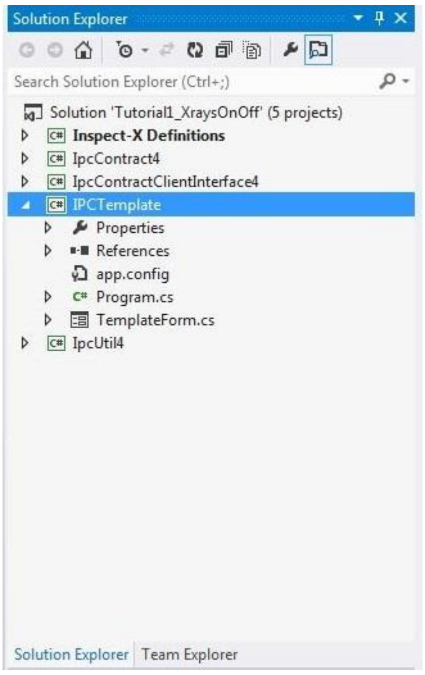
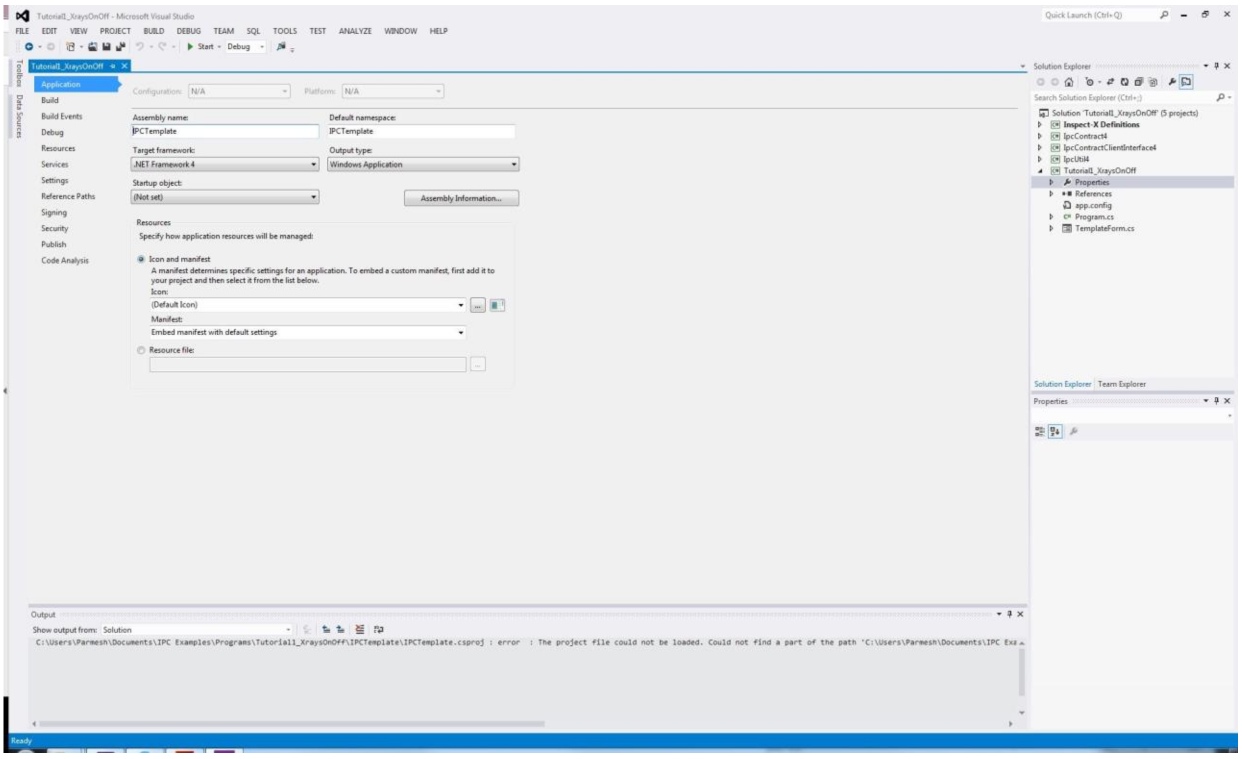


Figure 3-10: Rename the IPCTemplate project to “Tutorial1_XraysOnOff”

- Open more contents of the “Tutorial1_XraysOnOff” project by double-clicking on it in the Solution Explorer, and then double-click on **Properties.**



Configuration: N/A   Platform: N/A

Assembly name: IPCTemplate
Default namespace: IPCTemplate
Target framework: .NET Framework 4
Output type: Windows Application
Startup object: (Not set)

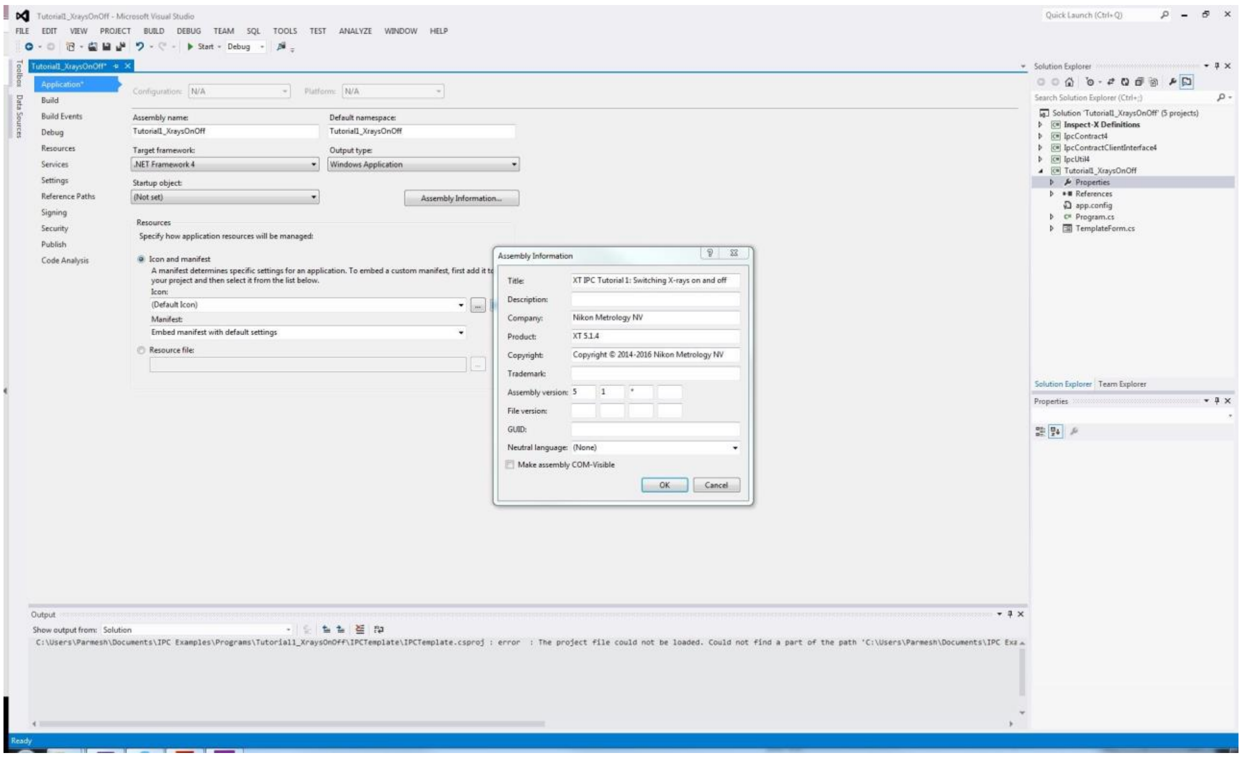
Resources
Specify how application resources will be managed:
[✓ Icon and manifest]
A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.
Icon: (Default Icon)
Manifest: Embed manifest with default settings
Resource file:

Output
Show output from: Solution

```
C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\IPCTemplate\IPCTemplate.csproj : error  : The project file could not be loaded. Could not find a part of the path 'C:\Users\Parmesh\Documents\IPC Exa
```

Figure 3-11: The properties of the “Tutorial1_XraysOnOff” project

Change both the **Assembly name** and **Default namespace** to “Tutorial1_XraysOnOff”. Click the **Assembly Information** button, and change the title to “XT IPC Tutorial 1: Switching X-rays on and off”.


Tutorial1_XraysOnOff - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW HELP
Start Debug
Tutorial1_XraysOnOff*
Application* Build Build Events Debug Resources Services Settings Reference Paths Signing Security Publish Code Analysis
Assembly Information...
Solution Explorer
Solution 'Tutorial1_XraysOnOff' (5 projects)
Inspect-X Definitions
IpcContract4
IpcContractClientInterface4
IpcUtil4
Tutorial1_XraysOnOff
Properties
References
app.config
Program.cs
TemplateForm.cs
Solution Explorer Team Explorer
Properties
Ready

Configuration: N/A   Platform: N/A

Assembly name: Tutorial1_XraysOnOff   Default namespace: Tutorial1_XraysOnOff

Target framework: .NET Framework 4   Output type: Windows Application

Startup object: (Not set)

Resources
Specify how application resources will be managed:
● Icon and manifest
A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.
Icon: (Default Icon)
Manifest: Embed manifest with default settings
○ Resource file:

Assembly Information

Title: XT IPC Tutorial 1: Switching X-rays on and off
Description:
Company: Nikon Metrology NV
Product: XT 5.1.4
Copyright: Copyright © 2014-2016 Nikon Metrology NV
Trademark:
Assembly version: 5 1 *
File version:
GUID:
Neutral language: (None)
☐ Make assembly COM-Visible
OK Cancel

Output
Show output from: Solution

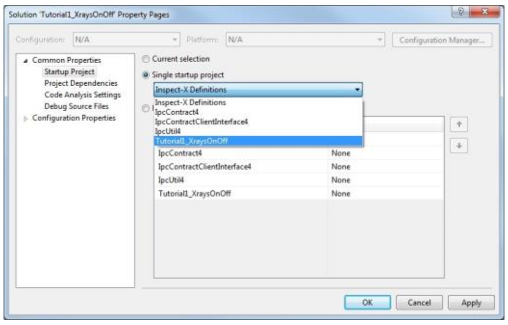
```
C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\IPCTemplate\IPCTemplate.csproj : error  : The project file could not be loaded. Could not find a part of the path 'C:\Users\Parmesh\Documents\IPC Exa
```

Figure 3-12: Changing the Assembly Information

Click **OK** on the dialog box, then CTRL-S on the keyboard to save, and close the tab.

Finally, the default startup project for the entire solution must be chosen to be our Tutorial1_XraysOnOff project.

- Select the “Solution ‘IPCTemplate (5 Projects)” at the top of the Solution Explorer, right-click and select **Properties**.
- In the list, next to **Single startup project**, select Tutorial1_XraysOnOff.



Configuration: N/A   Platform: N/A

○ Current selection
● Single startup project
Inspect-X Definitions
Inspect-X Definitions
IpcContract4
IpcContractClientInterface4
IpcUtil4
Tutorial1_XraysOnOff

IpcContract4	None
IpcContractClientInterface4	None
IpcUtil4	None
Tutorial1_XraysOnOff	None

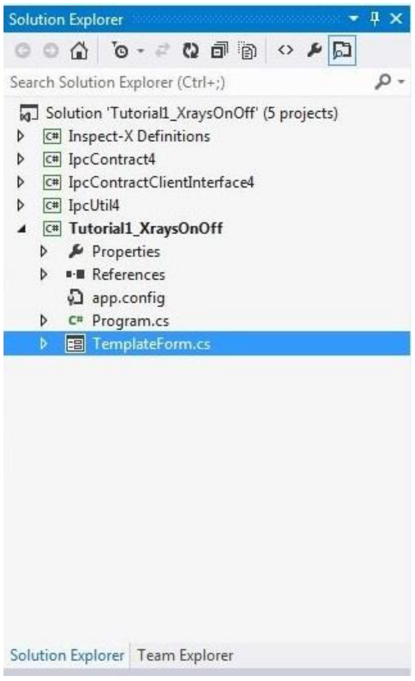
Figure 3-13: Changing the Startup project

- Click **OK** to finish.

3.2 The front-end user form and back-end code

The application consists of the front-end User Form, which is the Graphical User Interface (GUI) that the final user interacts with. Behind all of this is the back-end C# code where we specify what we want the application to do. We will change the names of the form and change the namespaces in the code in preparation for creating our application.

- Navigate to the “Tutorial1_XraysOnOff” project in the Solution Explorer, and click the horizontal arrow to reveal its components. At the bottom will be “TemplateForm.cs”.

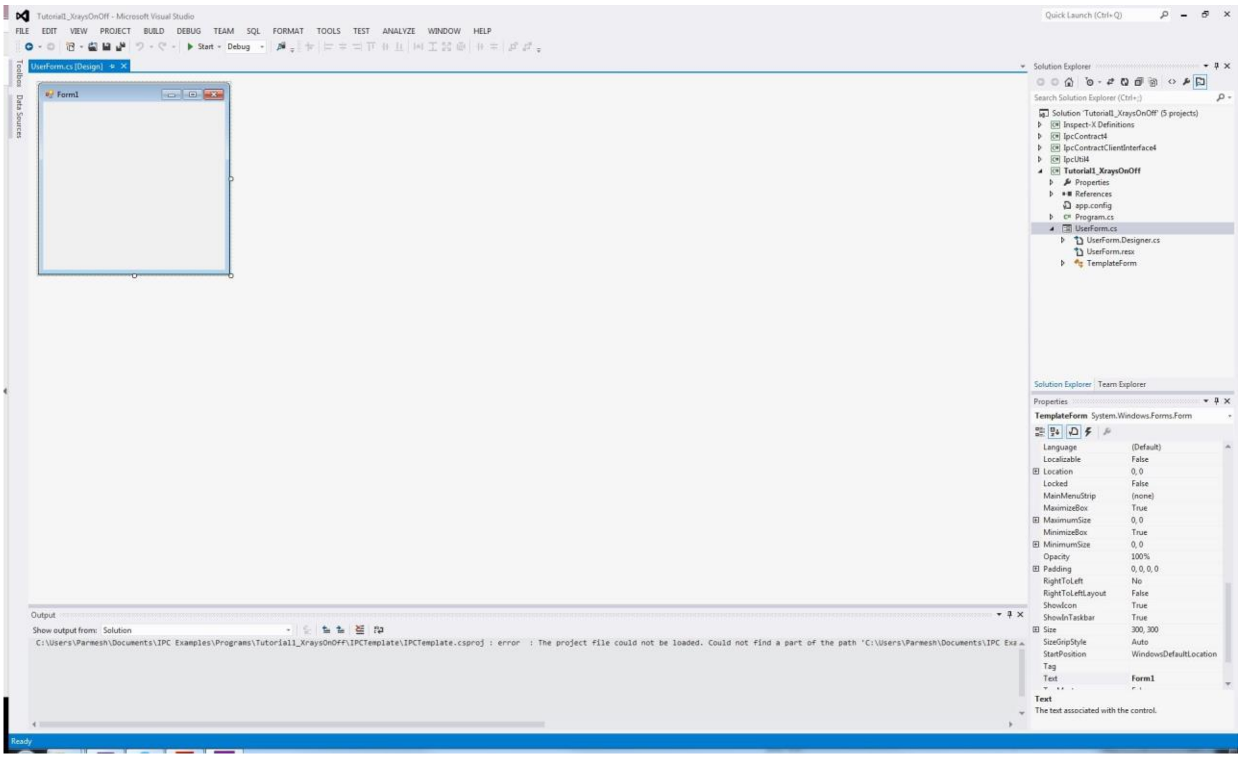


```
Solution 'Tutorial1_XraysOnOff' (5 projects)
  Inspect-X Definitions
  IpcContract4
  IpcContractClientInterface4
  IpcUtil4
  Tutorial1_XraysOnOff
    Properties
    References
    app.config
    Program.cs
    TemplateForm.cs
```

Figure 3-14: Locating TemplateForm.cs

To begin with, rename this to “UserForm.cs”. Upon renaming the form, the Solution Explorer will reveal three further components nested within “UserForm.cs”. The bottom component will still carry the name “TemplateForm”; this cannot be renamed in the usual manner, and we will rename it shortly.

- Double-clicking the UserForm.cs will bring up the Designer for the form.


Tutorial1_XraysOnOff - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL FORMAT TOOLS TEST ANALYZE WINDOW HELP
Start
Debug
Quick Launch (Ctrl+Q)
UserForm.cs [Design]
Toolbox
Data Sources
Form1
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution Explorer
Team Explorer
Properties
Output
Ready

```
Solution 'Tutorial1_XraysOnOff' (5 projects)
  Inspect-X Definitions
  IpcContract4
  IpcContractClientInterface4
  IpcUtil4
  Tutorial1_XraysOnOff
    Properties
    References
    app.config
    Program.cs
    UserForm.cs
      UserForm.Designer.cs
      UserForm.resx
      TemplateForm
```

TemplateForm System.Windows.Forms.Form

Language	(Default)
Localizable	False
⊞ Location	0, 0
Locked	False
MainMenuStrip	(none)
MaximizeBox	True
⊞ MaximumSize	0, 0
MinimizeBox	True
⊞ MinimumSize	0, 0
Opacity	100%
⊞ Padding	0, 0, 0, 0
RightToLeft	No
RightToLeftLayout	False
ShowIcon	True
ShowInTaskbar	True
⊞ Size	300, 300
SizeGripStyle	Auto
StartPosition	WindowsDefaultLocation
Tag	
Text	**Form1**

Text
The text associated with the control.

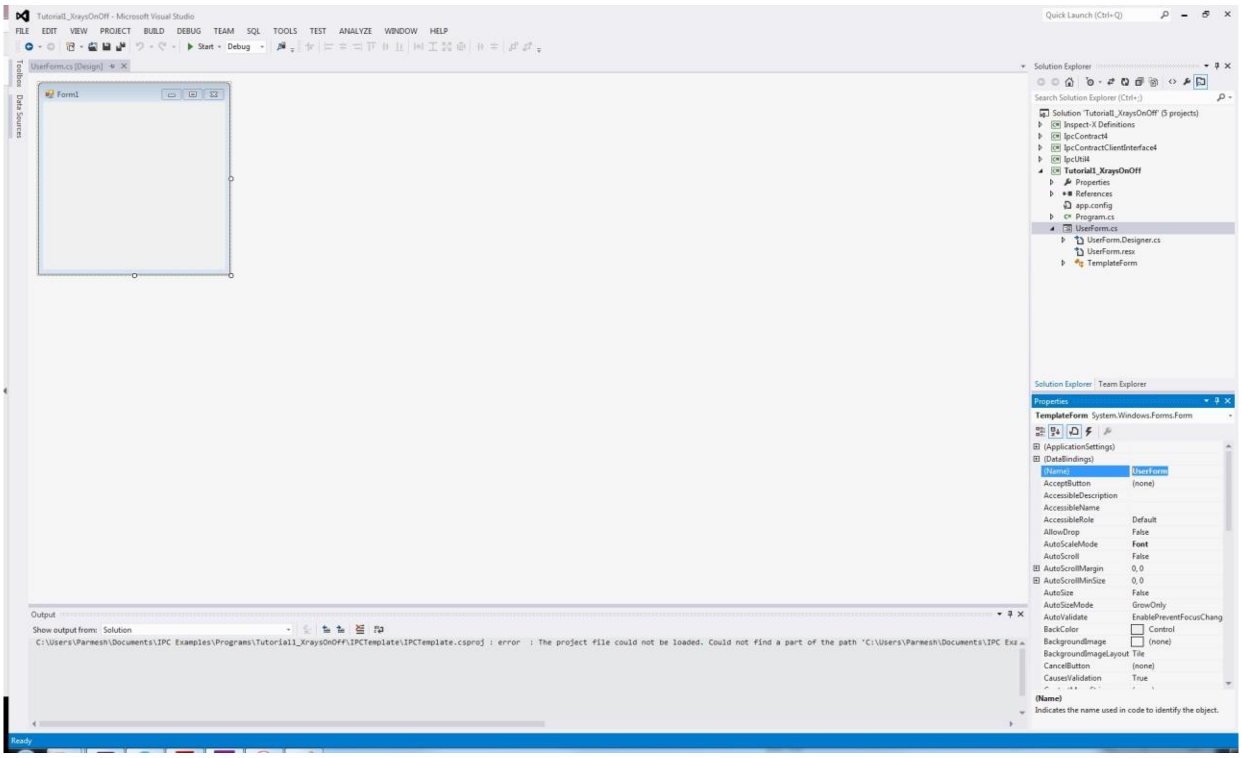
Show output from: Solution

```
C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\IPCTemplate\IPCTemplate.csproj : error  : The project file could not be loaded. Could not find a part of the path 'C:\Users\Parmesh\Documents\IPC Exa
```

Figure 3-15: The UserForm Designer

This is where we will create buttons, trackbars, menus, and so on, that the user can interact with.

The properties for each of the items on the form can be modified in the **Properties** pane. For example, when the entire form is selected, the **Name** can be changed from “TemplateForm” to “UserForm”.



```
Output
Show output from: Solution
C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\IPCTemplate\IPCTemplate.csproj : error  : The project file could not be loaded. Could not find a part of the path 'C:\Users\Parmesh\Documents\IPC Exa
```

```
Properties
TemplateForm System.Windows.Forms.Form
(ApplicationSettings)
(DataBindings)
(Name)                    UserForm
AcceptButton              (none)
AccessibleDescription
AccessibleName
AccessibleRole            Default
AllowDrop                 False
AutoScaleMode             Font
AutoScroll                False
AutoScrollMargin          0, 0
AutoScrollMinSize         0, 0
AutoSize                  False
AutoSizeMode              GrowOnly
AutoValidate              EnablePreventFocusChang
BackColor                 Control
BackgroundImage           (none)
BackgroundImageLayout     Tile
CancelButton              (none)
CausesValidation          True
(Name)
Indicates the name used in code to identify the object.
```

Figure 3-16: Changing the Form name to UserForm

Similarly, the **Text** property can be changed to “Tutorial 1- X-rays on and off”. Note that this changes the title at the top of the form.

- Click the **Start** button on the top menu to build the form, and launch a debug version of the application.

  At this stage, the form should compile successfully, and a blank form should appear on the screen.

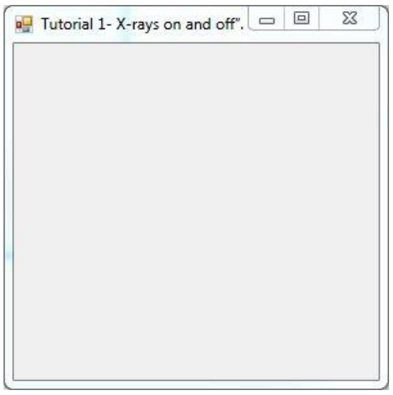
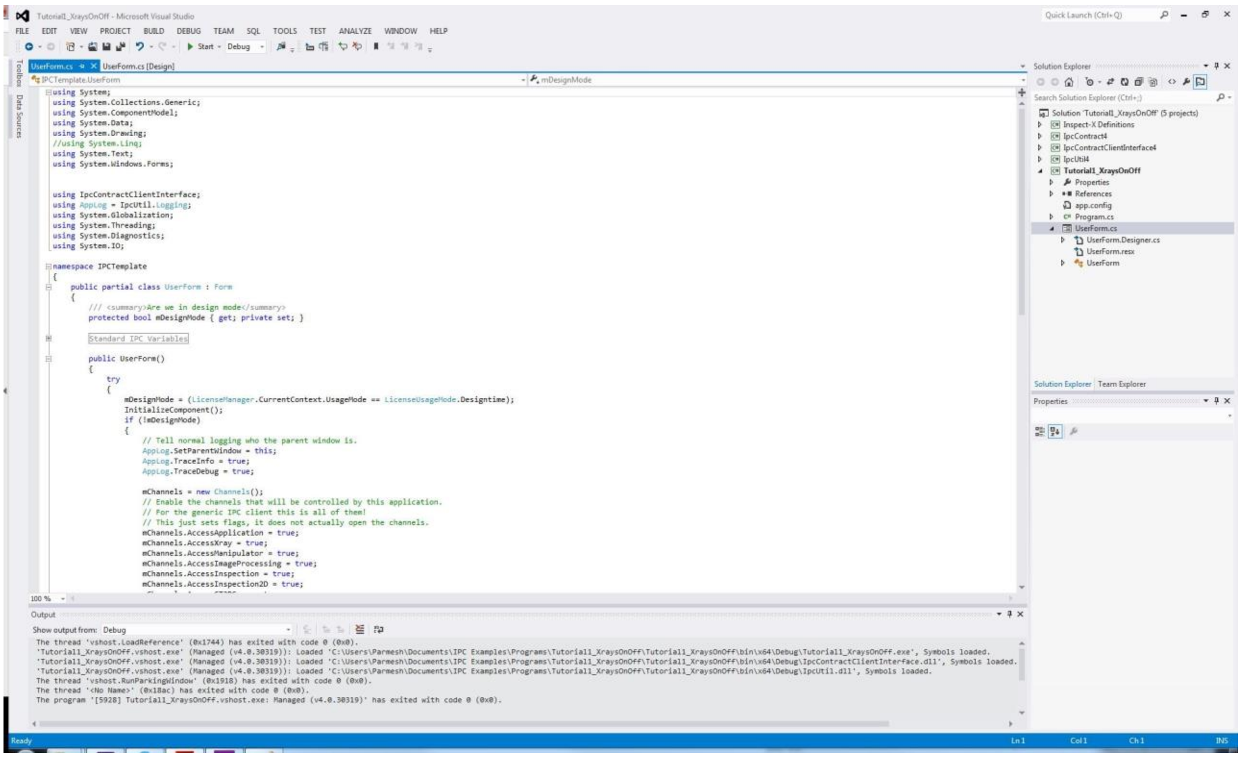


Figure 3-17: The debug application

- After closing the debug application, the back-end code can be revealed by pressing **F7**, and you can return to the Designer by pressing SHIFT+F7.



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
//using System.Linq;
using System.Text;
using System.Windows.Forms;


using IpcContractClientInterface;
using AppLog = IpcUtil.Logging;
using System.Globalization;
using System.Threading;
using System.Diagnostics;
using System.IO;

namespace IPCTemplate
{
    public partial class UserForm : Form
    {
        /// <summary>Are we in design mode</summary>
        protected bool mDesignMode { get; private set; }

        Standard IPC Variables

        public UserForm()
        {
            try
            {
                mDesignMode = (LicenseManager.CurrentContext.UsageMode == LicenseUsageMode.Designtime);
                InitializeComponent();
                if (!mDesignMode)
                {
                    // Tell normal logging who the parent window is.
                    AppLog.SetParentWindow = this;
                    AppLog.TraceInfo = true;
                    AppLog.TraceDebug = true;

                    mChannels = new Channels();
                    // Enable the channels that will be controlled by this application.
                    // For the generic IPC client this is all of them!
                    // This just sets flags, it does not actually open the channels.
                    mChannels.AccessApplication = true;
                    mChannels.AccessXray = true;
                    mChannels.AccessManipulator = true;
                    mChannels.AccessImageProcessing = true;
                    mChannels.AccessInspection = true;
                    mChannels.AccessInspection2D = true;
```

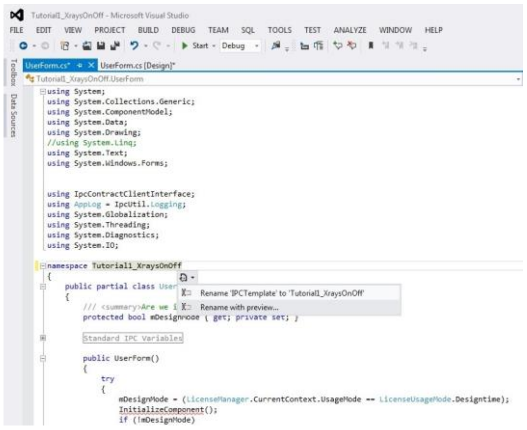
Show output from: Debug

```
The thread 'vshost.LoadReference' (0x1744) has exited with code 0 (0x0).
'Tutorial1_XraysOnOff.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\Tutorial1_XraysOnOff\bin\x64\Debug\Tutorial1_XraysOnOff.exe', Symbols loaded.
'Tutorial1_XraysOnOff.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\Tutorial1_XraysOnOff\bin\x64\Debug\IpcContractClientInterface.dll', Symbols loaded.
'Tutorial1_XraysOnOff.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\Tutorial1_XraysOnOff\bin\x64\Debug\IpcUtil.dll', Symbols loaded.
The thread 'vshost.RunParkingWindow' (0x1918) has exited with code 0 (0x0).
The thread '<No Name>' (0x18ac) has exited with code 0 (0x0).
The program '[5928] Tutorial1_XraysOnOff.vshost.exe: Managed (v4.0.30319)' has exited with code 0 (0x0).
```

Figure 3-18: The back-end code behind the UserForm

- One final renaming task remains. At the top of the code, below the references is a line marked `namespace IPCTemplate`. Change “IPCTemplate” to “Tutorial1_XraysOnOff”.

  Upon doing so, a small red box appears at the end of the new name. Hovering over this brings up the option of a menu for auto-renaming of all instances of the IPCTemplate. Click to access the menu, and choose the first option, **Rename ‘IPCTemplate’ to ‘Tutorial1_XraysOnOff’**.


Tutorial1_XraysOnOff - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW HELP
Start Debug
UserForm.cs*
UserForm.cs [Design]*
Tutorial1_XraysOnOff.UserForm
Rename 'IPCTemplate' to 'Tutorial1_XraysOnOff'
Rename with preview...

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
//using System.Linq;
using System.Text;
using System.Windows.Forms;


using IpcContractClientInterface;
using AppLog = IpcUtil.Logging;
using System.Globalization;
using System.Threading;
using System.Diagnostics;
using System.IO;

namespace Tutorial1_XraysOnOff
{
    public partial class User
    {
        /// <summary>Are we i
        protected bool mDesignMode { get; private set; }

        Standard IPC Variables

        public UserForm()
        {
            try
            {
                mDesignMode = (LicenseManager.CurrentContext.UsageMode == LicenseUsageMode.Designtime);
                InitializeComponent();
                if (!mDesignMode)
```

Figure 3-19: Changing the Namespace

  All instances within the entire solution automatically update - we can see this by the red lines under `InitialiseComponent();` disappearing.

We have now completed setting up our project, and are now ready to understand more about mechanics of the IPC code.

3.3 An overview of the code

Before we change the mechanics of the code to construct our application, it will be useful to overview the existing template code to understand the mechanics of the IPC interface.

The Nikon X-ray CT machine is controlled directly by the Inspect-X software on the acquisition computer. Inspect-X then communicates with our custom application through eight different communication channels, with each channel corresponding to a different aspect of the CT machine. This is illustrated in the following diagram:

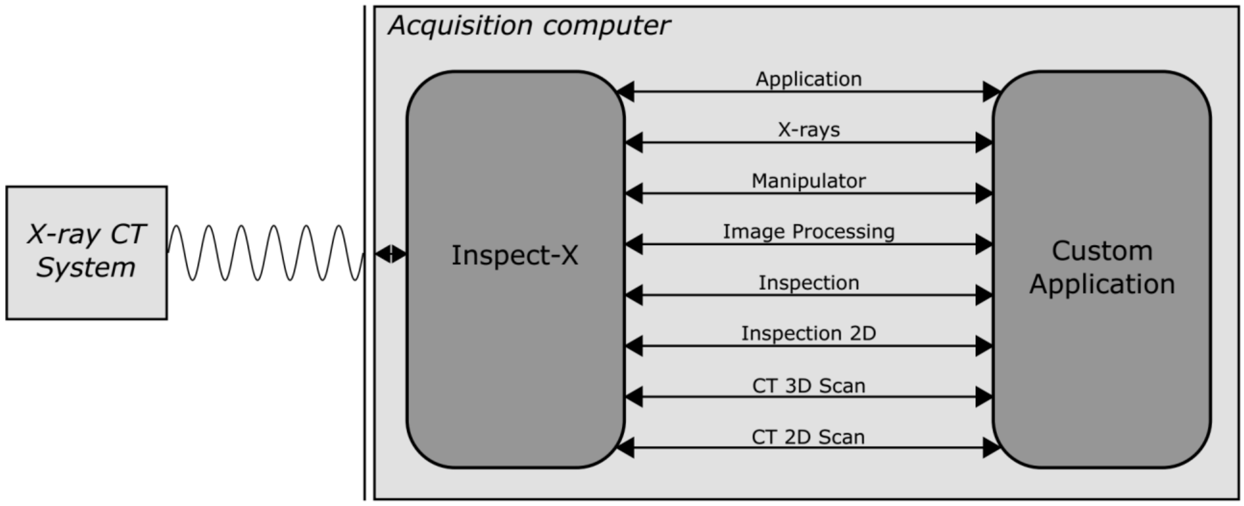


Figure 3-20: A sketch showing how communication occurs between the X-ray CT machine and the software running acquisition computer, with the client application communicating with Inspect-X, which communicates with the X-ray CT system

Communication along each channel is specific to that part of the X-ray CT system. For example, the X-rays channel allows communication with the X-ray subsystem and the Image-Processing channel allows communication with the Image Processing subsystem. As can be seen in the diagram, the Inspect-X software acts as the intermediary between the custom application and the X-ray CT system, and so Inspect-X is required to be running for our custom application to work.

Let us take a closer look at the X-ray channel to understand more about the communication:

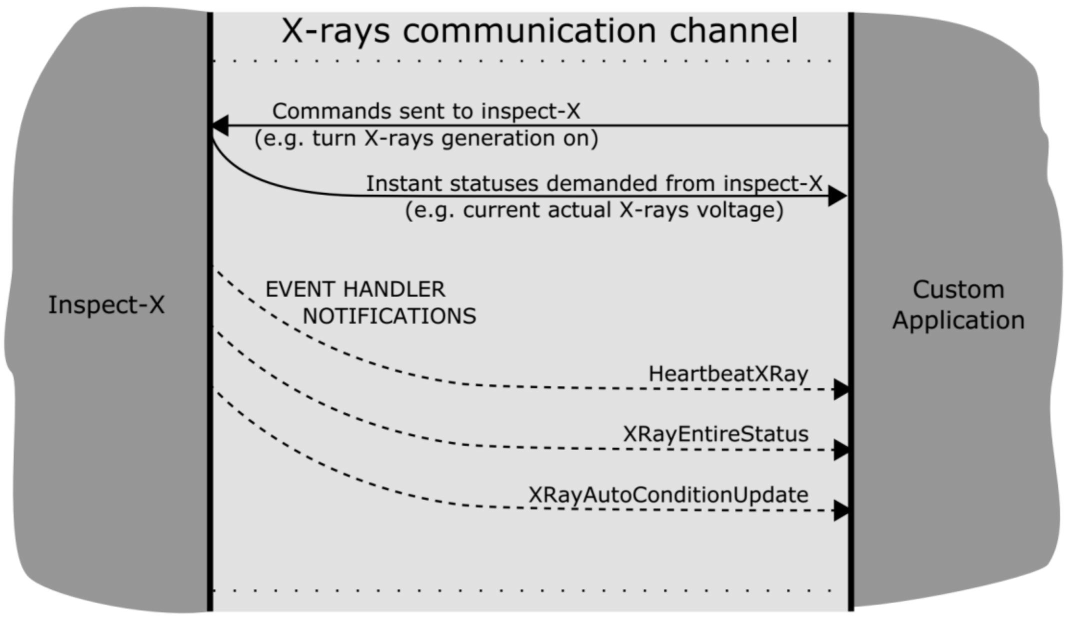

X-rays communication channel
Commands sent to inspect-X
(e.g. turn X-rays generation on)
Instant statuses demanded from inspect-X
(e.g. current actual X-rays voltage)
EVENT HANDLER NOTIFICATIONS
HeartbeatXRay
XRayEntireStatus
XRayAutoConditionUpdate
Inspect-X
Custom Application

Figure 3-21: A sketch showing how communication occurs along the X-ray channel

Once the channel is open, the custom application can send commands to Inspect-X, for example instructing X-rays to be switched on, or instructing the demand voltage to be set to a certain level. The custom application can also demand statuses from Inspect-X, for example it could request the actual current voltage on the X-ray system, and Inspect-X would immediately provide this information.

In general, communication is asynchronous: once a command is sent to Inspect-X, say to switch X-rays on, Inspect-X does not provide any feedback and our code will move onto the next line without knowing whether the command has been implemented or not. This is a significant difference to anyone familiar with the former VBA programming module, where if X-rays were switched on, then Inspect-X would feedback with the generation status and our code would wait for this status to be received before moving on. The asynchronous nature provides far greater flexibility to create complex multi-threaded application that perform many tasks simultaneously.

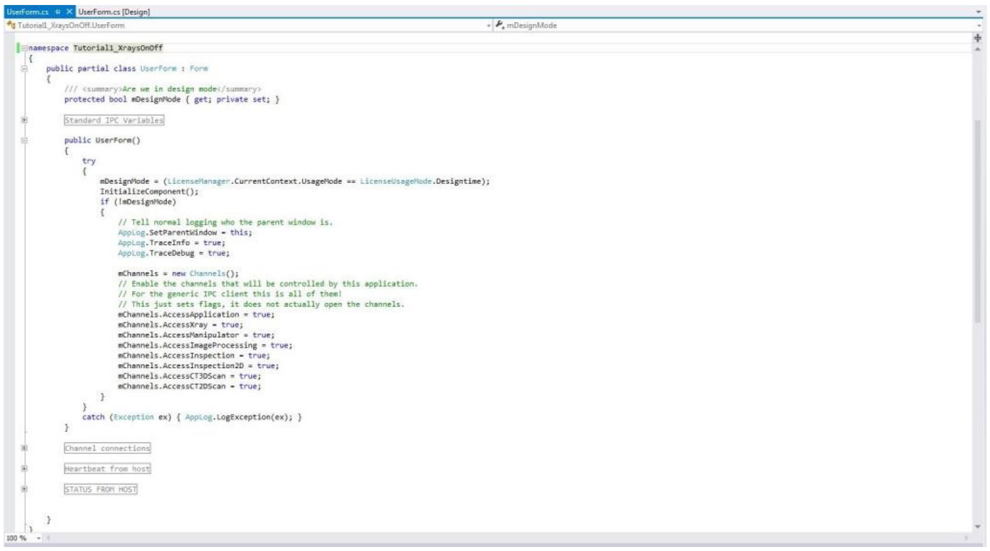
Although instantaneous feedback is not provided, Inspect-X is able to provide notifications of important events using the C# event handling mechanism. For example, when the entire X-ray status changes, the `mEventSubscriptionInspectXStatus` event is raised, and a function named `EventHandlerXRayEntireStatus` is called. These events with associated event-handlers and callback functions allow the user to create their own feedback loops. The events for a selected number communication channels are listed the following table:

Table 3-1: List of events for the application, X-rays, manipulator and image-processing channels

Channel	Event	Description
Application	mEventSubscriptionHeartbeat	Heartbeat status from Application channel
	mEventSubscriptionInspectXStatus	Status of Inspect-X
	mEventSubscriptionInspectXAlarms	Alarms raised by Inspect-X
X-rays	mEventSubscriptionHeartbeat	Heartbeat status from X-rays
	mEventSubscriptionEntireStatus	Status of X-rays
	mEventSubscriptionAutoConditionUpdate	Progress update on auto-condition
Manipulator	mEventSubscriptionHeartbeat	Heartbeat status from Manipulator
	mEventSubscriptionManipulatorMove	Manipulator move events
	mEventSubscriptionDoorStateChanged	Change in door status
	mEventSubscriptionAxisPositionChanged	Change in subscription axis position
Image-Processing	mEventSubscriptionHeartbeat	Heartbeat status from Image-Processing
	mEventSubscriptionImageProcessing	Image-Processing events

Further details and a complete list of events and event handlers for all of the channels can be found in the Inspect-X IPC Programming Manual.

We can identify different parts of the code with each part of this communication process. Scrolling down, the remaining code should appear like the figure below. If it appears that there is more code showing, then any regions encased with `#region` and `#endregion` commands can be collapsed. Conversely, any region (shown with a grey box around it) can be expanded to reveal the code within it.



```
namespace Tutorial1_XraysOnOff
{
    public partial class UserForm : Form
    {
        /// <summary>Are we in design mode</summary>
        protected bool mDesignMode { get; private set; }

        Standard IPC Variables

        public UserForm()
        {
            try
            {
                mDesignMode = (LicenseManager.CurrentContext.UsageMode == LicenseUsageMode.Designtime);
                InitializeComponent();
                if (!mDesignMode)
                {
                    // Tell normal logging who the parent window is.
                    AppLog.SetParentWindow = this;
                    AppLog.TraceInfo = true;
                    AppLog.TraceDebug = true;

                    mChannels = new Channels();
                    // Enable the channels that will be controlled by this application.
                    // For the generic IPC client this is all of them!
                    // This just sets flags, it does not actually open the channels.
                    mChannels.AccessApplication = true;
                    mChannels.AccessXray = true;
                    mChannels.AccessManipulator = true;
                    mChannels.AccessImageProcessing = true;
                    mChannels.AccessInspection = true;
                    mChannels.AccessInspection2D = true;
                    mChannels.AccessCT3DScan = true;
                    mChannels.AccessCT2DScan = true;
                }
            }
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        Channel connections

        Heartbeat from host

        STATUS FROM HOST

    }
}
```

Figure 3-22: An overview of the remaining template form code

We can see that the `namespace Tutorial1_XraysOnOff`, contains a `public class UserForm`. At the top, there is a region called “Standard IPC Variables”, followed by a function called `public UserForm()`. This is followed by three further regions called “Channel connections”, “Heartbeat from host” and “STATUS FROM HOST”. Each region contains skeleton code which we will modify and supplement to create our application.

The `public UserForm()` contains eight lines of code, which decide which communication channels are switched on:

```
mChannels.AccessApplication = true;

mChannels.AccessXray = true;

mChannels.AccessManipulator = true;

mChannels.AccessImageProcessing = true;

mChannels.AccessInspection = true;

mChannels.AccessInspection2D = true;

mChannels.AccessCT3DScan = true;

mChannels.AccessCT2DScan = true;
```

To switch a communication channel off, we can simply change `true` to `false`. The “Channel connections” region contains functions which open and close communication along the specified channels. We can also decide on which event notifications we add to each channel. Whenever an event notification is raised, we can decide how we wish to react and process them. The functions called every time a “Heartbeat” event notification is received are defined in the “Heartbeat from host” region, and code for all of the other event notifications can be found in the “STATUS FROM HOST” region.

Let us examine each of these in further detail as we develop our application to turn X-rays on and off.

3.4 Channel connections

3.4.1 Specifying the channels to switch on

In the template form, the eight lines within the `UserForm` function shows that all the channels are switched on. For this tutorial, we will use only the Application channel and the X-rays channel. The Application channel is needed to open all communication with Inspect-X, that is, for signals on the X-ray, Manipulator and Imaging Processing channels, and so on, to occur. This application will also be using X-rays, so we need the X-ray channel.

- Leave the flags for the Application and X-ray channels as true, and set the flags for the other six channels to false.

3.4.2 Opening the channels and attaching event handlers

The `Channel connections` region defines opening and closing of the communication channels. If the region has been collapsed, it will say `Channel connections` with a grey box around it. Double-clicking this will expand the region, revealing two functions called:

```
private Channels.EConnectionState ChannelsAttach()

private bool ChannelsDetach()
```

Firstly, the `ChannelsAttach` function opens the channels specified earlier opened with the line:

```
Channels.EConnectionState State = mChannels.Connect();
```

The two functions also let us attach/detach event handlers to the appropriate channels for the events that we wish to monitor. The template displays all of the events for each of the channels, with the events grouped in '`if`' clauses for the communication channel they correspond to. The code will only be activated, however, for the channels set to `true` in ***Specifying the channels to switch on*** (on page 17), and so it is perfectly safe to delete code for any channel that is switched off.

Examine the `ChannelsAttach` function, and consider the '`if`' clause for the Application channel:

```
if (mChannels.Application != null)

{

        mChannels.Application.mEventSubscriptionHeartbeat.Event

        += new EventHandler

        <CommunicationsChannel_Application.EventArgsHeartbeat>

        (EventHandlerHeartbeatApp);

        mChannels.Application.mEventSubscriptionInspectXStatus.Event

        += new EventHandler

        <CommunicationsChannel_Application.EventArgsInspectXStatus>

        (EventHandlerInspectXStatus);

        mChannels.Application.mEventSubscriptionInspectXAlarms.Event

        += new EventHandler

        <CommunicationsChannel_Application.EventArgsInspectXAlarms>

        (EventHandlerInspectXAlarms);

}
```

The code begins by querying whether `mChannels.Application` it set to true. If it is, then event handlers are attached (through the **+=**) for the all three events within the Application channel (see ***An overview of the code*** (on page 14)), namely for the `Application Heartbeat`, for the Inspect-X status and for Inspect-X alarms. For example, the `mEventSubscriptionInspectXAlarms` event has an event-handler called `EventArgsInspectXAlarms`. When Inspect-X notifies our application of one of these events, the event handler will call the functions in parentheses, which are appropriately known as callback functions. For example, the `EventHandlerInspectXAlarms` callback function will be executed by the `EventArgsInspectXAlarms` event handler whenever Inspect-X raises an Inspect-X Alarm in the Application communication channel.

- For this simple first program that switches X-rays on and off, we will only be interested in the following events:

  ```
  mChannels.Application.mEventSubscriptionHeartbeat.Event
  mChannels.Xray.mEventSubscriptionHeartbeat.Event
  mChannels.Xray.mEventSubscriptionEntireStatus.Event
  ```

  Code for all of the other events can be deleted from the `ChannelsAttach` functions. In particular, all of the '`if`' clauses for the channels apart from `mChannels.Application` and `mChannels.Xray` can be deleted, and the any event within these two channels that is not in the list above can be deleted.

For more information or simple tutorials on events and event handlers in C#, please see the further resources in ***Getting started with C#*** (on page 2).

3.4.3 Detaching event handlers

Similarly, the `ChannelsDetach` function removes event handlers from the communication channels when we have finished. Only those event handlers which have been attached in ***Opening the channels and attaching event handlers*** (on page 17) need to be removed.

- The code for the `ChannelsDetach` function can be simplified by deleting those event handlers we are not interested in, leaving only code for `mEventSubscriptionHeartbeat`, `mEventSubscriptionHeartbeat` and `mEventSubscriptionEntireStatus`.

Notice that in this `ChannelsDetach` function, the event handlers are detached using the `-=` command, rather than the `+=` used to attach the event handlers in ***Opening the channels and attaching event handlers*** (on page 17).

The code can be checked against the sample code for ***tutorial 1*** (on page 35).

3.4.4 Linking channel connections to the user form

The final part of establishing the channel connections it to invoke the `ChannelsAttach` function when the form is loaded and the `ChannelsDetach` function when the form is closed.

1. Firstly, find the region at the top of the code called “Standard IPC Variables”. Immediately after this region (that is, after `#endregion Standard IPC Variables`), create a new region called Application Variables through the following code:

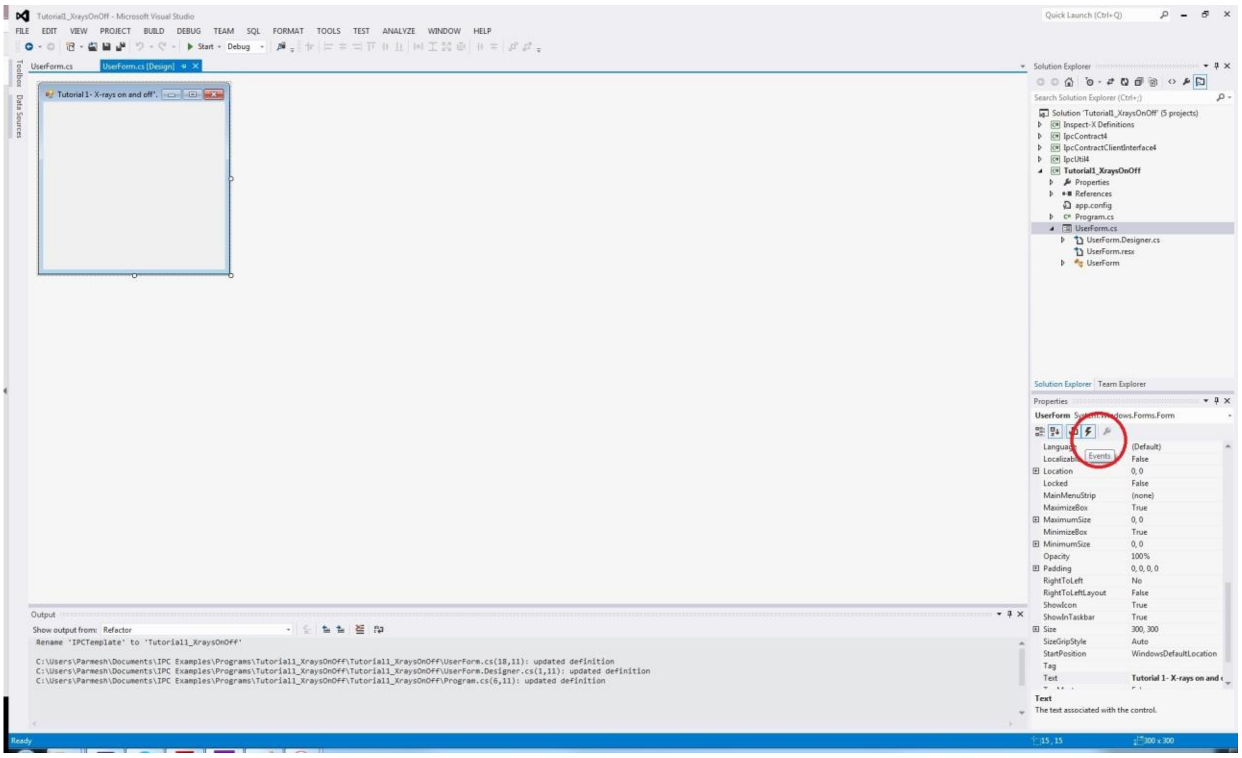
   ```
   #region Application Variables

   #endregion Application Variables
   ```

   We will add variables to this section as we develop our application. To begin with, we will initialise a variable that will keep track of the connection status of our application.
2. Type the following code within the new region:

   ```
   /// <summary> Status of the application </summary>
   private Channels.EConnectionState mApplicationState;
   ```

   This creates a new variable that will hold the application status.

3. Next, open the Designer for the form by pressing SHIFT+F7. Select the entire form, and navigate to the **Properties** pane. The Events associated with the form can be selected by pressing the button with a lightning strike in it.



```
Output
Show output from: Refactor
Rename 'IPCTemplate' to 'Tutorial1_XraysOnOff'

C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\Tutorial1_XraysOnOff\UserForm.cs(18,11): updated definition
C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\Tutorial1_XraysOnOff\UserForm.Designer.cs(1,11): updated definition
C:\Users\Parmesh\Documents\IPC Examples\Programs\Tutorial1_XraysOnOff\Tutorial1_XraysOnOff\Program.cs(6,11): updated definition
```

Language	(Default)
Localizable	False
⊞ Location	0, 0
Locked	False
MainMenuStrip	(none)
MaximizeBox	True
⊞ MaximumSize	0, 0
MinimizeBox	True
⊞ MinimumSize	0, 0
Opacity	100%
⊞ Padding	0, 0, 0, 0
RightToLeft	No
RightToLeftLayout	False
ShowIcon	True
ShowInTaskbar	True
⊞ Size	300, 300
SizeGripStyle	Auto
StartPosition	WindowsDefaultLocation
Tag	
Text	**Tutorial 1- X-rays on and**

Text
The text associated with the control.

15 , 15 300 x 300

Figure 3-23: Events associated with Form

Find “Load” and double click on it to create a new form event associated with loading the form. This code will be executed every time the form is loaded. This will take us back to the code-editor, where a new function will have been created:

```
private void UserForm_Load(object sender, EventArgs e)
{

}
```

Within this function, place the following code to attach the channels, whilst catching any exceptions that may be raised:

```
try
{
        // Attach channels
        mApplicationState = ChannelsAttach();
}
catch (Exception ex) { AppLog.LogException(ex); }
```

4. Similarly, create a new event for “FormClosing”, and place within it the following code:

```
try
{
        // Detach channels
        ChannelsDetach();
}
catch (Exception ex) { AppLog.LogException(ex); }
```

5. To keep the entire code tidy, place these form functions within a new region called “Form Functions”.

3.5 Defining callback functions

Now that we have initialised the channel connections, we must define how we want our application to process signals that it receives from Inspect-X about the X-ray CT system. This is done by defining the callback functions, which we invoked in the circular parentheses when attaching and detaching our channels above. The asynchronous nature means that we have the flexibility to react immediately within our program when an event is raised or simply store the state in a variable, which we can look up later. For example, if an Alarm event is raised, then we may want to immediately close our application, but if the X-rays status event is raised, then we may store the state in a variable which we can look up later.

After simplifying our code in ***Specifying the channels to switch on*** (on page 17) to remove the code for events we are not interested in, we should be left with code that handles the following three events:

```
mChannels.Application.mEventSubscriptionHeartbeat.Event

mChannels.Xray.mEventSubscriptionHeartbeat.Event

mChannels.Xray.mEventSubscriptionEntireStatus.Event
```

which have the associated callback functions given in parentheses:

```
EventHandlerHeartbeatApp

EventHandlerHeartbeatXRay

EventHandlerXRayEntireStatus
```

3.5.1 Heartbeat callback functions

The first two of these callback functions respond to the ‘heartbeat’ that each of the channels produces to indicate the status of the communication channels. The “Heartbeat from Host” region contains callback functions for each of the eight communication channels.

- At this stage, the heartbeat callback functions for the six channels we are not using can be discarded.

We are left with the `EventHandlerHeartbeatApp` and `EventHandlerHeartbeatXRay` functions. The code for the former is shown below:

```
void EventHandlerHeartbeatApp(object aSender, CommunicationsChannel_Application.EventArgsHeartbeat e)
{
        try
        {
                if (mChannels == null || mChannels.Application == null)
                return;
                if (this.InvokeRequired)
                        this.BeginInvoke((MethodInvoker)delegate
                        { EventHandlerHeartbeatApp(aSender, e); });
                else
                {
                        //your code goes here....
                }
        }
                catch (ObjectDisposedException) { } // ignore
                catch (Exception ex) { AppLog.LogException(ex); }
}
```

The code which we want to be executed when a heartbeat signal is received is placed where it says `// your code goes here`. For example, we may have a heartbeat animation on the User Interface form or a text box containing the number of seconds that the client has been connected. We will not place any code here in this tutorial, but the IPCDemo contains an example of a heartbeat animation.

3.5.2 All other callback functions

All the other callback functions are handled in the “STATUS from host” region. Opening the region will show 8 further regions for each communication channel. Each region contains skeleton code for each of the possible non-heartbeat event callback functions on that channel.

- Our sample tutorial program only has one callback function left, namely

  ```
  EventHandlerXRayEntireStatus
  ```

  which corresponds to the event:

  ```
  mChannels.Xray.mEventSubscriptionEntireStatus.Event
  ```

  To keep the code succinct, all the other callback functions can be deleted.

Examining the one remaining callback function, we can see from the name that this event is fired every time Inspect-X sends a new signal with an update on the Entire Status of the X-ray system. Each of the different status messages that could be received are listed as different cases in the switch clause; for each case, we have space to insert our own code when that particular status is received.

Let us create a new variable called `XRaysStable`, that will act as a flag that turns to true when a success status is received. Later on, we can check whether X-rays have stabilised by checking the value of this variable.

1. Firstly, create a `private Boolean` variable in the Application Variables region called `mXraysStable` and initialise it to `false`.

   ```
   /// <summary> Flag for X-ray stability </summary>
   private Boolean mXraysStable = false;
   ```

2. Next, within the status callback function, find the case for “Success”. Under this, set the `mXraysStable` flag to be true. The flag will remain false until the X-rays are both on and stable, when it will be set to true.
3. Within the status callback function, set the `mXraysStable` flag to be false under the “SwitchedOff” case. This will reset the flag when X-rays are turned off.

A good debugging tip is to `Debug.Print` the statuses received in the callback functions to see what signals are actually coming from Inspect-X, and when. See ***Debugging*** (on page 24) for more details.

Now we are ready to write our custom routine for switching X-rays on, wait for stability and then wait five seconds before turning off.

3.6 X-ray routine

Let us construct a short routine to switch the X-rays on, wait for stability and then wait a further five seconds before turning the X-rays off again.

Rather than copying and pasting code, it is recommended to type the code and use IntelliSense to understand how different variables in IPC are nested.

- After the last region in the code (which should now be the Form Functions region), create a new region called X-ray functions. Within this, create a new `private void` function called `XrayRoutine` that has no arguments.

  ```
  private void XrayRoutine()
  {

  }
  ```

- The first thing we will do is check that the channels we attached correctly; else none of the subsequent operations will work.

```
// If ApplicationState is not connected then immediately exit the routine
if (mApplicationState != Channels.EConnectionState.Connected)
        return;
```

- Next we set the `mXraysStable` flag to `false`. This is so that the flag is correctly set to true only after the X-rays have been turned on in this execution.

```
// Set mXraysStable flag to false
mXraysStable = false;
```

- It is then time to turn the X-rays on:

```
// Turn the X-rays on.
mChannels.Xray.XRays.GenerationDemand(true);
```

- We will then wait in a loop until we receive a signal that the X-rays have stabilised. We check whether the X-rays flag is `false`, and if so we wait for 5 milliseconds and reassess. Once the X-rays flag turns to `true`, this would loop would end.

```
// Wait until X-rays have stabilised
while (!mXraysStable)
            Thread.Sleep(5);
```

- Once the X-rays have stabilised, we then wait for five seconds (5000 milliseconds).

```
// Once stable, wait for a further 5 seconds
Thread.Sleep(5000);
```

- Finally, we turn the X-rays off, and wait until they have turned off by checking that the `mXraysStable` returns to `false`.

```
// Turn the X-rays off
mChannels.Xray.XRays.GenerationDemand(false);

// Wait until X-rays have turned off
while (mXraysStable)
        Thread.Sleep(5);
```

3.7 Initialising a new thread

In order for the communication from Inspect-X to be handled in a parallel manner, at the same time as our own routine is running, we must execute our routine on a new thread.

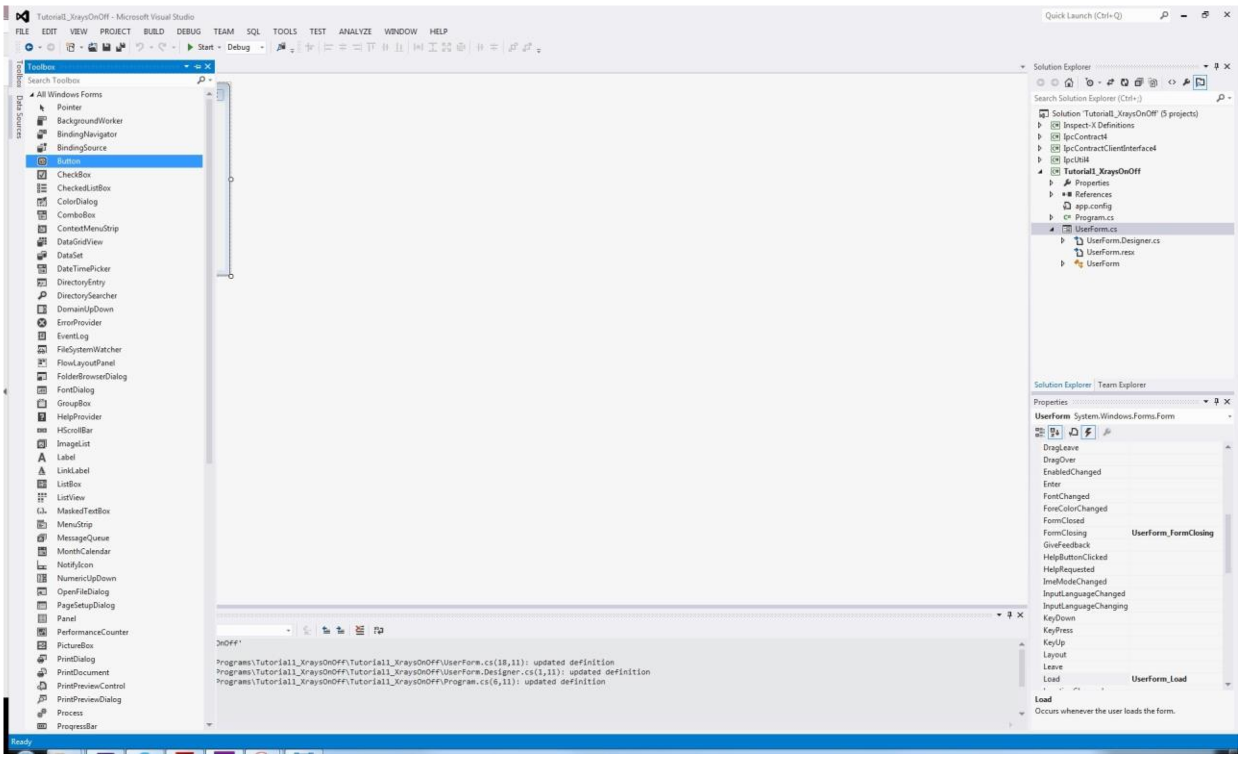
- To set up for this, create a new `private Thread` variable in the Application Variables region and initialise it to null:

```
/// <summary> Thread for X-ray Routine </summary>
private Thread mXrayRoutineThread = null;
```

3.8 Finalising the user interface

In order for our user to execute our X-ray routine, we need a button on the User Interface for them to click Start.

- Go to the Form Designer (SHIFT+F7) and create a button from the toolbox on the left.



```
OnOff'

Programs\Tutorial1_XraysOnOff\Tutorial1_XraysOnOff\UserForm.cs(18,11): updated definition
Programs\Tutorial1_XraysOnOff\Tutorial1_XraysOnOff\UserForm.Designer.cs(1,11): updated definition
Programs\Tutorial1_XraysOnOff\Tutorial1_XraysOnOff\Program.cs(6,11): updated definition
```

Figure 3-24: Creating a button on the UserForm

After resizing it to an appropriate size, go to the **Properties** pane. Change the **Name** to “btn_Start” and the **Text** to “Start”.

- Double-clicking on the button will generate a stub function `btn_Start_Click` for code that is executed when the button is clicked.

```
private void btn_Start_Click(object sender, EventArgs e)
{

}
```

- We wish to run our X-ray routine, so within the `btn_Start_Click` function, we assign our function to the thread we created, and then start it.

```
// Assign the XrayRoutine to the mXrayRoutineThread
mXrayRoutineThread = new Thread(XrayRoutine);
// Start the thread
mXrayRoutineThread.Start();
```

- The `btn_Start_Click` routine can be moved to the “Form Functions” region to keep similar functions together.

One final safety measure remains. Whilst our `XrayRoutine` is running, we do not want the user to be able to press the **Start** button again (and thus initialise another thread which runs the `XrayRoutine`). To prevent this, we disable the **Start** button immediately when the `XrayRoutine` is started, and enable it again when the Routine finishes.

As we have initialised a new thread for `XrayRoutine`, we are on a different thread to the User Form functions, and so the Invoke property is needed to change Form properties from within the `XrayRoutine` function.

- Go to the `XrayRoutine` function, and at the top of the function will be three lines that check whether the channels have been attached correctly.

  After this, place the following code that disables the **Start** button:

```
// For safety, disable the Start button
this.Invoke((MethodInvoker)delegate { btn_Start.Enabled = false; });
```

- To re-enable the **Start** button at the end, insert the following code immediately before the end of the `XrayRoutine`:

```
// Re-enable the Start button
this.Invoke((MethodInvoker)delegate { btn_Start.Enabled = true; });
```

We are now ready to test our simple application.

3.9 Running the application

Before testing our application, remember that Inspect-X is needed for any communication with the X-ray system to occur (see ***An overview of the code*** (on page 14)).

First, ensure that the Inspect-X is running and that there are no X-ray alarms. Then, clicking **Run** below the menu bar in Visual Studio should successfully build the application.

Clicking the **Start** button in our application will switch the X-rays on, stay on for five seconds after stability, and then automatically switch the X-rays off again.

3.10 Troubleshooting tutorial 1

3.10.1 General troubleshooting

If the program fails to build then check that each of the steps in the tutorial have been followed correctly.

If the program builds and runs successfully, but the X-rays do not come on, then ensure try manually switching X-rays on and off from Inspect-X. If this does not work, then it is likely there are alarms on the X-ray CT system which can be corrected. If it still does not work despite there being no alarms, then the problem is between Inspect-X and our application, and we need to check the code as below.

3.10.2 Debugging

A useful way to find run-time problems in our code is to use `Debug.Print` statements at important parts of the code. We will place debug statements at key points in our Tutorial 1 application.

- To check whether channels are attached correctly, find the `UserForm_Load` function. With the `try` loop, place the following code at the end:

```
if (mApplicationState == Channels.EConnectionState.Connected)
        Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Connected to Inspect-X");
else
        Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Problem in connecting to
            Inspect-X");
```

- Similarly, to check whether channels detach correctly, place the following code at the end of the `try` loop in the `TestForm_FormClosing`:

```
Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Disconnected from Inspect-X");
```

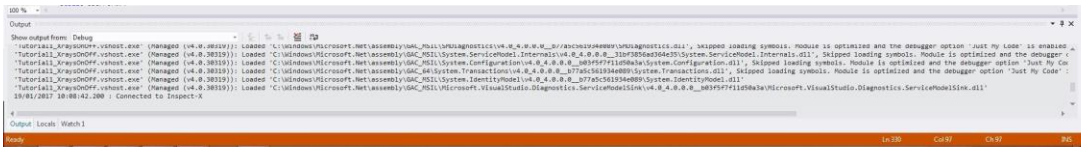
- To see what X-ray Entire Status events are being raised by Inspect-X, go to the `EventHandlerXRayEntireStatus` function (which is found in the 'STATUS FROM HOST' region, nested under 'XRay'). Immediately above the `switch` loop, where it says `// Your code goes here...`, place the following line:

```
Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " :
    e.EntireStatus.XRaysStatus.GenerationStatus.State=" +
    e.EntireStatus.XRaysStatus.GenerationStatus.State.ToString());
```

- To check the value of `mXraysStable`, place the following code after the end of the `switch` loop:

```
Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : mXraysStable=" +
    mXraysStable.ToString());
```

The results from these debug statements will be printed to the Output window (or Immediate window depending on Visual Studio settings), which can be displayed whilst the Debug application is running by going to **Debug > Windows > Output (or Immediate)**.



```
'Tutorial1_XraysOnOff.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\SMDiagnostics\v4.0_4.0.0.0__b77a5c561934e089\SMDiagnostics.dll', Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled
'Tutorial1_XraysOnOff.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.ServiceModel.Internals\v4.0_4.0.0.0__31bf3856ad364e35\System.ServiceModel.Internals.dll', Skipped loading symbols. Module is optimized and the debugger (
'Tutorial1_XraysOnOff.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Configuration\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Configuration.dll', Skipped loading symbols. Module is optimized and the debugger option 'Just My Coc
'Tutorial1_XraysOnOff.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_64\System.Transactions\v4.0_4.0.0.0__b77a5c561934e089\System.Transactions.dll', Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' :
'Tutorial1_XraysOnOff.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.IdentityModel\v4.0_4.0.0.0__b77a5c561934e089\System.IdentityModel.dll'
'Tutorial1_XraysOnOff.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.VisualStudio.Diagnostics.ServiceModelSink\v4.0_4.0.0.0__b03f5f7f11d50a3a\Microsoft.VisualStudio.Diagnostics.ServiceModelSink.dll'
19/01/2017 10:08:42.200 : Connected to Inspect-X
```

Figure 3-25: Debug output in the Output window

3.10.3 A known bug in Inspect-X 5.1 with the X-ray Entire Status event

A known bug in the current version of Inspect-X (5.1) is that the “Success” X-ray Entire Status event is sometimes not raised. Instead, a second “WaitingForStability” event is raised. The `mXraysStable` is thus never set to true, and so our tutorial program hangs within the `XrayRoutine` waiting for stability. In this case, the Output log may look similar to the following:

```
18/01/2017 17:24:51.613 : Connected to Inspect-X

18/01/2017 17:25:33.905 :e.EntireStatus.XRaysStatus.GenerationStatus.State=SwitchedOff

18/01/2017 17:25:33.906 : mXraysStable=False

18/01/2017 17:25:36.940 :e.EntireStatus.XRaysStatus.GenerationStatus.State=WaitingForStability

18/01/2017 17:25:36.941 : mXraysStable=False

18/01/2017 17:25:38.763 :e.EntireStatus.XRaysStatus.GenerationStatus.State=WaitingForStability

18/01/2017 17:25:38.764 : mXraysStable=False
```

Shortly following the erroneous second “WaitingForStability” event, Inspect-X does in fact set the X-ray Entire Status to “Success”, but does not raise an event for it.

The simply way to rectify this bug is by manually updating the X-ray Entire Status if the “WaitingForStability” event is raised twice.

- Close the debug application, and stop the debugger.
- Add the following code to Application Variables section to declare the variables that will be used for the manual status update:

```
/// <summary> Entire Xray Status (for bug correction) summary>
private IpcContract.XRay.EntireStatus mXrayEntireStatus;

/// <summary> Generation status summary>
private IpcContract.XRay.GenerationStatus.EXRayGenerationState mXrayGenerationStatus;

/// <summary> Stability event counter summary>
private int mXraysStabilityCounter = 0;
```

- Go to the `EventHandlerXRayEntireStatus` function, and find the `switch` loop. After the `WaitingForStability` case, where it says `// Your code goes here...`, place the following code to manually update the X-ray Entire Status:

```
// Increment stability counter;
mXraysStabilityCounter++;

// If stability counter is greater than 1 then must manually check update X-ray Entire status
if (mXraysStabilityCounter > 1)
{
        // Manual loop to update X-ray Entire Status until "Success"

        Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Manually checking for
            stability");

        do
        {
                // First sleep for a small amount of time to allow status updates
                Thread.Sleep(100);
                // Then get a updated X-ray Entire Status
                mXrayEntireStatus = mChannels.Xray.GetXRayEntireStatus();
                // Find generation part of Entire Status
                mXrayGenerationStatus = mXrayEntireStatus.XRaysStatus. GenerationStatus.State;

                Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " :
                    mXrayGenerationStatus=" + mXrayGenerationStatus.ToString());
        }
        while (mXrayGenerationStatus != IpcContract.XRay.
            GenerationStatus.EXRayGenerationState.Success);

        Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Manually found stable X-
            rays- Proceed");

        // Once "Success" obtained then set mXraysStable flag to true
        mXraysStable = true;

        // Reset stability counter
        mXraysStabilityCounter = 0;
}
```

3.10.4 Further errors

If you still have trouble compiling, building and running this first tutorial program, then you are advised to liaise with your Nikon point of contact.

3.10.5 Sample answer for tutorial 1

A sample answer for the tutorial 1, including debugging and a fix for the known bug above, can be found in ***Sample code - tutorial 1*** (on page 35) at the end of this Quick Start Guide.

4 Tutorial 2: Moving the manipulator

In this second tutorial, we will extend our knowledge and experience of the IPC system by writing a program to control the manipulator.

4.1 Set up folders, solution and project

- Following ***Creating and setting up the Visual Studio project*** (on page 5) and ***The front-end user form and back-end code*** (on page 10), create a new copy of the IPCTemplate and rename it.
- Rename the entire solution and IPCTemplate Project, changing the Project and Solution properties as necessary. Ensure that the entire solution builds successfully.
- Rename the TemplateForm.cs file, and appropriately set the form properties. Within the form code, change the Namespace.
- Test to ensure that the debug application compiles and builds successfully.

4.2 Establish channel connections and callback functions

This section is based on ***Channel connections*** (on page 17) and ***Defining callback functions*** (on page 20) in the previous ***tutorial*** (on page 5).

- As in ***Specifying the channels to switch on*** (on page 17), set the initial flags to allow access to the Application and Manipulator channels only.
- The only events we are interested in are the following:

```
mChannels.Application.mEventSubscriptionHeartbeat.Event
mChannels.Manipulator.mEventSubscriptionHeartbeat.Event
mChannels.Manipulator.mEventSubscriptionManipulatorMove.Event
```

  All other events can be removed from the `ChannelsAttach` and `ChannelsDetach` functions.
- Finally, we need to link our channels to our user form. Following ***Linking channel connections to the user form*** (on page 18), create an `mApplicationState` variable, and then assign the `ChannelsAttach` and `ChannelsDetach` functions to the “Load” and “FormClosing” events of the main user form.
- Remove any code for callback functions associated with events that we are not interested in.

4.3 Understanding manipulator move events

Before we construct our application to move the manipulator, it is worthwhile to understand how different Manipulator Move Events are raised. We can do this using `Debug.Print` of ***Debugging*** (on page 24).

- Go to the `EventHandlerManipulatorMoveEvent` callback function that is called whenever a Manipulator Move Event is raised. This can be found in the “STATUS FROM HOST” region, nested under “Manipulator”. Immediately above `switch (e.MoveEvent)`, insert the following debugging statement:

```
Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : e.MoveEvent=" +
    e.MoveEvent.ToString());
```

- Compile and build the debug application. Ensure Inspect-X is running and run the debug application. Open the Output/Intermediate window in Visual Studio to monitor the debug output.
- Move the manipulator in different ways from Inspect-X (for example, homing, joystick control, position go), and monitor the debug output. Try to understand which events are raised, and when.

- Consider what happens when the demanded position of the manipulator is the same as its current position. What events are raised?

  For systems with a rotate axis, what happens if the axis is asked to move to a position that is factor of 360 degrees away?

4.4 Manipulator rotate application

We will now use this knowledge of Manipulator Move events to develop a simple useful application. We want a button on the user interface that causes the manipulator to rotate from its current position by a set amount. The user can specify the amount through a numeric up-down box that is also present on the user interface.

- Create a variable in the 'Application Variables' section that represents the Rotate axis of the manipulator.

```
/// <summary> Define the rotate axis (constant) summary>
const IpcContract.Manipulator.EAxisName mRotateAxis = IpcContract.Manipulator.EAxisName.Rotate;
```

- Decide on the appropriate event(s) that will signal when the manipulator has started and stopped moving. Create appropriately named Boolean flags for these in the 'Application Variables' section.

  Set the flag(s) to be true at appropriate place(s) in the `EventHandlerManipulatorMoveEvent` callback function when the relevant event(s) are raised.
- Create an appropriately named `private decimal` variable in the 'Application Variables' section to store the rotate amount, with an initial value of 10.
- On the Designer for the UserForm, add a numeric up-down box from the Toolbox. Resize it, give it an appropriate name and set its initial value to also be 10. Add a label beside the box to tell the user what variable it is for.
- Double-clicking the numeric up-down box creates a function that is called every time the user updates the value in the box. Within this function, set the value of our global rotate amount to the value of the numeric up-down box. The line of code will be similar to below, but differing due to whichever names we choose.

```
mRotateAmount = NumericUpDown_RotateAmount.Value;
```

- Create a button on the Designer for the user interface, and change its properties as appropriate. Pressing this button will initiate our move routine.
- Construct a `private void ManipulatorMove()` function that will run when the button on the user interface is pressed. The routine should first check to see whether the application has connected to Inspect-X. In addition to initialising any necessary variables, the routine should then look up the current manipulator position of the rotate axis, set a new target position for the rotate axis, instruct the rotate axis to move and wait until the manipulator has finished moving.

The functions that are available in each channels can be found by going to `mChannels.<ChannelName>`, and looking at its member functions. For example, in the manipulator channel under the Axis member, we can find the `Position` function to look up the position of a particular axis:

```
mChannels.Manipulator.Axis.Position(mRotateAxis);
```

- Create a new thread variable. Add code to run the `ManipulatorMove` function on a new thread when the button is clicked.
- Add appropriate lines to the `ManipulatorMove` function that disables everything on the user interface after it has been clicked, and re-enables it once the our Manipulator Move routine has finished.
- Compile, build and test the debug application to ensure that it works correctly.

4.5 Sample answer

A sample answer for this tutorial can be found at the end of this Guide in ***Sample code - tutorial 2*** (on page 42).

4.6 Extensions

1. Modify the program to move the X axis by a specified amount instead the rotate axis. What other precautions need to be taken with the X or Y axis?
2. Add an extra numeric up-down button to the user interface so that travel distances for two separate axes can be set. Modify the `ManipulatorMove` routine so that both axes move together.
3. Add an extra button to the user interface that will home all of the axes.

5 Tutorial 3: Acquiring an image

Continuing our exploration of the IPC interfaces, we will write a simple application that will acquire an image from the detector using Inspect-X.

5.1 Set up folders, solution and project

- Following ***Creating and setting up the Visual Studio project*** (on page 5), ***The front-end user form and back-end code*** (on page 10) and ***Set up folders, solution and project*** (on page 27) , create a new solution from a copy of the IPCTemplate. Rename properties as appropriate.
- Test to ensure that the debug application compiles and builds successfully.

5.2 Establish channel connections and identify appropriate events

- Set the initial flags to allow access to the Application and Image Processing channels only.
- Remove all of the events associated with the other channels from the `ChannelsAttach` and `ChannelsDetach` functions.
- Create an `mApplicationState` variable, and then assign the `ChannelsAttach` and `ChannelsDetach` functions to the “Load” and “FormClosing” events of the main user form.
- Place `Debug.Print` statements within the one non-heartbeat callback function. Build, compile and run the debug application.
- From within Inspect-X, perform a variety of Image Processing tasks and use the debug output to observe which events are raised.
- Decide whether there are any unnecessary event-handlers and remove these from the `ChannelsAttach` and `ChannelsDetach` functions.

5.3 Image-processing routine

Let us now create a routine that will average a user-defined number of images, and save the averaged image to disk. The user should be able to specify the number of images to average along with the file name.

- Create Boolean flags for the important events, and set them to be true when the appropriate events are raised.
- Create an integer variable with an initial value of 1 for the number of images to average. Add a numeric up-down box to the user form in the designer, and update the value of the integer variable when the numeric up-down box is changed.
- Create a string constant (`const string`) that stores the directory in which the final images will be saved. Ensure that the literal string value that this constant is given represents the path of a directory that exists.
- Create a `private string` variable that will store the filename, and give it an initial value. Add a text box to the user form, and update the value of the filename variable when the text box text is changed.
- Construct a `private void ImageAverageSave()` function for averaging and saving the image.

  The function should first check that the appropriate channels are connected. Next, it should create a variable for the actual image file path, concatenating the directory, filename and image extension. The routine should then average the user-defined number of images and wait until averaging has completed. The image should be saved, waiting until the save has completed successfully.
- Create a new thread variable. On the Designer for the user interface, insert a button that will run the `ImageAverageSave` routine on a new thread when it is clicked.

- Add code to the `ImageAverageSave` routine to disable and enable the button where appropriate.

5.4 Sample answer

A sample answer for this tutorial can be found at the end of this Guide in ***Sample code - tutorial 3*** (on page 48).

5.5 Extensions

1. Modify the program to switch into the live imaging mode and capture the live image rather than take an average of images.
2. Correct the image before saving. For example, one method of correcting the raw image would be to subtract the background black field, divide by the background white field before finally multiplying by a target value, say 60,000.

6 IPC - Expanding the potential of Nikon X-ray CT systems

This Quick Start Guide has introduced you to the basics of IPC programming provided in Inspect-X, and has shown how to write very simple applications on the X-ray, manipulator and image processing sub-systems. These can be combined to build up more complex applications, that could do almost anything that you could envisage. Let us briefly discuss a few more examples, out of the unlimited scope that is possible.

6.1 Time-lapse imaging

The imaging acquisition and save routines studied in ***Tutorial 3: Acquiring an image*** (on page 31) could be combined in a loop with `Thread.Sleep()` command to create a routine that acquires radiographs in a time-lapsed manner.

6.2 Batch 3D scans

This Guide has utilised the application, X-ray, manipulator and image processing communication channels, but there are a number of other channels, which provide other useful functionality. For instance, the CT 3D Scan communication channel allows us to start a 3D CT scan from a saved profile.

```
// Get profile list
List<string> profiles = mChannels.CT3DScan.ProfileList();


if (profiles.Count > 0)
{
        // Sample Info
        System.Collections.Generic.Dictionary<string, Object> sampleInfo =
        new System.Collections.Generic.Dictionary<string, Object>();


        sampleInfo.Add("Dataset name", "Test");


        // Response variable
        Inspect_X_Definitions.CTResponse response;
        Debug.Print("profiles[0]=" + profiles[0].ToString());
        // Start run
        response = mChannels.CT3DScan.Run(profiles[0], sampleInfo);
}
```

Several different profiles could be predefined, for example high and low resolutions scans of the same sample, with the above code adapted to perform a batch scan of each profile in turn. The scans can also be automatically reconstructed, with Inspect-X returning status updates on the progress of the reconstruction.

6.3 Integration with 3rd party software

One of the advantages of developing programs through IPC is the ability to integrate and interact with other software separate to Inspect-X, for example VG-Studio or Microsoft Word. After acquiring and reconstructing the data, VG-Studio could be used to analyse whether measurements are within certain tolerances: Are there an acceptable number of defects? Are the walls of the sample within a given tolerated thickness? These results could then be neatly presented in a report using Microsoft Word. In this way, a completely automated acquisition, reconstruction, analysis and reporting pipeline could be created.

6.4 Integration with 3rd party hardware

Similarly, IPC can be used to integrate with hardware that is separate from the Nikon X-ray CT system. For example, the alarm callback functions could be linked to control room software, or the results reported from automated reconstruction and analysis could be linked directly to the production line, with production paused if tolerances fall below a given value.

6.5 Limitless potential...!

The IPC interface thus expands and extends the potential functionality of the Nikon X-ray CT system, allowing us to perform and automate many complicated tasks that are not possible simply through the Inspect-X software. The full range of IPC commands can be found in the IPC Programming Manual which accompanies the Inspect-X installation.

This Guide has simple scratched the surface of what is possible, but the possibilities and scope for using IPC are endless. In short, it allows the Nikon X-ray CT system to be used in an unlimited manner!

7 Appendix: Sample code

7.1 Tutorial 1: UserForm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
//using System.Linq;
using System.Text;
using System.Windows.Forms;


using IpcContractClientInterface;
using AppLog = IpcUtil.Logging;
using System.Globalization;
using System.Threading;
using System.Diagnostics;
using System.IO;

namespace Tutorial1_XraysOnOff
{
    public partial class UserForm : Form
    {
        /// <summary>Are we in design modesummary>
        protected bool mDesignMode { get; private set; }

        #region Standard IPC Variables

        /// <summary>This ensures consistent read and write culturesummary>
        private NumberFormatInfo mNFI = new CultureInfo("en-GB", true).NumberFormat; // Force UN English culture

        /// <summary>Collection of all IPC channels, this object always exists.summary>
        private Channels mChannels = new Channels();

        #endregion Standard IPC Variables

        #region Application Variables

        /// <summary> Status of the application summary>
        private Channels.EConnectionState mApplicationState;

        /// <summary> Flag for X-ray stability summary>
        private Boolean mXraysStable = false;

        /// <summary> Thread for X-ray Routine summary>
        private Thread mXrayRoutineThread = null;

        /// <summary> Entire Xray Status (for bug correction) summary>
        private IpcContract.XRay.EntireStatus mXrayEntireStatus;

        /// <summary> Generation status summary>
        private IpcContract.XRay.GenerationStatus.EXRayGenerationState mXrayGenerationStatus;

        /// <summary> Stability event counter summary>
        private int mXraysStabilityCounter = 0;

        #endregion Application Variables

        public UserForm()
        {
            try
            {
                mDesignMode = (LicenseManager.CurrentContext.UsageMode == LicenseUsageMode.Designtime);
                InitializeComponent();
                if (!mDesignMode)
                {
                    // Tell normal logging who the parent window is.
                    AppLog.SetParentWindow = this;
                    AppLog.TraceInfo = true;
                    AppLog.TraceDebug = true;

                    mChannels = new Channels();
                    // Enable the channels that will be controlled by this application.
                    // For the generic IPC client this is all of them!
                    // This just sets flags, it does not actually open the channels.
                    mChannels.AccessApplication = true;
                    mChannels.AccessXray = true;
                    mChannels.AccessManipulator = false;
                    mChannels.AccessImageProcessing = false;
                    mChannels.AccessInspection = false;
                    mChannels.AccessInspection2D = false;
                    mChannels.AccessCT3DScan = false;
                    mChannels.AccessCT2DScan = false;
                }
            }
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        #region Channel connections

        /// <summary>Attach to channel and connect any event handlerssummary>
        /// <returns>Connection statusreturns>
        private Channels.EConnectionState ChannelsAttach()
        {
            try
            {
                if (mChannels != null)
                {
                    Channels.EConnectionState State = mChannels.Connect();
                    if (State == Channels.EConnectionState.Connected)  // Open channels
                    {
                        // Attach event handlers (as required)

                        if (mChannels.Application != null)
                        {
                            mChannels.Application.mEventSubscriptionHeartbeat.Event +=
                                new EventHandler <CommunicationsChannel_Application.EventArgsHeartbeat
                                >(EventHandlerHeartbeatApp);
                        }

                        if (mChannels.Xray != null)
                        {
                            mChannels.Xray.mEventSubscriptionHeartbeat.Event +=
                                new
                                EventHandler<CommunicationsChannel_XRay.EventArgsHeartbeat>(EventHandlerHeartbeatX
                                Ray);
                            mChannels.Xray.mEventSubscriptionEntireStatus.Event +=
                                new EventHandler <CommunicationsChannel_XRay.EventArgsXRayEntireStatus>
                                (EventHandlerXRayEntireStatus);
                        }

                    }
                    return State;
                }
            }
            catch (Exception ex) { AppLog.LogException(ex); }
            return Channels.EConnectionState.Error;
        }

        /// <summary>Detach channel and disconnect any event handlerssummary>
        /// <returns>true if OKreturns>
        private bool ChannelsDetach()
        {
            try
            {
                if (mChannels != null)
                {
                    // Detach event handlers

                    if (mChannels.Application != null)
                    {
                        mChannels.Application.mEventSubscriptionHeartbeat.Event -=
                            new EventHandler <CommunicationsChannel_Application.EventArgsHeartbeat>
                                  (EventHandlerHeartbeatApp);
                    }

                    if (mChannels.Xray != null)
                    {
                        mChannels.Xray.mEventSubscriptionHeartbeat.Event -=
                            new EventHandler <CommunicationsChannel_XRay.EventArgsHeartbeat>
                                  (EventHandlerHeartbeatXRay);
                        mChannels.Xray.mEventSubscriptionEntireStatus.Event -=
                            new EventHandler <CommunicationsChannel_XRay.EventArgsXRayEntireStatus>
                                  (EventHandlerXRayEntireStatus);
                    }

                    Thread.Sleep(100); // A breather for events to finish!
                    return mChannels.Disconnect(); // Close channels
                }
            }
            catch (Exception ex) { AppLog.LogException(ex); }
            return false;
        }

        #endregion Channel connections

        #region Heartbeat from host

        void EventHandlerHeartbeatApp(object aSender, CommunicationsChannel_Application.EventArgsHeartbeat e)
        {
            try
            {
                if (mChannels == null || mChannels.Application == null)
                    return;
                if (this.InvokeRequired)
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatApp(aSender, e); });
                else
                {
                    //your code goes here....
                }
            }
            catch (ObjectDisposedException) { } // ignore
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        void EventHandlerHeartbeatXRay(object aSender, CommunicationsChannel_XRay.EventArgsHeartbeat e)
        {
            try
            {
                if (mChannels == null || mChannels.Xray == null)
                    return;
                if (this.InvokeRequired)
```

```
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatXRay(aSender, e); });
                else
                {
                    //your code goes here....
                }
            }
            catch (ObjectDisposedException) { } // ignore
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        void EventHandlerHeartbeatMan(object aSender, CommunicationsChannel_Manipulator.EventArgsHeartbeat e)
        {
            try
            {
                if (mChannels == null || mChannels.Manipulator == null)
                    return;
                if (this.InvokeRequired)
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatMan(aSender, e); });
                else
                {
                    //your code goes here....
                }
            }
            catch (ObjectDisposedException) { } // ignore
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        void EventHandlerHeartbeatIP(object aSender, CommunicationsChannel_ImageProcessing.EventArgsHeartbeat e)
        {
            try
            {
                if (mChannels == null || mChannels.ImageProcessing == null)
                    return;
                if (this.InvokeRequired)
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatIP(aSender, e); });
                else
                {
                    //your code goes here...
                }
            }
            catch (ObjectDisposedException) { } // ignore
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        void EventHandlerHeartbeatInspection(object aSender, CommunicationsChannel_Inspection.EventArgsHeartbeat
e)
        {
            try
            {
                if (mChannels == null || mChannels.Inspection == null)
                    return;
                if (this.InvokeRequired)
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatInspection(aSender, e); });
                else
                {
                    //your code goes here....
                }
            }
            catch (ObjectDisposedException) { } // ignore
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        void EventHandlerHeartbeatInspection2D(object aSender,
               CommunicationsChannel_Inspection2D.EventArgsHeartbeat e)
        {
            try
            {
                if (mChannels == null || mChannels.Inspection2D == null)
                    return;
                if (this.InvokeRequired)
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatInspection2D(aSender, e); });
                else
                {
                    //your code goes here....
                }
            }
            catch (ObjectDisposedException) { } // ignore
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        void EventHandlerHeartbeatCT3DScan(object aSender, CommunicationsChannel_CT3DScan.EventArgsHeartbeat e)
        {
            try
            {
                if (mChannels == null || mChannels.CT3DScan == null)
                    return;
                if (this.InvokeRequired)
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatCT3DScan(aSender, e); });
                else
                {
                    //your code goes here....
                }
            }
            catch (ObjectDisposedException) { } // ignore
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        void EventHandlerHeartbeatCT2DScan(object aSender, CommunicationsChannel_CT2DScan.EventArgsHeartbeat e)
        {
            try
            {
                if (mChannels == null || mChannels.CT2DScan == null)
                    return;
                if (this.InvokeRequired)
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatCT2DScan(aSender, e); });
                else
                {
                    //your code goes here....
                }
            }
            catch (ObjectDisposedException) { } // ignore
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        #endregion Heartbeat from host

        #region STATUS FROM HOST

        #region XRay

        void EventHandlerXRayEntireStatus(object aSender, CommunicationsChannel_XRay.EventArgsXRayEntireStatus e)
        {
            try
            {
                if (mChannels == null || mChannels.Xray == null)
                    return;
                if (this.InvokeRequired)
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerXRayEntireStatus(aSender, e); }); //
                        Make it non blocking if called form this UI thread
                else
                {
                    if (e.EntireStatus != null)
                    {

                        Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " :
                        e.EntireStatus.XRaysStatus.GenerationStatus.State=" +
                        e.EntireStatus.XRaysStatus.GenerationStatus.State.ToString());

                        switch (e.EntireStatus.XRaysStatus.GenerationStatus.State)
                        {
                            case IpcContract.XRay.GenerationStatus.EXRayGenerationState.Success:
                                // Set mXraysStable Flag to true indicating stability has been reached
                                mXraysStable = true;
                                break;
                            case IpcContract.XRay.GenerationStatus.EXRayGenerationState.WaitingForStability:
                                // Increment stability counter;
                                mXraysStabilityCounter++;

                                // If stability counter is greater than 1 then must manually check update X-ray
                                  Entire status
                                if (mXraysStabilityCounter > 1)
                                {
                                    // Manual loop to update X-ray Entire Status until "Success"

                                    Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Manually
                                           checking for stability");

                                    do
                                    {
                                        // First sleep for a small amount of time to allow status updates
                                        Thread.Sleep(100);
                                        // Then get a updated X-ray Entire Status
                                        mXrayEntireStatus = mChannels.Xray.GetXRayEntireStatus();
                                        // Find generation part of Entire Status
                                        mXrayGenerationStatus =
                                        mXrayEntireStatus.XRaysStatus.GenerationStatus.State;

                                        Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " :
                                        mXrayGenerationStatus=" + mXrayGenerationStatus.ToString());
                                    }
                                    while (mXrayGenerationStatus !=
                                    IpcContract.XRay.GenerationStatus.EXRayGenerationState.Success);

                                    Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Manually
                                    found stable X-rays- Proceed");

                                    // Once "Success" obtained then set mXraysStable flag to true
                                    mXraysStable = true;

                                    // Reset stability counter
                                    mXraysStabilityCounter = 0;
                                }
                                break;
                            case IpcContract.XRay.GenerationStatus.EXRayGenerationState.NoXRayController:
                                // Your code goes here...
                                break;
                            case IpcContract.XRay.GenerationStatus.EXRayGenerationState.StabilityTimeout:
                                // Your code goes here...
                                break;
                            case IpcContract.XRay.GenerationStatus.EXRayGenerationState.StabilityXRays:
                                // Your code goes here...
                                break;
                            case IpcContract.XRay.GenerationStatus.EXRayGenerationState.SwitchedOff:
                                // Set reset flag when X-rays are turned off
                                mXraysStable = false;
                                break;
                        }
                        Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : mXraysStable=" +
                         mXraysStable.ToString());
                    }
                }
            }
            catch (Exception ex) { AppLog.LogException(ex); }
        }


        #endregion

        #endregion Status from host


        #region Form Functions

        private void UserForm_Load(object sender, EventArgs e)
        {
            try
            {
                // Attach channels
                mApplicationState = ChannelsAttach();

                if (mApplicationState == Channels.EConnectionState.Connected)
                    Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Connected to Inspect-X");
                else
                    Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Problem in connecting to
                         Inspect-X");
            }
            catch (Exception ex) { AppLog.LogException(ex); }
        }



        private void UserForm_FormClosing(object sender, FormClosingEventArgs e)
        {
            try
            {
                // Detach channels
                ChannelsDetach();

                Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Disconnected from Inspect-X");
            }
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        private void btn_Start_Click(object sender, EventArgs e)
        {
            // Assign the XrayRoutine to the mXrayRoutineThread
            mXrayRoutineThread = new Thread(XrayRoutine);

            // Start the thread
            mXrayRoutineThread.Start();
        }

        #endregion Form Functions

        #region X-ray functions

        private void XrayRoutine()
        {
            // If ApplicationState is not connected then immediately exit the routine
            if (mApplicationState != Channels.EConnectionState.Connected)
                return;

            // For safety, disable the Start button
            this.Invoke((MethodInvoker)delegate { btn_Start.Enabled = false; });

            // Set mXraysStable flag to false
            mXraysStable = false;

            // Turn the X-rays on.
            mChannels.Xray.XRays.GenerationDemand(true);

            // Wait until X-rays have stabilised
            while (!mXraysStable)
                Thread.Sleep(5);

            // Once stable, wait for a further 5 seconds
            Thread.Sleep(5000);

            // Turn the X-rays off
            mChannels.Xray.XRays.GenerationDemand(false);

            // Wait until X-rays have turned off
            while (mXraysStable)
                Thread.Sleep(5);

            // Re-enable the Start button
            this.Invoke((MethodInvoker)delegate { btn_Start.Enabled = true; });
        }

        #endregion X-ray functions


    }
}
```

7.2 Tutorial 2: UserForm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
//using System.Linq;
using System.Text;
using System.Windows.Forms;


using IpcContractClientInterface;
using AppLog = IpcUtil.Logging;
using System.Globalization;
using System.Threading;
using System.Diagnostics;
using System.IO;

namespace Tutorial2_MovingManipulator
{
   public partial class UserForm : Form
   {
      /// <summary>Are we in design modesummary>
      protected bool mDesignMode { get; private set; }

      #region Standard IPC Variables

      /// <summary>This ensures consistent read and write culturesummary>
    private NumberFormatInfo mNFI = new CultureInfo("en-GB", true).NumberFormat; // Force UN English culture
    /// <summary>Collection of all IPC channels, this object always exists.summary>
      private Channels mChannels = new Channels();

      #endregion Standard IPC Variables

        #region Application Variables

        /// <summary> Status of the application summary>
        private Channels.EConnectionState mApplicationState;

        /// <summary> Define the rotate axis (constant) summary>
        const IpcContract.Manipulator.EAxisName mRotateAxis = IpcContract.Manipulator.EAxisName.Rotate;

        /// <summary> Go signal sent flag summary>
        private bool mManipulatorGoSignalSent = false;
        /// <summary> Go complete flag summary>
        private bool mManipulatorGoComplete = false;

        /// <summary> Rotate amount summary>
        private decimal mRotateAmount = 10;

        /// <summary> Manipulator thread variable summary>
        private Thread mManipulatorThread = null;

        #endregion Application Variables

        public UserForm()
      {
         try
         {
            mDesignMode = (LicenseManager.CurrentContext.UsageMode == LicenseUsageMode.Designtime);
            InitializeComponent();
            if (!mDesignMode)
            {
               // Tell normal logging who the parent window is.
               AppLog.SetParentWindow = this;
               AppLog.TraceInfo = true;
               AppLog.TraceDebug = true;

               mChannels = new Channels();
               // Enable the channels that will be controlled by this application.
               // For the generic IPC client this is all of them!
               // This just sets flags, it does not actually open the channels.
               mChannels.AccessApplication = true;
               mChannels.AccessXray = false;
               mChannels.AccessManipulator = true;
               mChannels.AccessImageProcessing = false;
                    mChannels.AccessInspection = false;
               mChannels.AccessInspection2D = false;
               mChannels.AccessCT3DScan = false;
               mChannels.AccessCT2DScan = false;
            }
         }
         catch (Exception ex) { AppLog.LogException(ex); }
      }

      #region Channel connections

      /// <summary>Attach to channel and connect any event handlerssummary>
      /// <returns>Connection statusreturns>
      private Channels.EConnectionState ChannelsAttach()
      {
         try
         {
            if (mChannels != null)
            {
               Channels.EConnectionState State = mChannels.Connect();
               if (State == Channels.EConnectionState.Connected)  // Open channels
               {
                  // Attach event handlers (as required)

                  if (mChannels.Application != null)
                  {
                     mChannels.Application.mEventSubscriptionHeartbeat.Event +=
                        new EventHandler <CommunicationsChannel_Application.EventArgsHeartbeat>
                        (EventHandlerHeartbeatApp);
                  }


                  if (mChannels.Manipulator != null)
                  {
                     mChannels.Manipulator.mEventSubscriptionHeartbeat.Event +=
                        new EventHandler <CommunicationsChannel_Manipulator.EventArgsHeartbeat>
                        (EventHandlerHeartbeatMan);
                     mChannels.Manipulator.mEventSubscriptionManipulatorMove.Event +=
                        new EventHandler <CommunicationsChannel_Manipulator.EventArgsManipulatorMoveEvent>
                        (EventHandlerManipulatorMoveEvent);
                  }

               }
               return State;
            }
         }
         catch (Exception ex) { AppLog.LogException(ex); }
         return Channels.EConnectionState.Error;
      }

      /// <summary>Detach channel and disconnect any event handlerssummary>
      /// <returns>true if OKreturns>
      private bool ChannelsDetach()
      {
         try
         {
            if (mChannels != null)
            {
               // Detach event handlers

               if (mChannels.Application != null)
               {
                  mChannels.Application.mEventSubscriptionHeartbeat.Event -=
                     new EventHandler <CommunicationsChannel_Application.EventArgsHeartbeat>
                        (EventHandlerHeartbeatApp);
               }


               if (mChannels.Manipulator != null)
               {
                  mChannels.Manipulator.mEventSubscriptionHeartbeat.Event -=
                     new EventHandler <CommunicationsChannel_Manipulator.EventArgsHeartbeat>
                        (EventHandlerHeartbeatMan);
                  mChannels.Manipulator.mEventSubscriptionManipulatorMove.Event -=
                     new EventHandler <CommunicationsChannel_Manipulator.EventArgsManipulatorMoveEvent>
                        (EventHandlerManipulatorMoveEvent);
               }

               Thread.Sleep(100); // A breather for events to finish!
              return mChannels.Disconnect(); // Close channels
            }
         }
         catch (Exception ex) { AppLog.LogException(ex); }
         return false;
      }

```

```
#endregion Channel connections

#region Heartbeat from host

void EventHandlerHeartbeatApp(object aSender, CommunicationsChannel_Application.EventArgsHeartbeat e)
{
   try
   {
      if (mChannels == null || mChannels.Application == null)
         return;
      if (this.InvokeRequired)
         this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatApp(aSender, e); });
      else
      {
         //your code goes here....
      }
   }
   catch (ObjectDisposedException) { } // ignore
   catch (Exception ex) { AppLog.LogException(ex); }
}

void EventHandlerHeartbeatMan(object aSender, CommunicationsChannel_Manipulator.EventArgsHeartbeat e)
{
   try
   {
      if (mChannels == null || mChannels.Manipulator == null)
         return;
      if (this.InvokeRequired)
         this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatMan(aSender, e); });
      else
      {
         //your code goes here....
      }
   }
   catch (ObjectDisposedException) { } // ignore
   catch (Exception ex) { AppLog.LogException(ex); }
}

#endregion Heartbeat from host

#region STATUS FROM HOST

#region Manipulator

void EventHandlerManipulatorMoveEvent(object aSender,
         CommunicationsChannel_Manipulator.EventArgsManipulatorMoveEvent e)
{
   try
   {
      if (mChannels == null || mChannels.Manipulator == null)
         return;
      if (this.InvokeRequired)
         this.BeginInvoke((MethodInvoker)delegate { EventHandlerManipulatorMoveEvent(aSender, e); }); //
         Make it non blocking if called form this UI thread
      else
      {
              Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : e.MoveEvent=" +
                  e.MoveEvent.ToString());

              switch (e.MoveEvent)
         {
         case IpcContract.Manipulator.EMoveEvent.HomingStarted:
            // Your code goes here...
            break;
         case IpcContract.Manipulator.EMoveEvent.HomingCompleted:
            // Your code goes here...
            break;
         case IpcContract.Manipulator.EMoveEvent.ManipulatorStartedMoving:
            // Your code goes here...
            break;
         case IpcContract.Manipulator.EMoveEvent.ManipulatorStoppedMoving:
            // Your code goes here...
            break;
         case IpcContract.Manipulator.EMoveEvent.FilamentChangePositionGoStarted:
            // Your code goes here...
            break;
         case IpcContract.Manipulator.EMoveEvent.GoCompleted:
                  // Set mManipulatorGoComplete flag to be true
                  mManipulatorGoComplete = true;
            break;
         case IpcContract.Manipulator.EMoveEvent.GoStarted:
                  // Set mManipulatorGoSignalSent flag to be true
                  mManipulatorGoSignalSent = true;
            break;
         case IpcContract.Manipulator.EMoveEvent.LoadPositionGoCompleted:
                  // Your code goes here...
            break;
         case IpcContract.Manipulator.EMoveEvent.LoadPositionGoStarted:
            // Your code goes here...
            break;
         case IpcContract.Manipulator.EMoveEvent.Error:
            // Your code goes here...
            break;
         default:
            break;
         }
      }
   }
   catch (Exception ex) { AppLog.LogException(ex); }
}


#endregion



  #endregion Status from host

  #region User functions

  private void UserForm_Load(object sender, EventArgs e)
  {
      try
      {
          // Attach channels
          mApplicationState = ChannelsAttach();

          if (mApplicationState == Channels.EConnectionState.Connected)
              Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Connected to Inspect-X");
          else
              Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Problem in connecting to Inspect-X");
      }
      catch (Exception ex) { AppLog.LogException(ex); }
  }



  private void UserForm_FormClosing(object sender, FormClosingEventArgs e)
  {
      try
      {
```

```
                // Detach channels
                ChannelsDetach();

                Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Disconnected from Inspect-X");
            }
            catch (Exception ex) { AppLog.LogException(ex); }
        }


        private void numericUpDown_RotateAmount_ValueChanged(object sender, EventArgs e)
        {
            mRotateAmount = numericUpDown_RotateAmount.Value;
        }


        private void btn_Start_Click(object sender, EventArgs e)
        {
            // Initialise a new thread for Manipulator move to run on
            mManipulatorThread = new Thread(ManipulatorMove);
            // Start the thread
            mManipulatorThread.Start();
        }


        #endregion User functions


        #region Manipulator routines

        private void ManipulatorMove()
        {
            // If ApplicationState is not connected then immediately exit the routine
            if (mApplicationState != Channels.EConnectionState.Connected)
                return;

            // For safety, disable the Start button
            this.Invoke((MethodInvoker)delegate
            {
                btn_Start.Enabled = false;
                lbl_RotateAmount.Enabled = false;
                numericUpDown_RotateAmount.Enabled = false;
            });

            // Variable for current manipulator position
            float aManipulatorCurrentPosition;

            // Variable for demanded manipulator position
            float aManipulatorDemandedPosition;

            // Set Movement flags to be false
            mManipulatorGoSignalSent = false;
            mManipulatorGoComplete = false;

            // Look up current position
            aManipulatorCurrentPosition = mChannels.Manipulator.Axis.Position(mRotateAxis);

            // Calculate new demanded position
            aManipulatorDemandedPosition = aManipulatorCurrentPosition + (float)mRotateAmount;

            // Set target position of rotate axis to be demanded position
            mChannels.Manipulator.Axis.Target(mRotateAxis, aManipulatorDemandedPosition);

            // Tell manipulator to move
            mChannels.Manipulator.Axis.Go(mRotateAxis);

            // while both movement flags are still not positive, then wait
            while (!mManipulatorGoSignalSent || !mManipulatorGoComplete)
                Thread.Sleep(10);

            // Re-enable the Start button
            this.Invoke((MethodInvoker)delegate
            {
                btn_Start.Enabled = true;
                lbl_RotateAmount.Enabled = true;
                numericUpDown_RotateAmount.Enabled = true;
            });

        }

        #endregion Manipulator routines



    }
}
```

7.3 Tutorial 3: UserForm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
//using System.Linq;
using System.Text;
using System.Windows.Forms;


using IpcContractClientInterface;
using AppLog = IpcUtil.Logging;
using System.Globalization;
using System.Threading;
using System.Diagnostics;
using System.IO;

namespace Tutorial3_AcquiringAnImage
{
    public partial class UserForm : Form
    {
        /// <summary>Are we in design modesummary>
        protected bool mDesignMode { get; private set; }

        #region Standard IPC Variables

        /// <summary>This ensures consistent read and write culturesummary>
        private NumberFormatInfo mNFI = new CultureInfo("en-GB", true).NumberFormat; // Force UN English culture

        /// <summary>Collection of all IPC channels, this object always exists.summary>
        private Channels mChannels = new Channels();

        #endregion Standard IPC Variables

        #region Application Variables

        // Application connection status
        private Channels.EConnectionState mApplicationState;

        // Flag for Average Complete
        private Boolean mImageAverageComplete = false;
        // Flag for Image Save complete
        private Boolean mImageSaveComplete = false;

        // Number of images to average
        private int mNumberImagesToAverage = 1;

        // String constant for Directory
        const string mDirectory = @"C:\Users\User\Pictures";

        // String for filename
        private string mFilename = "untitled";

        // Thread for image average save routine
        private Thread mThreadImageAverageSave = null;

        #endregion Application Variables

        public UserForm()
        {
            try
            {
                mDesignMode = (LicenseManager.CurrentContext.UsageMode == LicenseUsageMode.Designtime);
                InitializeComponent();
                if (!mDesignMode)
                {
                    // Tell normal logging who the parent window is.
                    AppLog.SetParentWindow = this;
                    AppLog.TraceInfo = true;
                    AppLog.TraceDebug = true;

                    mChannels = new Channels();
                    // Enable the channels that will be controlled by this application.
                    // For the generic IPC client this is all of them!
                    // This just sets flags, it does not actually open the channels.
                    mChannels.AccessApplication = true;
                    mChannels.AccessXray = false;
                    mChannels.AccessManipulator = false;
                    mChannels.AccessImageProcessing = true;
                    mChannels.AccessInspection = false;
                    mChannels.AccessInspection2D = false;
                    mChannels.AccessCT3DScan = false;
                    mChannels.AccessCT2DScan = false;
                }
            }
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        #region Channel connections

        /// <summary>Attach to channel and connect any event handlerssummary>
        /// <returns>Connection statusreturns>
        private Channels.EConnectionState ChannelsAttach()
        {
            try
            {
                if (mChannels != null)
                {
                    Channels.EConnectionState State = mChannels.Connect();
                    if (State == Channels.EConnectionState.Connected)  // Open channels
                    {
                        // Attach event handlers (as required)

                        if (mChannels.Application != null)
                        {
                            mChannels.Application.mEventSubscriptionHeartbeat.Event +=
                                new EventHandler <CommunicationsChannel_Application.EventArgsHeartbeat>
                                  (EventHandlerHeartbeatApp);
                        }


                        if (mChannels.ImageProcessing != null)
                        {
                            mChannels.ImageProcessing.mEventSubscriptionHeartbeat.Event +=
                                new EventHandler <CommunicationsChannel_ImageProcessing.EventArgsHeartbeat>
                                  (EventHandlerHeartbeatIP);
                            mChannels.ImageProcessing.mEventSubscriptionImageProcessing.Event +=
                                new EventHandler <CommunicationsChannel_ImageProcessing.EventArgsIPEvent>
                                  (EventHandlerIPEvent);
                        }


                    }
                    return State;
                }
            }
            catch (Exception ex) { AppLog.LogException(ex); }
            return Channels.EConnectionState.Error;
        }

        /// <summary>Detach channel and disconnect any event handlerssummary>
        /// <returns>true if OKreturns>
        private bool ChannelsDetach()
        {
            try
            {
                if (mChannels != null)
                {
                    // Detach event handlers

                    if (mChannels.Application != null)
                    {
                        mChannels.Application.mEventSubscriptionHeartbeat.Event -=
                            new EventHandler <CommunicationsChannel_Application.EventArgsHeartbeat>
                                  (EventHandlerHeartbeatApp);
                    }

                    if (mChannels.ImageProcessing != null)
                    {
                        mChannels.ImageProcessing.mEventSubscriptionHeartbeat.Event -=
                            new EventHandler <CommunicationsChannel_ImageProcessing.EventArgsHeartbeat>
                                  (EventHandlerHeartbeatIP);
                        mChannels.ImageProcessing.mEventSubscriptionImageProcessing.Event -=
                            new EventHandler <CommunicationsChannel_ImageProcessing.EventArgsIPEvent>
                                  (EventHandlerIPEvent);
                    }


                    Thread.Sleep(100); // A breather for events to finish!
                    return mChannels.Disconnect(); // Close channels
                }
            }
            catch (Exception ex) { AppLog.LogException(ex); }
            return false;
        }

        #endregion Channel connections

        #region Heartbeat from host

        void EventHandlerHeartbeatApp(object aSender, CommunicationsChannel_Application.EventArgsHeartbeat e)
        {
            try
            {
                if (mChannels == null || mChannels.Application == null)
                    return;
                if (this.InvokeRequired)
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatApp(aSender, e); });
                else
                {
                    //your code goes here....
                }
            }
            catch (ObjectDisposedException) { } // ignore
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        void EventHandlerHeartbeatIP(object aSender, CommunicationsChannel_ImageProcessing.EventArgsHeartbeat e)
        {
            try
            {
                if (mChannels == null || mChannels.ImageProcessing == null)
                    return;
                if (this.InvokeRequired)
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerHeartbeatIP(aSender, e); });
                else
                {
                    //your code goes here...
                }
            }
            catch (ObjectDisposedException) { } // ignore
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        #endregion Heartbeat from host

        #region STATUS FROM HOST

        #region ImageProcessing

        void EventHandlerIPEvent(object aSender, CommunicationsChannel_ImageProcessing.EventArgsIPEvent e)
        {
            try
            {
                if (mChannels == null || mChannels.ImageProcessing == null)
                    return;
                if (this.InvokeRequired)
                    this.BeginInvoke((MethodInvoker)delegate { EventHandlerIPEvent(aSender, e); }); // Make it non
                        blocking if called form this UI thread
                else
                {

                    Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : e.IPEvent.EventType=" +
                        e.IPEvent.EventType.ToString());

                    switch (e.IPEvent.EventType)
                    {
                        case IpcContract.ImageProcessing.IPEvent.EEventType.Live:
                            // Your code goes here...
                            break;
                        case IpcContract.ImageProcessing.IPEvent.EEventType.Capture:
                            // Your code goes here...
                            break;
                        case IpcContract.ImageProcessing.IPEvent.EEventType.Average:
                            // Your code goes here...
                            break;
                        case IpcContract.ImageProcessing.IPEvent.EEventType.AverageComplete:
                            // flag set to true when image averaging complete
                            mImageAverageComplete = true;
                            break;
                        case IpcContract.ImageProcessing.IPEvent.EEventType.LoadImageComplete:
                            // Your code goes here...
                            break;
                        case IpcContract.ImageProcessing.IPEvent.EEventType.SaveImageComplete:
                            // flag set to true when image saved
                            mImageSaveComplete = true;
                            break;
                        default:
                            // Your code goes here...
                            break;
                    }
                }
            }
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        #endregion

        #endregion Status from host

        #region User functions

        private void UserForm_Load(object sender, EventArgs e)
        {
            try
            {
                // Attach channels
                mApplicationState = ChannelsAttach();

                if (mApplicationState == Channels.EConnectionState.Connected)
                    Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Connected to Inspect-X");
                else
                    Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Problem in connecting to
                        Inspect-X");
            }
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        private void UserForm_FormClosing(object sender, FormClosingEventArgs e)
        {
            try
            {
                // Detach channels
                ChannelsDetach();

                Debug.Print(DateTime.Now.ToString("dd/MM/yyyy H:mm:ss.fff") + " : Disconnected from Inspect-X");
            }
            catch (Exception ex) { AppLog.LogException(ex); }
        }

        private void numericUpDown_NumberImagesToAverage_ValueChanged(object sender, EventArgs e)
        {
            mNumberImagesToAverage = (int) numericUpDown_NumberImagesToAverage.Value;
        }

       private void textBox1_TextChanged(object sender, EventArgs e)
        {
            mFilename = textBox_Filename.Text.ToString();
        }

        private void btn_Start_Click(object sender, EventArgs e)
        {
            // Initialise the thread with the ImageAverageSave routine
            mThreadImageAverageSave = new Thread(ImageAverageSave);
            // Start the thread
            mThreadImageAverageSave.Start();
        }

        #endregion User functions

        #region Image Processing Routines

        private void ImageAverageSave()
        {

            // If ApplicationState is not connected then immediately exit the routine
            if (mApplicationState != Channels.EConnectionState.Connected)
                return;

            // For safety, disable the Start button
            this.Invoke((MethodInvoker)delegate { btn_Start.Enabled = false; });

            // Set filepath
            string aFilepath = mDirectory + "\ " + mFilename + @".tif";

            // Set flags to false
            mImageAverageComplete = false;
            mImageSaveComplete = false;

            // Average set number of images
            mChannels.ImageProcessing.Image.Average(mNumberImagesToAverage, false);

            // Wait until average is complete
            while (!mImageAverageComplete)
                Thread.Sleep(10);

            // Save image
            mChannels.ImageProcessing.Image.SaveAsTiff(aFilepath, false, false, false);

            // Wait until save has completed
            while (!mImageSaveComplete)
                Thread.Sleep(10);

            // Re-enable the Start button
            this.Invoke((MethodInvoker)delegate { btn_Start.Enabled = true; });

        }

        #endregion Image Processing Routines

    }
}
```

NIKON METROLOGY NV
Geldenaaksebaan 329
B-3001 Leuven, Belgium
phone: +32 16 74 01 00 fax: +32 16 74 01 03
sales.nm@nikon.com

NIKON CORPORATION
Shinagawa Intercity Tower C, 2-15-3, Konan, Minato-ku,
Tokyo 108-6290, Japan
phone: +81 3 6433 3701 fax: +81 3 6433 3784
www.nikon.com/products/industrial-metrology

NIKON METROLOGY EUROPE NV
phone: +32 16 74 01 01
sales.europe.nm@nikon.com

NIKON METROLOGY GMBH
phone: +49 6023 91733-0
sales.germany.nm@nikon.com

NIKON METROLOGY SARL
phone: +33 1 60 86 09 76
sales.france.nm@nikon.com

NIKON METROLOGY, INC.
phone: +1 810 2204360
sales.us.nm@nikon.com

NIKON METROLOGY UK LTD.
phone: +44 1332 811349
sales.uk.nm@nikon.com

More offices and resellers at **www.nikonmetrology.com**

NIKON INSTRUMENTS (SHANGHAI) CO. LTD.
phone: +86 21 5836 0050
phone: +86 10 5869 2255 (Beijing office)
phone: +86 20 3882 0550 (Guangzhou office)

NIKON SINGAPORE PTE. LTD.
phone: +65 6559 3618
nsg.industrial-sales@nikon.com

NIKON MALAYSIA SDN. BHD.
phone: +60 3 7809 3609

NIKON INSTRUMENTS KOREA CO. LTD.
phone: +82 2 2186 8400