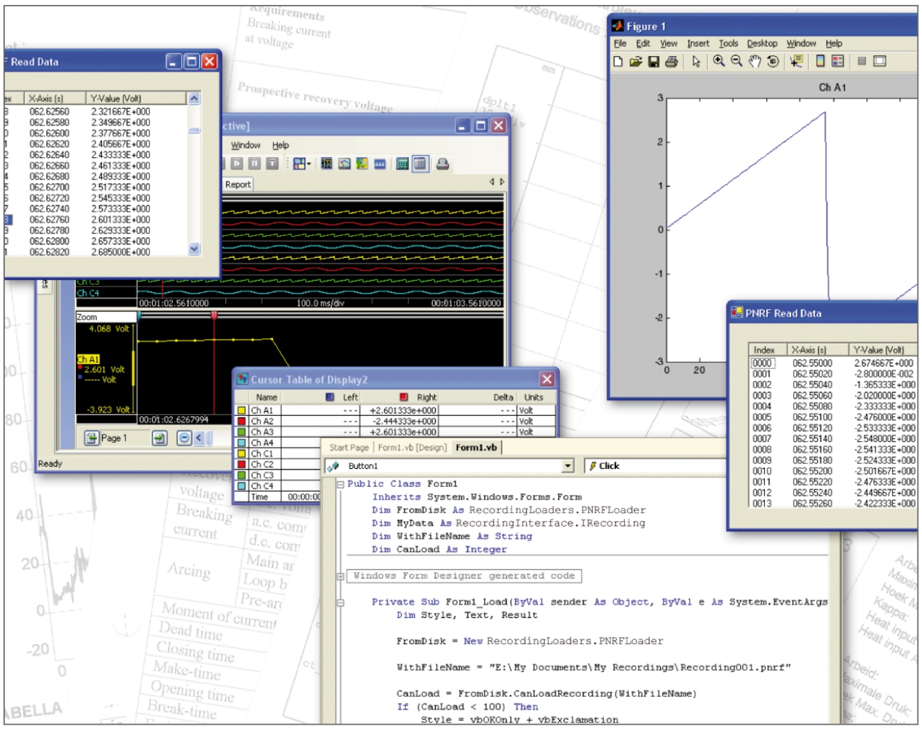
Genesis HIGH SPEED

User Manual

English



X-Axis (s)	Y-Value (Volt)
062.62560	2.321667E+000
062.62580	2.349667E+000
062.62600	2.377667E+000
062.62620	2.405667E+000
062.62640	2.433333E+000
062.62660	2.461333E+000
062.62680	2.489333E+000
062.62700	2.517333E+000
062.62720	2.545333E+000
062.62740	2.573333E+000
062.62760	2.601333E+000
062.62780	2.629333E+000
062.62800	2.657333E+000
062.62820	2.685000E+000

Index	X-Axis (s)	Y-Value (Volt)
0000	062.55000	2.674667E+000
0001	062.55020	-2.800000E-002
0002	062.55040	-1.365333E+000
0003	062.55060	-2.020000E+000
0004	062.55080	-2.333333E+000
0005	062.55100	-2.476000E+000
0006	062.55120	-2.533333E+000
0007	062.55140	-2.548000E+000
0008	062.55160	-2.541333E+000
0009	062.55180	-2.524333E+000
0010	062.55200	-2.501667E+000
0011	062.55220	-2.476333E+000
0012	062.55240	-2.449667E+000
0013	062.55260	-2.422333E+000

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim FromDisk As RecordingLoaders.PNRFLoader
    Dim MyData As RecordingInterface.IRecording
    Dim WithFileName As String
    Dim CanLoad As Integer

 Windows Form Designer generated code

    Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs
        Dim Style, Text, Result

        FromDisk = New RecordingLoaders.PNRFLoader

        WithFileName = "E:\My Documents\My Recordings\Recording001.pnrf"

        CanLoad = FromDisk.CanLoadRecording(WithFileName)
        If (CanLoad < 100) Then
            Style = vbOKOnly + vbExclamation
```

PNRF Reader SDK

Perception

I2697-1.1 en

HBM

Document version 1.1 - July 2009

HBM GmbH
Im Tiefen See 45
64293 Darmstadt
Germany
Tel: +49 6151 80 30
Fax: +49 6151 8039100
Email: info@hbm.com
www.hbm.com/highspeed

Table of Contents

1 Perception PNRF SDK

1.1 General

Introduction

Welcome to the PNRF Software Development Kit (PNRF SDK). This SDK provides the software, documentation and examples required to use the PNRF Application Programmers Interface (API). This document gives you an introduction to the PNRF API. The interface allows you to write external programs that can read data files generated by Perception software in the Perception Native Recording File (PNRF) format.

This document is not intended to be a PNRF API reference. In this document we will give you only an introduction to the API. Once completed you should be able to find your way in the API.

In this document examples are written in Microsoft Visual Basic .NET, using the Microsoft Visual Studio 2005 development environment.

Features

- Read waveform data from any Perception Native Recording File (*.pnrf) file

Installation

The PNRF SDK contains everything you need to install and use the PNRF API. When you received the SDK on a CD you must install the SDK from the CD onto your hard disk; you cannot use the API from the CD.

To install:

1 Start Windows 2000, XP and insert the CD in the CD-Rom drive.
2 In the Windows Task Bar click the Start button, point to and click Run....
3 In the Run dialog type d:\setup (or e:\setup, depending on your CD-ROM drive assignment) in the Open: text input field and click OK.
4 Follow the on-screen instructions.

1.2 SDK Audience

You must be proficient in your programming language and compiler usage in order to write custom PNRF API programs.

The SDK includes sample projects for Microsoft Visual Basic .NET.

The documentation assumes you understand your HBM equipment and basic acquisition terminology.

Understanding acquisition terminology is vital to understanding digital recordings: trigger, sample rate, pre-/post trigger, etc.

1.3 What you can expect

The PNRF Reader SDK gives you access to the proprietary PNRF File format. **The format itself is not documented: you can not retrieve the data or any other information just by studying a file format description.** To use the PNRF Reader SDK and to actually read data from a PNRF file you will need to write your own program. The PNRF Reader SDK provides you the tools needed to get data and information from a PNRF file:

- The required software components (DLL's). These components act as a software layer between the file contents and a programming language. This layer can provide information on what is in the file, not on how it is in the file. Use this layer to request data and information.
- The information on how to use this software layer: this manual. This manual is an introduction to the use of the software layer, not a comprehensive reference. Appendices are supplied for additional information.

Apart from writing a dedicated stand-alone program it is also possible to interface directly to other programs. There are a number of analysis packages that include a programming environment that allows you to include external DLL's and make use of them. Refer to the documentation supplied with your package to find out the possibilities.

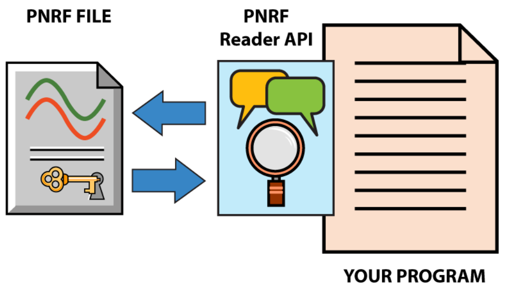


Figure 1.1: Exchange of PNRF file(s)

The PNRF Reader API translates your commands and data requests. Data is returned in ready-to-use blocks.

2 Getting it all to work

2.1 Introduction

In this introduction we will demonstrate how to set up a Visual Basic application that uses the PNRF API. We will also show how to create a simple program that takes information from a PNRF file and displays the result.

Step 1

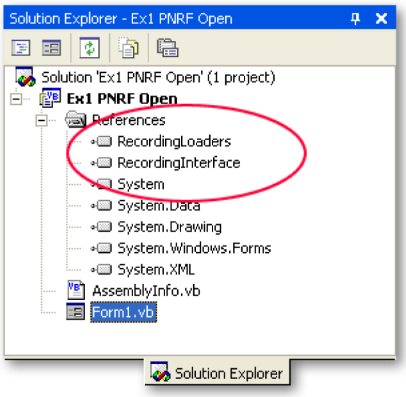
Start the Visual Basic .NET development environment. In the New Project start-up screen select a Visual Basic Project that creates a Windows Application. We will name this project "Ex1 PNRF Open".

The project will be created and you are now presented a form.

Step 2

First we need to include the PNRF API function calls in this project.

To do this select the ***Add Reference...*** command in the ***Project*** menu. You are now presented a list of registered references. Use the ***Browse...*** button to open the Select Component dialog. In this dialog browse to the **C:\Program Files\Common Files\HBM\Components** folder. In this folder select **percPNRFLoader.dll** and **percRecordingInterface.olb** and click Open to add these files to the list of References. Select OK.

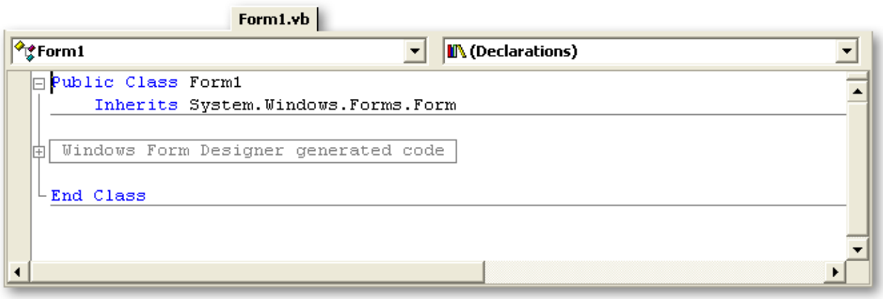


```
Solution Explorer - Ex1 PNRF Open
Solution 'Ex1 PNRF Open' (1 project)
  Ex1 PNRF Open
    References
      RecordingLoaders
      RecordingInterface
      System
      System.Data
      System.Drawing
      System.Windows.Forms
      System.XML
    AssemblyInfo.vb
    Form1.vb
Solution Explorer
```

Figure 2.1: Component dialog

Now the RecordingLoaders and RecordingInterface are added to your list of included references. You can use the **Object Browser** to have a look at the included functionality: select one of the references and expand the tree to see what's inside.

Step 3
Select ***Code*** in the ***View*** menu. Now you can enter the actual programming code. Your view should look like this:



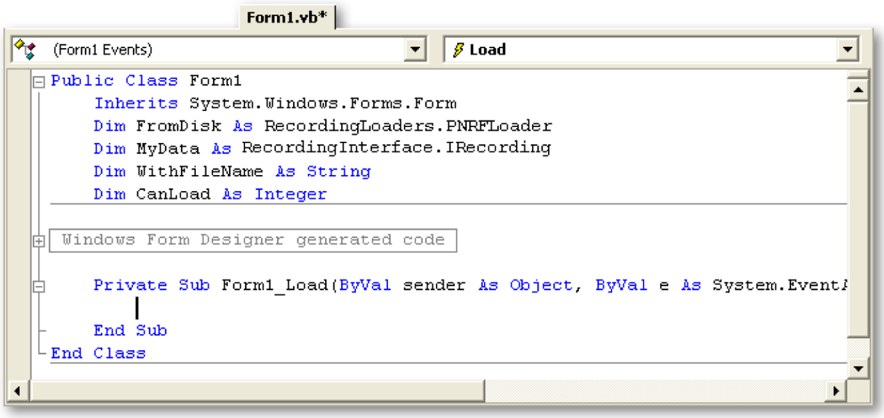
```
Public Class Form1
    Inherits System.Windows.Forms.Form

 Windows Form Designer generated code

End Class
```

Figure 2.2: Visual Basic - Enter programming code

Enter the following code below the line that starts with `Inherits`:

```
Dim FromDisk As
RecordingLoaders.PNRFLoader                   ' Required
Dim MyData As
RecordingInterface.IRecording                 ' Required
Dim WithFileName As String        ' Optionally at this point
Dim CanLoad As Integer            ' Optionally at this point
```

Step 4
We will now enter the code required for the actual program. Select in the **(Form1 Events)** section the **Load** event. Or select the form designer and double-click on the form. Your View will now look like this:



```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim FromDisk As RecordingLoaders.PNRFLoader
    Dim MyData As RecordingInterface.IRecording
    Dim WithFileName As String
    Dim CanLoad As Integer

 Windows Form Designer generated code

    Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.Event
    
    End Sub
End Class
```

Figure 2.3: Visual Basic - Load event

Below the line that starts with `Private Sub` enter the following code:

```
FromDisk = New RecordingLoaders.PNRFLoader
```

At this point the PNRFLoader is available as an object. We now can open a recording file if we know the location and name of the file. Assume:

```
WithFileName = "E:\My Documents\My Recordings
\Recording001.pnrf"
```

then we can enter:

```
MyData = FromDisk.LoadRecording(WithFileName)
```

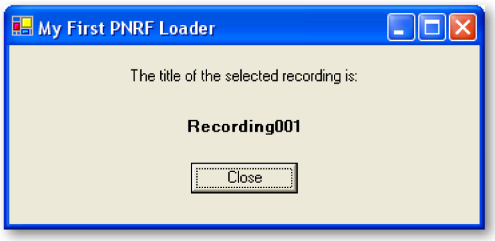
However, to make sure that the loader can read the selected file we will re-arrange and add the following code. This also includes output.

```
CanLoad = FromDisk.CanLoadRecording(WithFileName)
If (CanLoad < 100) Then
   Style = vbOKOnly + vbExclamation
   Text = "This file cannot be loaded."
   Result = MsgBox(Text, Style, "Load Recording")
   Output.Text = "no data"
Else
   MyData = FromDisk.LoadRecording(WithFileName)
   Output.Text = MyData.Title
End If
```

The `FromDisk.CanLoadRecording()` returns a value between 0 and 100. A zero value indicates that the loader can definitely not load the requested file, a value of 100 means the loader can read and interpret the file for sure. A value in between can be interpreted as a percentage of certainty to do a succesful load.

Make sure you have included a label with name Output on your form window.

Step 5
To test your first program (!) select function key F5 to run the program. When all is well the program should run without error messages and the title of the recording should be displayed on your form.

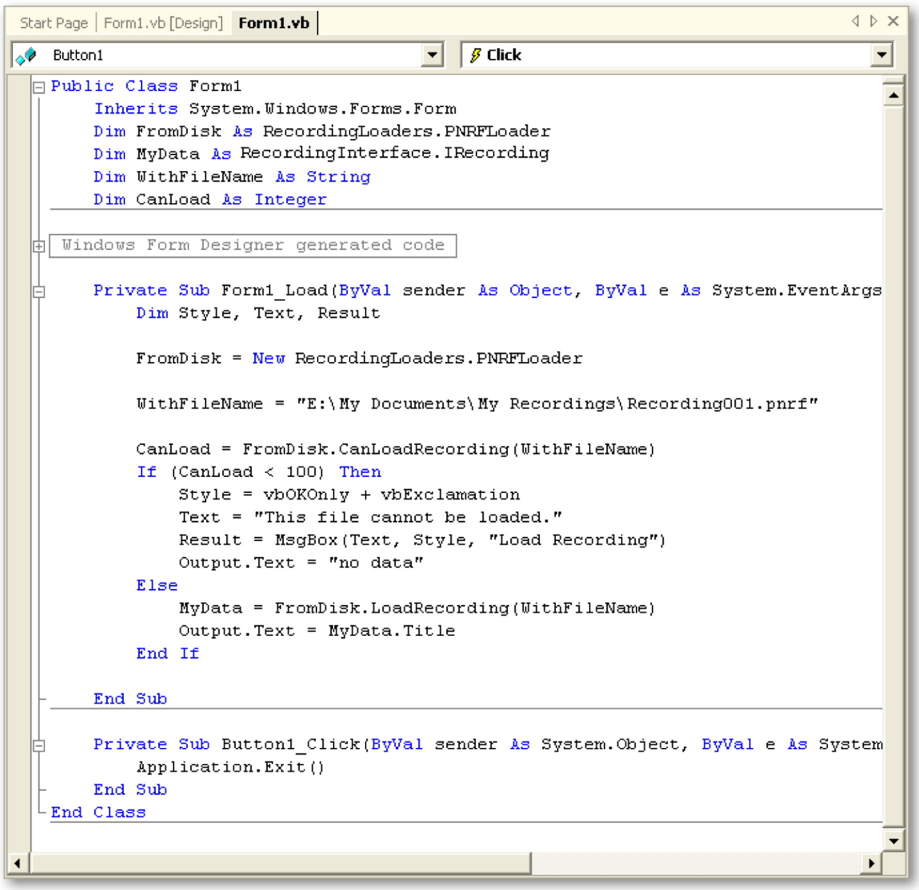

My First PNRF Loader

The title of the selected recording is:

Recording001

Figure 2.4: PNRF Loader dialog

The complete code of this program could look like this (without error trapping):



```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim FromDisk As RecordingLoaders.PNRFLoader
    Dim MyData As RecordingInterface.IRecording
    Dim WithFileName As String
    Dim CanLoad As Integer

 Windows Form Designer generated code 

    Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs
        Dim Style, Text, Result

        FromDisk = New RecordingLoaders.PNRFLoader

        WithFileName = "E:\My Documents\My Recordings\Recording001.pnrf"

        CanLoad = FromDisk.CanLoadRecording(WithFileName)
        If (CanLoad < 100) Then
            Style = vbOKOnly + vbExclamation
            Text = "This file cannot be loaded."
            Result = MsgBox(Text, Style, "Load Recording")
            Output.Text = "no data"
        Else
            MyData = FromDisk.LoadRecording(WithFileName)
            Output.Text = MyData.Title
        End If

    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System
        Application.Exit()
    End Sub
End Class
```

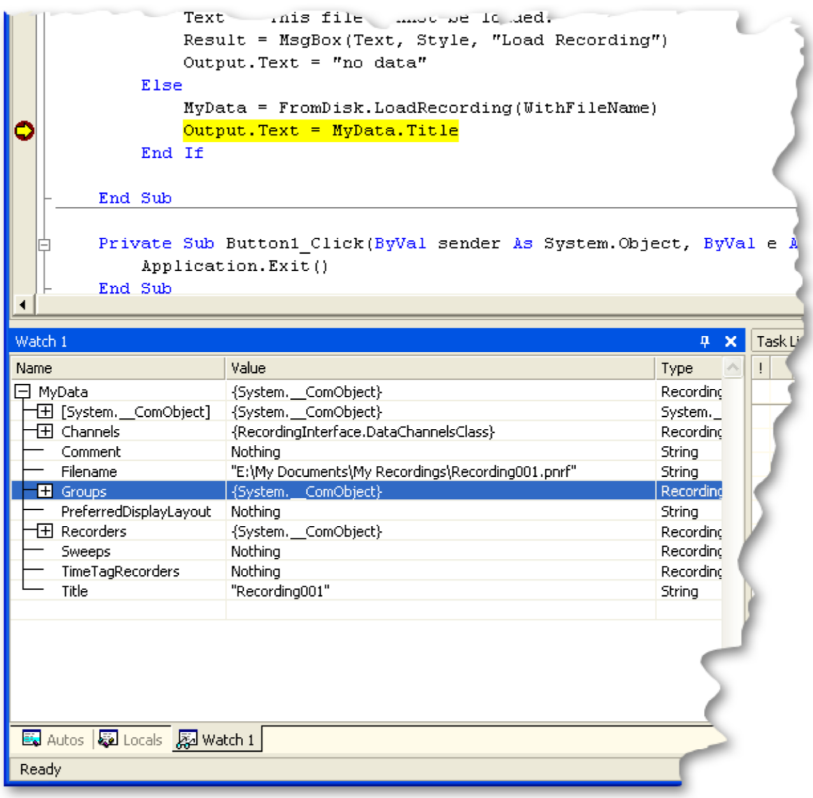
Figure 2.5: Visual Basic - Complete Code (Example)

Within your program you can set breakpoints for debugging purposes: a position in a program at which execution pauses and control returns to you.

When execution pauses you can investigate properties from objects through the Watch Window. This is a helpful tool when developing software with the PNRF API.

You can set a watch and have a look at the various properties as well as the current values of the properties.

Here an example is given in which the MyData object is used as watch expression.



```
            Text [illegible] his file [illegible] be lo[illegible]ed.
            Result = MsgBox(Text, Style, "Load Recording")
            Output.Text = "no data"
        Else
            MyData = FromDisk.LoadRecording(WithFileName)
            Output.Text = MyData.Title
        End If

    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e A
        Application.Exit()
    End Sub
```

Name	Value	Type
MyData	{System.__ComObject}	Recording
[System.__ComObject]	{System.__ComObject}	System._
Channels	{RecordingInterface.DataChannelsClass}	Recording
Comment	Nothing	String
Filename	"E:\My Documents\My Recordings\Recording001.pnrf"	String
Groups	{System.__ComObject}	Recording
PreferredDisplayLayout	Nothing	String
Recorders	{System.__ComObject}	Recording
Sweeps	Nothing	Recording
TimeTagRecorders	Nothing	Recording
Title	"Recording001"	String

Figure 2.6: Various properties - PNRF Recording file

Here you can see the various basic properties of a PNRF Recording File.

In the next chapter we will investigate these properties in more detail.

3 Inside the PNRF

3.1 Introduction

By now you should know how to create a Visual Basic .NET project an how to load a PNRF data file. So, let's go one step further and try to find out what data is actually stored within the file.

In general data (or waveforms) of multiple sources is stored within a single file. All HBM Genesis HighSpeed equipment/software uses the concept of a **recorder**. A recorder can have one or more **channels** that acquire data. All channels within a single recorder have the same timebase settings: sample rate, recording length and trigger parameters. As an extension to this concept, **groups** are used to combine various channels into a logical configuration. This does not alter the arrangement of the channels within the recorders.

Therefore, within a PNRF recording file, information and properties is stored as part of groups, recorders and channels. Within this document we will only describe the use of recorders and channels.

A recording can be a single continuous acquisition, a collection of acquisition sections (sweeps) or a combination of both. We will look into these concepts a little bit later in this document.

3.2 Search for recorders and channels

In this chapter we will demonstrate how recorders and channels are organized within the PNRF file and how we can have a look at properties of these items.

Create a dialog

In the example we will search and display the names of the recorders and the channels in the file as well as the type of channels. A "next" command button will be used to step through the various items.

Create a dialog that looks like this in designer mode:

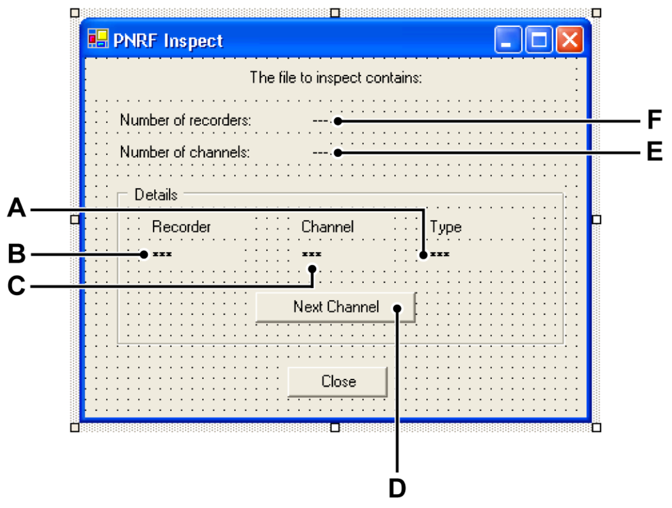


Figure 3.1: PNRF Inspect dialog

A	Type
B	RecName
C	ChName
D	NxtButton
E	NOFChannelsCnt
F	NOFRecordersCnt

Write the code

First we will start with the declaration of various global variables. Add these to the ones we have defined in the previous chapter.

```
Dim MaxRecorders As Integer             ' number of
                                       recorders in file
```

```
Dim MaxChannels As Integer                 ' number of
                                           channels in file
Dim CurrentRecorder As Integer             ' recorder to
                                           inspect
Dim CurrentChannel As Integer              ' channel to
                                           inspect
Dim MaxChannelsInCurrentRecorder As        ' number of
Integer                                    channels
                                           ' in selected
                                           recorder to inspect
```

Proceed as we have done in the previous chapter to open the file. If the file cannot be opened set:

```
MaxRecorders = 0
MaxChannels = 0
NOFRecordersCnt.Text = MaxRecorders
NOFChannelsCnt.Text = MaxChannels
```

When the file can be opened, load the file and retrieve the number of recorders and the number of channels and display this information:

```
MaxRecorders = MyData.Recorders.Count
MaxChannels = MyData.Channels.Count
NOFRecordersCnt.Text = MaxRecorders
NOFChannelsCnt.Text = MaxChannels
```

If you want your code to be bullet proof, add a test to verify if `MaxChannels > 0`. In rare circumstances files can be created without data. This is noted by `Channels.Count = 0`.

The actual retrieval of the information is done in the `NxtButton_Click` routine. However, before we can start we need to initialise some stuff:

```
CurrentRecorder = 1
CurrentChannel = 1
If ((MaxRecorders = 1) And (MaxChannels = 1)) Then
   NxtButton.Enabled = False
Else
   NxtButton.Enabled = True
End If
RecName.Text = MyData.Recorders(1).Name
ChName.Text = MyData.Recorders(1).Channels(1).Name
If (MyData.Recorders(1).Channels(1).ChannelType =
RecordingInterface.DataChannelType.DataChannelType
_Analog)
Then
   Type.Text = "Analog"
Else
   Type.Text = "Digital"
End If
```

First we set the current recorder and channel to "1". When there is only one recorder with one channel, the `NxtButton` is disabled.

Now we can fetch the recorder and channel name. The recorder has two 'names': **Name** and **PhysicalName**. The PhysicalName is the name given by the system and typically reflects the physical position of the recorder within an acquisition mainframe. The Name is set by a user and defaults to the PhysicalName.

Finally the type of the channel is determined. Currently this can be either **analog** or **digital**.

The `NxtButton_Click` routine can have the following code:

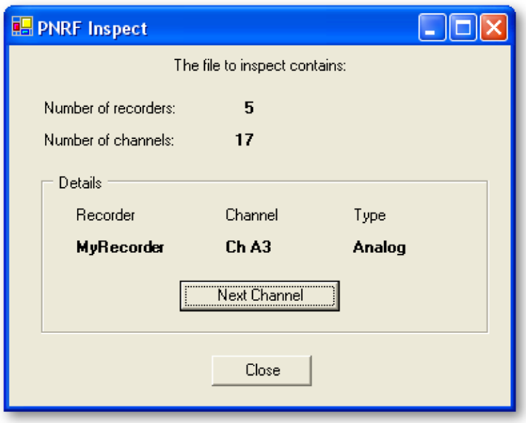
```
MaxChannelsInCurrentRecorder =
MyData.Recorders(CurrentRecorder).
Channels.Count
CurrentChannel = CurrentChannel + 1
If (CurrentChannel > MaxChannelsInCurrentRecorder) Then
   CurrentChannel = 1
   CurrentRecorder = CurrentRecorder + 1
   If (CurrentRecorder > MaxRecorders) Then
      CurrentRecorder = 1
   End If
End If
```

```
RecName.Text = MyData.Recorders(CurrentRecorder).Name
ChName.Text = MyData.Recorders(CurrentRecorder).
Channels(CurrentChannel).Name
If (MyData.Recorders(CurrentRecorder).Channels
(CurrentChannel).
ChannelType = RecordingInterface.DataChannelType.
DataChannelType_Analog) Then
   Type.Text = "Analog"
Else
   Type.Text = "Digital"
End If
```

The second part of this routine deals with the display of the results of the selected recorder and channel as we have seen in the initialization section.

The first part of the routine is used to step through the recorders and channels.

When all is OK you can run the program and the final result should look like this:


PNRF Inspect

The file to inspect contains:

Number of recorders: 5

Number of channels: 17

Details

Recorder	Channel	Type
MyRecorder	Ch A3	Analog

Figure 3.2: PNRF Inspect dialog (Result)

4 Reading Waveform Data

4.1 Introduction

Within the PNRF data files waveform data is stored on a per-channel basis, i.e. within the stored data of a channel all information is available to reconstruct the complete waveform. Although channels are bundled in recorders and groups, no additional information is necessary to reconstruct the original data. The fact that channels belong to a recorder, however, can be used to simplify programming. All channels within a single recorder have the same timebase settings: sample rate, recording length and trigger parameters.

There are many ways to retrieve data from a PNRF file. In this chapter we will concentrate on one type only: the retrieval of raw data (1:1) in floating point format. Other types will be mentioned briefly when applicable.

4.2 A word on segments

An important concept within the PNRF file is the concept of a **segment**: a portion of the data that spans a time interval in which the timebase (x-axis information) as well as the amplifier (y-axis information) remain stable, i.e. there are no changes. A segment is a self-contained piece of recorded waveform data.

Within a recording there can be 0 (zero), 1 or more segments, Zero being no data. To retrieve data from a recording, you need to specify the start and end time of the data that you want to retrieve. The relevant number of segments will be returned as we will demonstrate in this chapter.

4.3 Before we start

Before we start we will introduce a Microsoft Visual Basic .NET feature that will simplify the actual programming as well as the resulting code: namespaces.

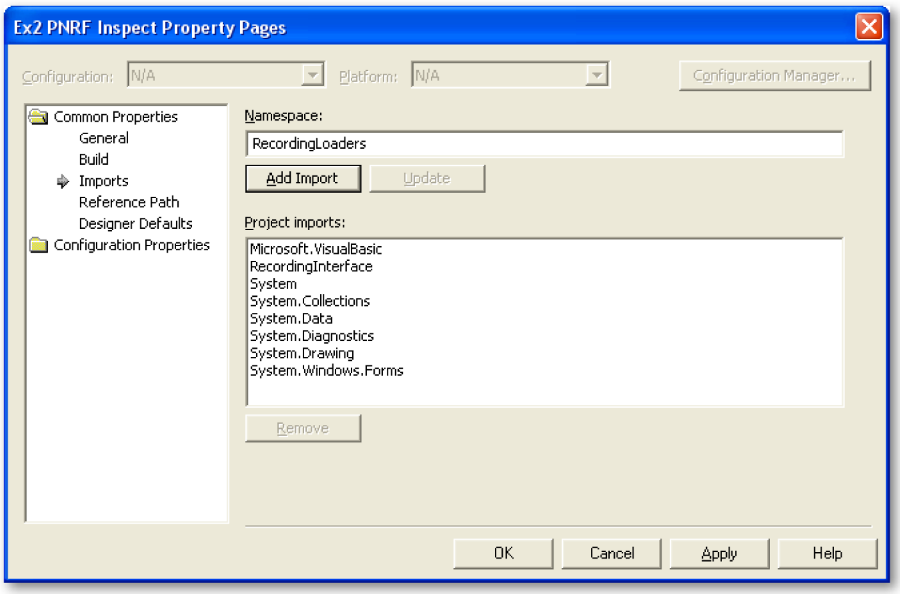
So far we have used fully qualified names for our coding. Fully qualified names are object references that are prefixed with the name of the namespace where the object is defined. This happens when we create a reference to the PNRF classes (by choosing Add Reference from the Project menu) and then use the fully qualified name for the object in our code. E.g.:

```
RecordingInterface.DataChannelType.DataChannelType_Analog
```

Fully qualified names prevent naming conflicts because the compiler can always determine which object is being used. However, the names themselves can get long and cumbersome. To get around this, we use namespaces that we define in the project properties.

To do this proceed as follows:

1. In your design environment select your project in the solution explorer.
2. In the menu select Project > Properties
3. In the dialog that comes up select in the left-hand column: Common Properties > Imports
4. In the Namepace text field enter **RecordingInterface** and click Add Import.
5. Do the same for **RecordingLoaders**
6. When done click OK.



Namespace:
RecordingLoaders

Project imports:
Microsoft.VisualBasic
RecordingInterface
System
System.Collections
System.Data
System.Diagnostics
System.Drawing
System.Windows.Forms

Figure 4.1: PNRF Inspect Property Pages dialog

4.4 Write the code

Declarations
Before we start: make sure you have included the **RecordingLoaders** in your list of Project > References as well as the **RecordingInterface** (see chapter “Getting it all to work” on page 10 for details). Also import both namespaces as decribed earlier.

Add the following lines to the public declarations:

```
Dim FromDisk As PNRFLoader     ' Required
Dim MyData As IRecording       ' Required
Dim WithFileName As String     ' Optionally at this point
Dim CanLoad As Integer         ' Optionally at this point
Dim MySource As IDataSrc       ' Recording data source
Dim Result As Object           ' Intermediate result
Dim Segments As                ' The data segments
IDataSegments
```

Note *Since we are using imported namespaces, the complete prefixes can be ommitted. We now have direct access to the interfaces.*

Initialization
The first part of the program is globally the same as in previous examples: create a loader, and verify if the file can be loaded. Here also note the absence of the prefix.

```
Dim Style, Text, Result       ' initialise variables
CanLoad = 0
FromDisk = New PNRFLoader    ' create new loader object
WithFileName = "E:\My Documents\My Recordings
\ForPNRFexample106.pnrf"
CanLoad = FromDisk.CanLoadRecording(WithFileName)   ' test
If (CanLoad < 100) Then
   Style = vbOKOnly + vbExclamation
   Text = "This file cannot be loaded."
   Result = MsgBox(Text, Style, "Load Recording")
   Application.Exit()
Else
   ' here goes the code if we can open the file
End If
```

At this point we can load the file:

```
MyData = FromDisk.LoadRecording(WithFileName)
```

Load data

Now the file is open and we can have a look of what is inside. We already have seen how to select recorders and channels in Chapter 3. For the sake of simplicity we will fetch data only from a fixed recorder/channel in the next example.

Create a data source

To fetch data from a channel we will need to create a data source. To do this, proceed as follows:

```
MySource = MyData.Recorders(1).Channels(1).DataSource
                (DataSourceSelect.DataSourceSelect_Mixed)
```

Note that there are multiple types of datasources:

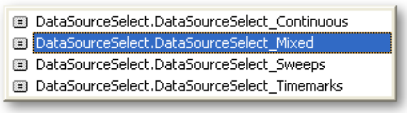

DataSourceSelect.DataSourceSelect_Continuous
DataSourceSelect.DataSourceSelect_Mixed
DataSourceSelect.DataSourceSelect_Sweeps
DataSourceSelect.DataSourceSelect_Timemarks

Figure 4.2: Multiple types of datasources

- **Continuous:** the retrieved data is continuous data only, i.e. in a mixed (dual-rate) recording the sweep (transient) data is ommitted.
- **Mixed:** use this mode by default. Now you are able to retrieve both types of data, continuous and sweeps.
- **Sweeps:** use this mode to retrieve sweeps only.
- **Timemarks:** can be used on recorder level only. Refer to the appendix on triggers.

Note *When using sweeps, retrieving data between sweeps will yield empty results.*

Since we now have an interface to a data source, we can try to get some data out of it:

```
MySource.Data(0, 100, Result)
```

This will retrieve data starting at t=0 seconds through t=100 seconds and place it in the intermediate **Result** object we have defined earlier.

Create a label on the form and add the following code:

```
Segments = Result
Label1.Text = Segments.Count
```

This will copy the data from **Result** into our **Segments** interface and display the segment count. The complete code after the between the `Else` and `End If` statement will look like this:

```
Else
   ' here goes the code if we can load
   MyData = FromDisk.LoadRecording(WithFileName)
   MySource = MyData.Recorders(1).Channels(1).DataSource
                (DataSourceSelect.DataSourceSelect_Mixed)
   MySource.Data(0, 100, Result)
   Segments = Result
   Label1.Text = Segments.Count
End If
```

Note *`Segments.Count` can be 0 (zero) even if there is data in the file. When the actual data starts at a time > 0 and you request data before that point, no segments and no data will be available.*

Inside a segment
Now that we have segments of data we would like to see what is in there. A variety of information is available. The most relevant ones at this point are:

- **NumberOfSamples:** the number of samples that are available within this segment in range 1 to 1 Gig (= segment limit).
- **StartTime / EndTime:** the start and end time respectively of this segment, expressed in seconds.
- **SampleInterval:** the time between two consecutive samples, expressed in seconds.

As we will see later in this section, the data can be retrieved as floating point data, integer data and original data:

- **Floating point data:** this data is in floating point format and scaled to the user scaling. The user scaling can be found by using the YRange or YFullRange methods.
- **Integer data:** raw ADC data. Needs to be scaled using the Y0 and YStep properties.
- **Original data:** one of the above, depending on the type of input channel.

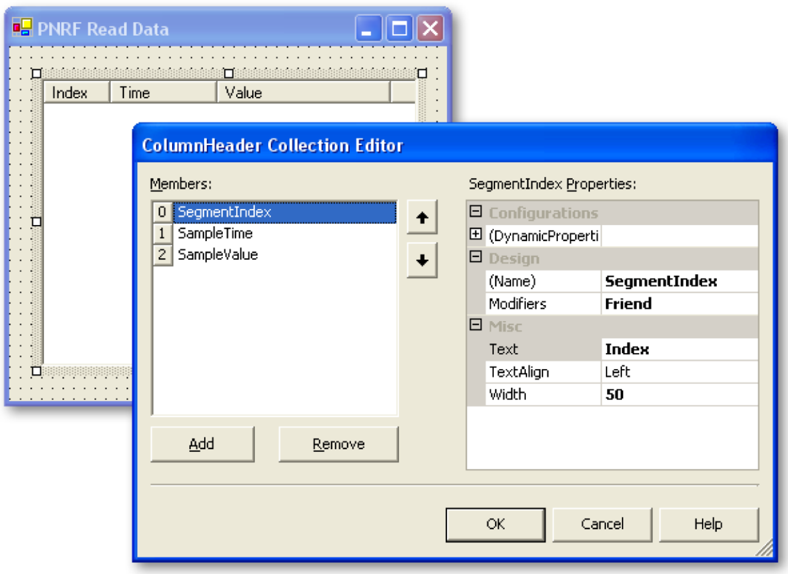
The units of both the X-axis and Y-axis can be found by using the `MySource.XUnit` and `MySource.YUnit` respectively.

With this information we must be able to retrieve and display data.

Create a dialog

Using the form we already have, remove the label that we used so far and add a **ListView** to the form. In this ListView we will use 3 columns, one for an index, one for the time and one for the actual value of the data. Make sure you have set the **View** property to **Details**.

Select the **Colums** property and click the More (...) button. This will call up the **ColumnHeader Collection Editor**. Create the three columns as shown below.



(Name)	SegmentIndex
Modifiers	Friend
Text	Index
TextAlign	Left
Width	50

Figure 4.3: PNRF Read Data and Column Header Collection Editor dialog

Continue with the code

Before we continue add the following declarations at the beginning of the code:

```
Dim i, Samples As Integer
Dim TimeStamp As Double
```

In the code we have come to the point that we have fetched a number of segments with data in it. We will concentrate on one segment only.

Replace the line:

```
Label1.Text = Segments.Count
```

in the previous example code with:

```
Samples = Segments(1).NumberOfSamples
```

Now we know the number of samples contained in the first segment. Note that this can be up to 1 GigaSample! We will use this to fetch the actual data:

```
Segments(1).Waveform(DataSourceResultType.DataSourceResul
tType_
Double64, 1, Samples, 1, Result)
```

The first parameter of this method defines the **data type** result as we have discussed earlier. Here we opt for the floating point data.
The second parameter is the **first sample** to fetch. Initially this should be "1".

Note *The third parameter defines the total **number of samples** we want. This can be less than the total number of samples available. You should **limit this to a reasonable value** that fits easily in PC memory like 1 MegaSample (can be 4 MegaByte).*

The next parameter sets the **reduction factor**. A reduction factor of "1" retrieves all samples on a one-by-one basis. A reduction factor of 3 or higher reduces the data by returning minmax pairs. Min-max pairs are also convenient for display purposes.

The last parameter defines the **result** location. The data is returned in the Result object. When you select a reduction factor of 2 or higher, the data is returned in Result as min-max pairs. In Visual Basic this will now be a two-dimensional array with Result(0, x) being the maximum values and Result(1, x) being the corresponding minimum values.

Now we can do some initialization stuff for the ListView as follows:

```
' create the listview entries
For i = 1 To Samples
   ListView1.Items.Add("")
Next
' add the x- and y-units to the column headers
ListView1.Columns(1).Text =
"X-Axis (" + MySource.XUnit + ")"
ListView1.Columns(2).Text =
"Y-Value (" + MySource.YUnit + ")"
```

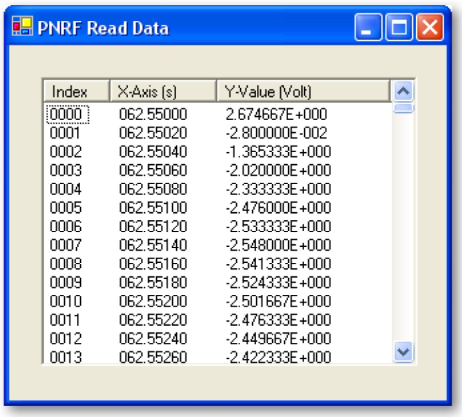
This will create sufficient ListView items. Also the column headers are modified to reflect the current X- and Y-units.

Finally we can present the data.

```
' fill the entries
For i = 0 To Samples - 1
    ' calculate time: start + index * step
    TimeStamp = Segments(1).StartTime + i * Segments(1).
          SampleInterval
    ' show index, time and value
    ListView1.Items(i).Text = Format(i, "0000")
    ListView1.Items(i).SubItems.Add(Format(TimeStamp,
    "000.00000"))
    ListView1.Items(i).SubItems.Add(Format(Result(i),
    "E"))
Next
```

Note *The indices run from 0 to Samples-1.*

When all is well and we run the program, the result could look like this:



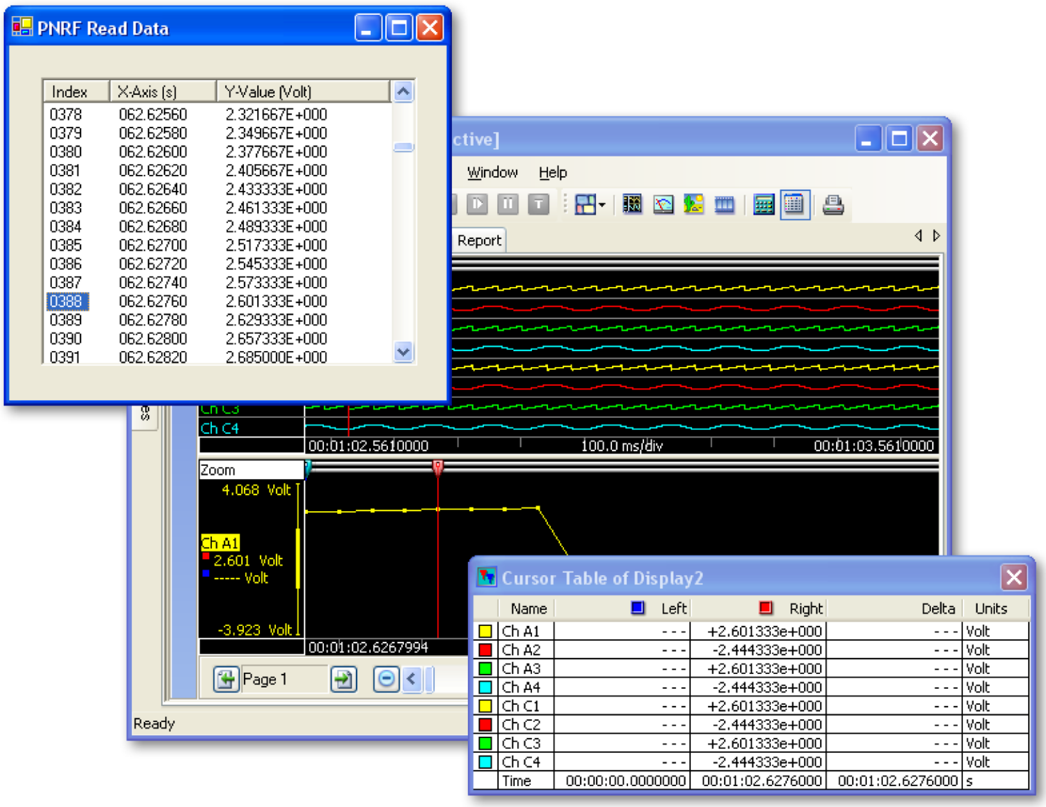
Index	X-Axis (s)	Y-Value (Volt)
0000	062.55000	2.674667E+000
0001	062.55020	-2.800000E-002
0002	062.55040	-1.365333E+000
0003	062.55060	-2.020000E+000
0004	062.55080	-2.333333E+000
0005	062.55100	-2.476000E+000
0006	062.55120	-2.533333E+000
0007	062.55140	-2.548000E+000
0008	062.55160	-2.541333E+000
0009	062.55180	-2.524333E+000
0010	062.55200	-2.501667E+000
0011	062.55220	-2.476333E+000
0012	062.55240	-2.449667E+000
0013	062.55260	-2.422333E+000

Figure 4.4: PNRF Read Data result list box

Creating and filling the ListView entries can take a long time. Therefore make sure that the number of samples is a few thousand or less.

4.5 Verify the result

To verify what has been accomplished you can use Perception to compare the results.



PNRF Read Data

Index	X-Axis (s)	Y-Value (Volt)
0378	062.62560	2.321667E+000
0379	062.62580	2.349667E+000
0380	062.62600	2.377667E+000
0381	062.62620	2.405667E+000
0382	062.62640	2.433333E+000
0383	062.62660	2.461333E+000
0384	062.62680	2.489333E+000
0385	062.62700	2.517333E+000
0386	062.62720	2.545333E+000
0387	062.62740	2.573333E+000
0388	062.62760	2.601333E+000
0389	062.62780	2.629333E+000
0390	062.62800	2.657333E+000
0391	062.62820	2.685000E+000

Cursor Table of Display2

	Name	🟦 Left	🟥 Right	Delta	Units
🟨	Ch A1	- - -	+2.601333e+000	- - -	Volt
🟥	Ch A2	- - -	-2.444333e+000	- - -	Volt
🟩	Ch A3	- - -	+2.601333e+000	- - -	Volt
🟦	Ch A4	- - -	-2.444333e+000	- - -	Volt
🟨	Ch C1	- - -	+2.601333e+000	- - -	Volt
🟥	Ch C2	- - -	-2.444333e+000	- - -	Volt
🟩	Ch C3	- - -	+2.601333e+000	- - -	Volt
🟦	Ch C4	- - -	-2.444333e+000	- - -	Volt
	Time	00:00:00.0000000	00:01:02.6276000	00:01:02.6276000	s

Start Perception and load the same file. Zoom in on a part of the channel data that falls inside the loaded segment, until you can see the individual samples (little squares). Drag a cursor to a location using 'sample snap': while dragging the cursor hold down the Controlkey. The cursor will snap to samples. Press the spacebar to call up the cursor window. Compare the results.

4.6 Begin and end of recording

In this chapter we have concentrated on fetching data from one segment, ranging from a - random - starting point to a - random - end point.

How do we find out what data is available? I.e. what is the very first starting point, very last end point and when did the recording start in the real world?

The very first relative start and very last relative end time can be found using the properties `MySource.Sweeps.StartTime` and `MySource.Sweeps.EndTime`. These values define the relative time of the first sample and the relative time of the last sample after the start of recording, no matter what is in between.

The relative start of the recording is always t=0. The corresponding real world time can be found using the UTC time function:

```
MySource.GetUTCTime(Year, YearDay, UTCTime, Valid)
```

in which:

- **Year:** integer representing the year, e.g. 2006
- **YearDay:** integer representing the number of the day within the year, e.g. 11 represents January 11, 40 represents February 9
- **UTCTime:** double representing the number of seconds after midnight. E.g. 49242.0 represents 13h40:42 (13x3600 + 40x60 + 42 = 49242)
- **Valid:** boolean is true when UTC time is available.

This concludes the introduction to the PNRF reader. Refer to the various appendices for more information on specific topics.

4.7 Data Values (from Info sheet)

Values from the Perception Info sheet (when available) are stored also within a PNRF file. You can retrieve these values through the recording interface:

```
Dim Icount As Integer
Dim DVType As DataSourceDataType
Dim DVValue As String

ICount = MyData.DataValues.Count
DVType = MyData.DataValues(1).DataType
DVValue = MyData.DataValues("Comment").Value
```

A Interfacing with MATLAB

A.1 Introduction

MATLAB® is a well-known language for technical computing. It integrates computation, visualization, and programming in an environment where problems and solutions are expressed in familiar mathematical notation.

MATLAB provides interfaces to clients or servers communicating via Component Object Model (COM). In this section we will describe how you can interface MATLAB to the PNRF Reader using COM. Most of what has been said in this manual is also true for the MATLAB environment. The syntax however is different.

The examples in this section use the default MATLAB desktop and are given as command line input. Version 7.1 (R14) of MATLAB is used to create these examples. Version 7.1 and Version 7.3 (R2006b) were used to test the examples.

A.2 Getting started

Create COM automation server and load data
To create a COM automation server you will need to use the **actxserver** as follows:

```
h = actxserver('progid')
```

This creates a COM server, and returns COM object, h, representing the server's default interface. Progid is the programmatic identifier of the component to instantiate in the server. For our application this becomes:

```
>> FromDisk = actxserver('Perception.Loaders.PNRF')

FromDisk =

   COM.Perception_Loaders_PNRF

>> FromDisk.get
   Description: 'Perception Recording File'
      Extension: 'PNRF'

>>
```

To see the properties available through the FromDisk interface we use **get**. To list the methods use **invoke**:

```
>> FromDisk.invoke
   CanLoadRecording = int32 CanLoadRecording(handle,
   string)
   LoadRecording = handle LoadRecording(handle, string)
   LoadRecordingFromInterface = handle
   LoadRecordingFromInterface(handle, handle)

>>
```

To interface to the data use:

```
>> MyData = FromDisk.LoadRecording('D:\temp\matlab.pnrf')

MyData =
   Interface.Perception_Recording_Interface.IRecording

>>
```

Inspect the file contents
At this point we have access to the recording interface of the PNRF file. Use **MyData. get** and **MyData.invoke** for additional information. Use this method to find the various capabilities as shown below.

```
>> MyData.Recorders.get
         Count: 5
     Recording: [1x1 Interface.Perception_Recording_
                Interface.IRecording]

>> MyData.Recorders.invoke
   Item = handle Item(handle, Variant)
>>
```

This means that there are 5 recorders and you can access each recorder by using the Item method. Continue to investigate until you see the following:

```
>> MyData.Recorders.Item(1).Channels.Item(1).get
          Name: 'Ch A1'
     Recording: [1x1 Interface.Perception_Recording_
                Interface.IRecording]
      Recorder: [1x1 Interface.Perception_Recording_
                Interface.IDataRecorder]
   ChannelType: 'DataChannelType_Analog'
     TimeShift: 0

>> MyData.Recorders.Item(1).Channels.Item(1).invoke
   DataSource = handle DataSource(handle,
   DataSourceSelect)
>>
```

A.3 Fetch data

Now you see the actual data source. This method returns a handle to the data source interface and requires the parameter `DataSourceSelect` as one of the following:

1 DataSourceSelect_Continuous
2 DataSourceSelect_Sweeps
3 DataSourceSelect_Mixed

To create the interface, assuming continuous data, use:

```
>> ItfData =
MyData.Recorders.Item(1).Channels.Item(1).DataSource(1)

ItfData =

   Interface.Perception_Recording_Interface.IDataSrc

>>
```

When a non-structure (empty) array is returned, try `DataSource(3)`. Investigate:

```
>> ItfData.get
            Name: 'Ch A1'
           XUnit: 's'
           YUnit: 'Volt'
        DataType: 'DataSourceDataType_AnalogWaveform'
        TimeInfo: 'DataSourceTimeInfo_Implicit'
          Status: 'DataSourceStatus_Static'
           Value: NaN
          Sweeps: [1x1 Interface.Perception_Recording_
                  Interface.IDataSweeps]
      Properties: [1x1 Interface.Perception_Recording_
                  Interface.IProperties]

>> ItfData.invoke
   Data = Variant(Pointer) Data(handle, double, double)
   GetUTCTime = [int32, int32, double, bool]
   GetUTCTime(handle)
   GetValueAtTime = Variant GetValueAtTime(handle, double)
>>
```

Assume we want the data between 63 and 78 seconds:

```
>> SegmentsOfData = ItfData.Data(63, 78)

SegmentsOfData =

   Interface.Perception_Recording_Interface.
   IDataSegments

>> SegmentsOfData.get
   Count: 1

>> SegmentsOfData.invoke
   Item = handle Item(handle, int32)
   Positions = [SafeArray Pointer(double), SafeArray
         Pointer(double), SafeArray Pointer(int32)]
         Positions(handle)
>>
```

There is one segment of data with the following information:

```
>> SegmentsOfData.Item(1).get
       StartTime: 63
         EndTime: 78
  SampleInterval: 2.0000e-004
 NumberOfSamples: 75001
 RelationToPrevio 'SegmentRelation_None'
              us:
              Y0: 0
           YStep: 3.3333e-004
     DisplayFrom: 10
       DisplayTo: -10
```

```
>> SegmentsOfData.Item(1).invoke
   BestReductionFactor = int32 BestReductionFactor
                         (handle, int32)
   DisplayRange = [double, double] DisplayRange(handle)
   MathCalculations = [Variant(Pointer), Variant(Pointer),
                      Variant(Pointer), Variant(Pointer)]
                      MathCalculations(handle,
                      int32, int32, int32)
   Waveform = Variant(Pointer) Waveform(handle,
              DataSourceResultType, int32, int32, int32)
   XFullRange = [double, double] XFullRange(handle)
   XRange = [double, double, double, int32] XRange(handle)
   YFullRange = [double, double, double, double, double,
                double]
                YFullRange(handle)
   YRange = [double, double] YRange(handle)
>>
```

Use the Waveform method to get the data. This method requires the following:

- **DataSourceResultType:** the data can be retrieved as floating point data (4), integer data (2) and original data (-1).
- **FirstSample** (int32): the first sample to retrieve.
- **ResultCount** (int32): the number of samples to retrieve.
- **Reduction** (int32): reduction factor

Example:

```
>> WaveformData = SegmentsOfData.Item(1).Waveform(4, 1,
200, 1)

WaveformData =

   Columns 1 through 9

...

>>
```

And to create a nice plot:

```
>> plot (WaveformData)
>> title(ItfData.Name)
>>
```

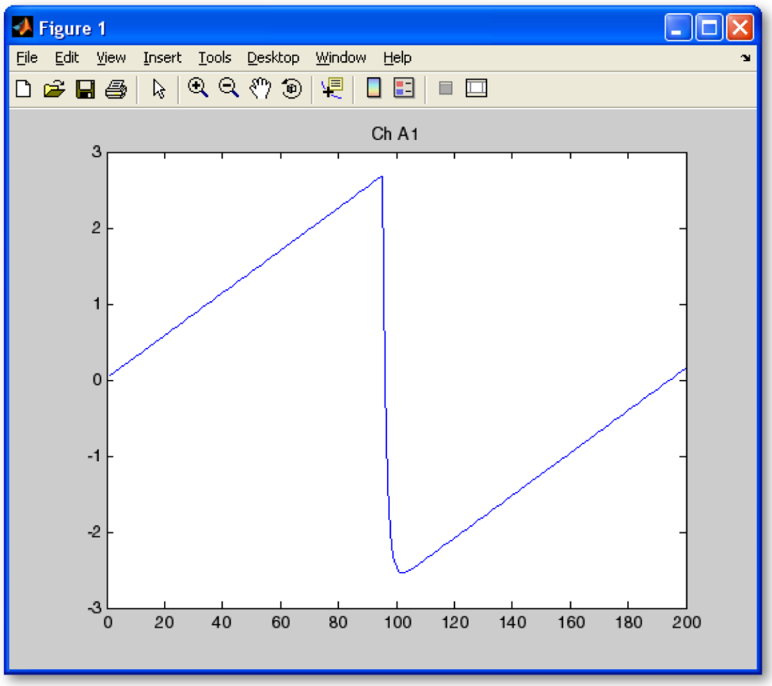


Figure A.1: MATLAB Plot

A.4 Begin and end of recording

To find the real world start of recording use:

```
>> [year day time valid] = ItfData.GetUTCTime


year =
   2006
day =
   11
time =
   49219
valid =
   1
>>
```

To find the relative start and stop of the recording use:

```
>> ItfData.Sweeps.get
           Count: 1
       StartTime: 62.5500
         EndTime: 78.9330
>>
```

You use `ItfData.Sweeps.Item(i).get` to fetch information about a specific sweep "i". Example: assume two segments. The results can be as follows:

```
>> ItfData.Sweeps.get
           Count: 2
       StartTime: 9.6000
         EndTime: 11.9330

>> ItfData.Sweeps.Item(1).get
       StartTime: 9.6000
         EndTime: 10.0998
     TriggerTime: 9.7000
   TriggerSource: 'TriggerSource_Manual'
        Finished: 1

>> ItfData.Sweeps.Item(2).get
       StartTime: 11.4332
         EndTime: 11.9330
     TriggerTime: 11.5332
   TriggerSource: 'TriggerSource_Manual'
        Finished: 1
>>
```

A.5 Data Values (from Info sheet)

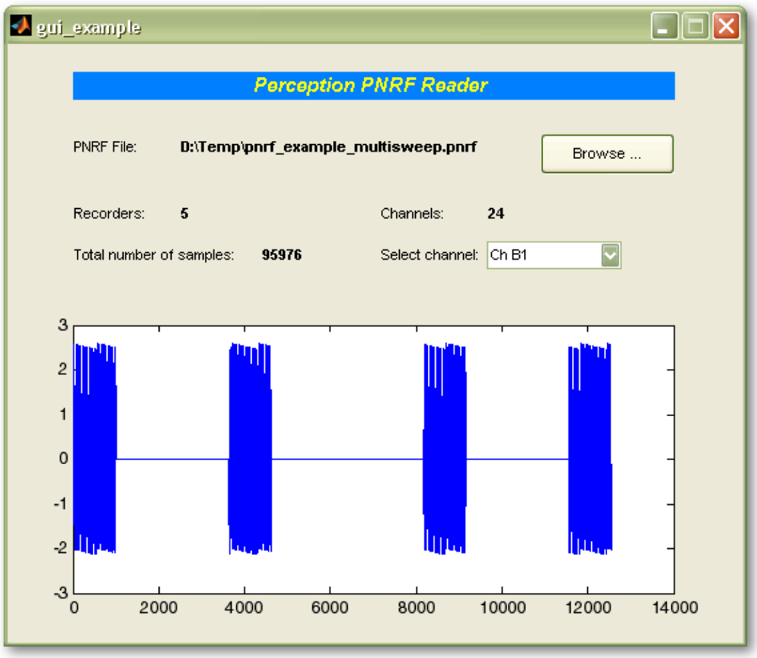
Values from the Perception Info sheet are stored also within a PNRF file. You can retrieve these values through the recording interface:

```
>> MyData.get
            Title: 'pnrf_example_runup'
        Recorders: [1x1 Interface.Perception_Recording_
                   Interface.IDataRecorders]
         Channels: [1x1 Interface.Perception_Recording_
                   Interface.IDataChannels]
TimeTagRecorders: []
           Sweeps: []
         Filename: 'D:\temp\pnrf_example_runup.pnrf'
          Comment: ''
           Groups: [1x1 Interface.Perception_Recording_
                   Interface.IDataGroups]
PreferredDisplayLayout: [1x9001 char]
       DataValues: [1x1 Interface.Perception_Recording_
                   Interface.IDataValues]

>> MyData.DataValues.get
            Count: 3

>> MyData.DataValues.Item(1).get
            Value: ''
         DataType: 'DataSourceDataType_String'
             Name: 'Comment'
            Units: ''

>> MyData.DataValues.Item(2).get
            Value: 'Administrator'
         DataType: 'DataSourceDataType_String'
             Name: 'UserName'
            Units: ''

>> MyData.DataValues.Item(3).get
            Value: 'LDS Test and Measurement'
         DataType: 'DataSourceDataType_String'
             Name: 'Company'
            Units: ''

>> MyData.DataValues.Item('Comment').get
            Value: ''
         DataType: 'DataSourceDataType_String'
             Name: 'Comment'
            Units: ''
>>
```

Note *Legacy PNRF files do not contain DataValues. Therefore the array will be empty in these files.*

A.6 GUI example

The following diagram shows an example of a MATLAB program with a user interface. The program allows you to select a file. Once opened it gives details about the recording. You can select a channel to be displayed.



PNRF File: D:\Temp\pnrf_example_multisweep.pnrf

Recorders: 5
Channels: 24

Total number of samples: 95976
Select channel: Ch B1

Figure A.2: MATLAB GUI

B Enumerations

B.1 Introduction

Various methods require a number as a means to select an option. these numbers are defined. This appendix lists all the enumerations that are currently in use.

B.1.1 List of enumerations

```
typedef enum {
   DataSourceSelect_Continuous = 1,
   DataSourceSelect_Sweeps = 2,
   DataSourceSelect_Mixed = 3,
   DataSourceSelect_Timemarks = 10
} DataSourceSelect;


typedef enum {
   DataChannelType_Analog = 1,
   DataChannelType_Event = 2
} DataChannelType;


typedef enum {
   DataSourceDataType_Unknown = 0,
   DataSourceDataType_Numerical = 1,
   DataSourceDataType_String = 2,
   DataSourceDataType_AnalogWaveform = 3,
   DataSourceDataType_DigitalWaveform = 4,
   DataSourceDataType_TimeMarks = 10
} DataSourceDataType;


typedef enum {
   DataSourceResultType_Original = 0xffffffff,
   DataSourceResultType_Int16 = 2,
   DataSourceResultType_Double64 = 4
} DataSourceResultType;


typedef enum {
   DataSourceStatus_Static = 0,
   DataSourceStatus_Dynamic = 1
} DataSourceStatus;
```

```
typedef enum {
   DataSourceTimeInfo_Implicit = 0,
   DataSourceTimeInfo_Explicit = 1,
   DataSourceTimeInfo_External = 2
} DataSourceTimeInfo;

typedef enum {
   SegmentRelation_None = 0,
   SegmentRelation_Continuous = 1,
   SegmentRelation_Overlapped = 2
} SegmentRelation;

typedef enum {
   DataSrcInfoMaskItem_Name = 1,
   DataSrcInfoMaskItem_XUnits = 2,
   DataSrcInfoMaskItem_YUnits = 4,
   DataSrcInfoMaskItem_UTCTime = 8
} DataSrcInfoMaskItem;

typedef enum {
   TriggerSource_NoTrigger = 0,
   TriggerSource_Unknown = 1,
   TriggerSource_Bus = 2,
   TriggerSource_Manual = 3,
   TriggerSource_External = 4,
   TriggerSource_Auto = 5,
   TriggerSource_Display = 6,
   TriggerSource_Channel = 7,
   TriggerSource_MyChannel = 99,
   TriggerSource_NoData = 0x80000000,
   TriggerSource_Mask = 65535
} TriggerSourceInfo;

typedef enum {
   TimeMarkType_Trigger = 1,
   TimeMarkType_TriggerAnnotation = 2,
   TimeMarkType_BookMark = 4,
   TimeMarkType_Marker = 8,
   TimeMarkType_VoiceMark = 16,
   TimeMarkType_EventMark = 32
} TimeMarkType;
```

```
typedef enum {
   EventMark_RecordingStart = 1,
   EventMark_RecordingEnd = 2
} EventMarkType;
```

C Summary of Commands

C.1 Introduction

Each object within the PNRF Reader and Recording interface can have properties and methods. This appendix lists the most relevant properties and methods that are currently in use.

C.1.1 List of properties and methods

PNRFLoader		
Properties		
	Description	File type description
	Extension	File type extension
Methods		
	CanLoadRecording	Yes / No
	LoadRecording	Load recording

IRecording		
Properties		
	Title	Title of recording
	Recorders	All recorders
	Channels	All channels
	Sweeps	Sweeps collection
	Filename	Filename
	Comment	Comment
	Groups	Groups collection
	PreferredDisplayLayout	Proprietary XML stream
	DataValues	Data of Perception Info sheet
Methods		
	GetUTCTime	UTC time start of recording
	DeleteOnRelease	Delete file on release of i'fces

IRecording.Recorders		
Properties		
	Count	Number of recorders
Methods		
	Item	Retrieve interface

IRecording.Recorders.Item(i) = IDataRecorder		
Properties		
	Name	Name of recorder
	PhysicalName	Physical name of recorder
	Channels	All channels in recorder

IRecording.Recorders.Item(i) = IDataRecorder	
Triggers	Collection of trigger infos
XUnits	X-Units (timebase)

IDataRecorder.Channels	
Properties	
Count	Number of channels
Methods	
Item	Retrieve interface

IDataRecorder.Channel.Item(i) = IDataChannel	
Properties	
Name	Name of channel
ChannelType	Type of channel
Methods	
DataSource	Retrieve pointer to data source

IDataSrc	
Properties	
Name	Name of data source
XUnit	Timebase units
YUnit	Y-axis units
DataType	Type of data, e.g. analog
TimeInfo	Timebase source
Sweeps	Sweeps information
Properties	Collection of properties
Methods	
Data	Fetch data segments

IDataSrc.Sweeps	
Properties	
Count	Number of sweeps
StartTime	Start time of total recording
EndTime	End time of total recording
Methods	
Item	Retrieve interface to sweeps

IDataSrc.Sweeps.Item(i) = IDataSweep		
Properties		
	StartTime	Start time of this sweep
	EndTime	End time of this sweep
	TriggerTime	Time of trigger
	TriggerSource	Source of trigger
	Finished	Sweep has finished yes / no
Methods		
	GetInfo	Get trigger information

IDataSegments		
Properties		
	Count	Number of data segments returnedby a call to the Data method ofIDataSrc
Methods		
	Item	Retrieve interface to segment

IDataSegments.Item(i) = IDataSegment		
Properties		
	StartTime	Start time of this segment
	EndTime	End time of this segment
	SampleInterval	Time interval between samples
	NumberOfSamples	Number of samples in thissegment
	Y0	Y0 value
	YStep	Input range / ADC # of values
Methods		
	Waveform	Returns the data

D Triggers and Markers

D.1 Introduction

There is more to a recording then only data. Within a recording you can also have markers. The following markers are currently supported within the PNRF Definition:

- **Bookmark** indicates text annotation of data.
- **Marker** a manually initiated mark, usually from a front panel button.
- **Trigger** indicates a trigger event.
- **Trigger Annotation** indicates a trigger condition.
- **Voicemark** a microphone / sound annotation of data.
- **Eventmark** indicates any other type of event. Currently the begin and end of a recording are supported.

Within a recording any number of these markers can be present. Therefore these markers are grouped in a separate data stream.

Note *The type of markers available depends on the instrument that created the recording. Therefore some markers may not be available as type of markers.*

For programming details refer to the relevant chapters in this document.

D.2 Triggers

Triggers and trigger annotations are easily accessed through the **`Recording.Recorders`** interface as well as the **`DataSource.Sweeps`** interface:

```
Dim TriggerCount, TriggerTime, TriggerType, TriggerSource

MyData = FromDisk.LoadRecording(WithFileName)
TriggerCount = MyData.Recorders(1).Triggers.Count
TriggerTime = MyData.Recorders(1).Triggers(1).Time
TriggerType = MyData.Recorders(1).Triggers(1).MarkType
```

When there are sweeps you can continue with:

```
MySource = MyData.Recorders(1).Channels(1).DataSource
          (DataSourceSelect.DataSourceSelect_Mixed)
TriggerTime = MySource.Sweeps(1).TriggerTime
TriggerSource = MySource.Sweeps(1).TriggerSource
```

D.3 Markers

To access the various markers you will need to use a different technique. As mentioned earlier, any number of markers can be present. Therefore these markers are grouped in a separate data stream.

To gain access to this datastream we use again the DataSource concept. However, we will now retrieve a TimeMarks stream of data. And, instead of a list of segments, a collection of markers is returned.

```
Dim MyMarkers As ITimeMarks
```

Instead of fetching data from a channel, we will now fetch data from the recorder, using the TimeMarks data stream:

```
MySource = MyData.Recorders(1).DataSource
             (DataSourceSelect.DataSourceSelect_Timemarks)
MySource.Data(0, 63, Result)
MyMarkers = Result
TriggerCount = MyMarkers.Count
```

Examine the first marker:

```
TriggerType = MyMarkers(1).MarkType
TriggerTime = MyMarkers(1).Time
```

A more sophisticated approach uses the following code:

```
Dim MyEventMark As IEventMark
Dim MyTriggerMark As ITrigger
```

```
Select Case MyMarkers(1).MarkType
     Case TimeMarkType.TimeMarkType_EventMark
          MyEventMark = MyMarkers(1)
          TriggerTime = MyEventMark.Time
          TriggerType = MyEventMark.EventType()
     Case TimeMarkType.TimeMarkType_Trigger
          MyTriggerMark = MyMarkers(1)
          TriggerTime = MyTriggerMark.Time
          TriggerSource = MyTriggerMark.TriggerSource
End Select
```

In the above code the type of marker determines the end result. For events the event type can be returned. For triggers the trigger source is available.

E Sweeps and Segments

E.1 Introduction

Each PNRF file contains one (1) **recording**. A recording has a start time and a stop time. Between these times data can be stored in various ways. However, all time information is related to a single start and stop time. Even if we would theoretically combine two recordings into one file, the end result would be a single recording with modified start and stop times. Although a recording is started at relative time t=0, this does not mean data is also archived starting at that time.

Within the PNRF definition a recording can have the following types of data storage by nature:

- **Continuous:** data is acquired and stored as a continuous data stream without gaps. There is one begin and one end.
- **Sweeps:** data is stored in blocks. Each block is a sweep and has its own start and stop time. Each sweep has a trigger. A recording can have multiple sweeps.
- **Mixed:** the stored data is a mixture of continuous data and sweeps, each with their own sample rate.

When you want to fetch data from a PNRF file you will need to consider this data source select issue. The safest option is to go always for the mixed data source.

When you retrieve the data, you will get initially a list of **segments**. Each segment is a piece of data with its own X- and Y- information as well as begin and end time. Data may be segmented due to timebase changes as well as amplifier range changes. Also gaps create segements.

We will demonstrate the concepts of segments with some examples.

E.2 Continous data

Assume a continous recording as depicted in Figure E.1.

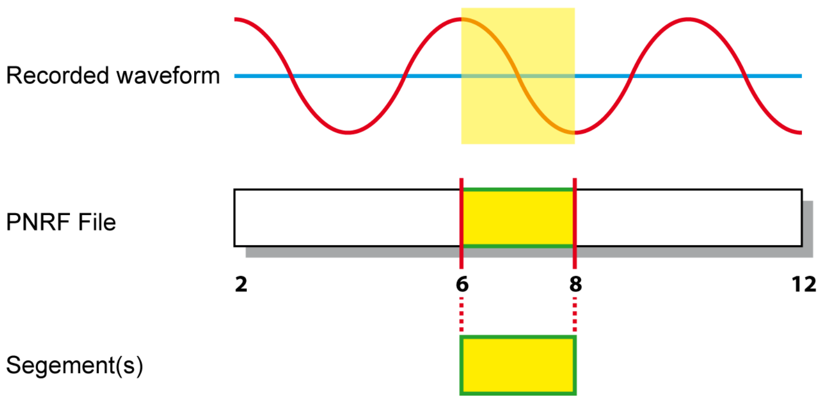


Figure E.1: Continous data recording

Use either the continuous or mixed mode select for the data source.

Start and end of the complete recording can be found by using the properties **`MySource.`**
`Sweeps.StartTime and MySource.Sweeps.EndTime.`

If you want to retrieve data between 6 and 8 seconds you will use something like:

```
MySource.Data(6, 8, Result)
Segments = Result
Label1.Text = Segments.Count
```

At this point **`Segments.Count`** should be one (1), i.e. the data you requested is contained within one segment. You can have a look at the properties of **`Segments(1)`** to find out the details.

E.3 Sweeps

Assume a recording with sweeps as shown in Figure E.2.

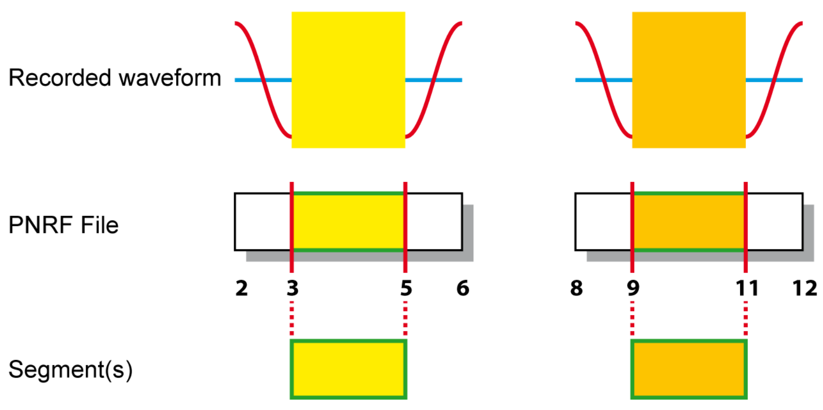


Figure E.2: Recording with sweeps (Part 1)

This is a single recording starting at t=2 that ends at t=12. When you want to retrieve data select either the sweeps or mixed mode select for the data source.

If you use `MySource.Data(3, 5, Result)` or `MySource.Data(9, 11, Result)` or something equal, the result will be a single segment.

In the following example we will retrieve data from the same file between t= 4 and t=10. To fetch this data proceed as usual:

```
MySource.Data(4, 10, Result)
Segments = Result
Label1.Text = Segments.Count
```

The value of `Segments.Count` however will now be 2.

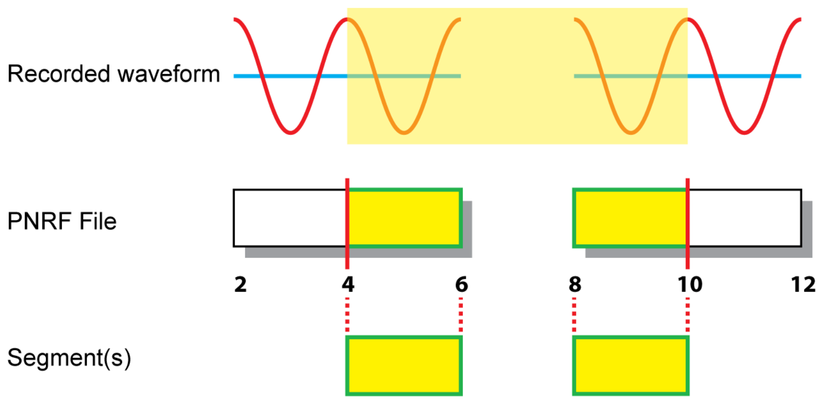


Figure E.3: Recording with sweeps (Part 2)

Use `Segments(1)` and `Segments(2)` to find out the details of each segment.

E.4 Mixed data

Within a PNRF file both continuous data and sweeps can be available. In this situation two data streams are available: one continuous and one with sweeps, typically at a higher sample rate. Depending on the choises you make you will get continuous data, sweeps or both.

Refer to Figure E.4.

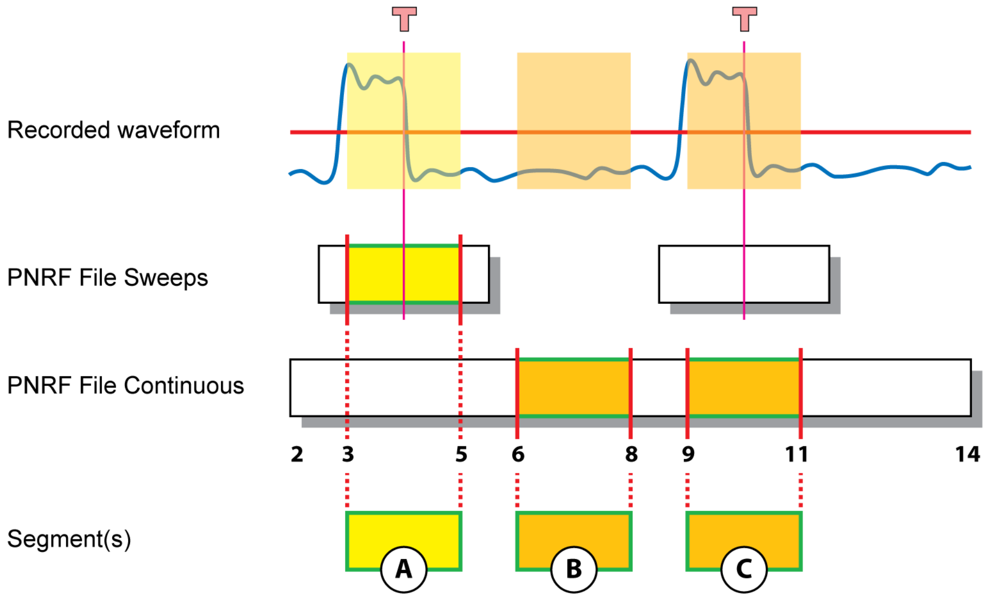


Figure E.4: Recording with continous data and sweeps

Here a situation is depicted in which both continuous data and sweeped data are available. Now there are three basic options:

A Request a single segment of sweeped data: use sweep data or mixed data as data source:

```
MySource = MyData.Recorders(1).Channels(1).DataSource
               (DataSourceSelect.DataSourceSelect_Sweeps)
MySource.Data(3, 5, Result)
```

```
MySource = MyData.Recorders(1).Channels(1).DataSource
               (DataSourceSelect.DataSourceSelect_Mixed)
MySource.Data(3, 5, Result)
```

As result a single segment will be returned with the sweeped data in it. Using the continuous data source would result in the situation as described at (C).

B Request a single segment of continuous data: select continuous or mixed data as data source:

```
MySource = MyData.Recorders(1).Channels(1).DataSource
               (DataSourceSelect.DataSourceSelect_Continuous)
MySource.Data(6, 8, Result)
```

```
MySource = MyData.Recorders(1).Channels(1).DataSource
                (DataSourceSelect.DataSourceSelect_Mixed)
MySource.Data(6, 8, Result)
```

As result a single segment will be returned with the continuous data in it. Using the sweeps data source would return nothing.

C Request a single segment of continuous data that overlaps sweep data: select continuous as data source:

```
MySource = MyData.Recorders(1).Channels(1).DataSource
               (DataSourceSelect.DataSourceSelect_Continuous)
MySource.Data(9, 11, Result)
```

As result a single segment will be returned with the continuous data in it. Using the sweeps or mixed data source would return the sweeped data as described at (A).

Usually you will not know exactly what kind of data is in the file. Therefore you will retrieve a section with 'random' start and end point. Depending on the type of data source select you can have various results.

Refer to Figure E.5

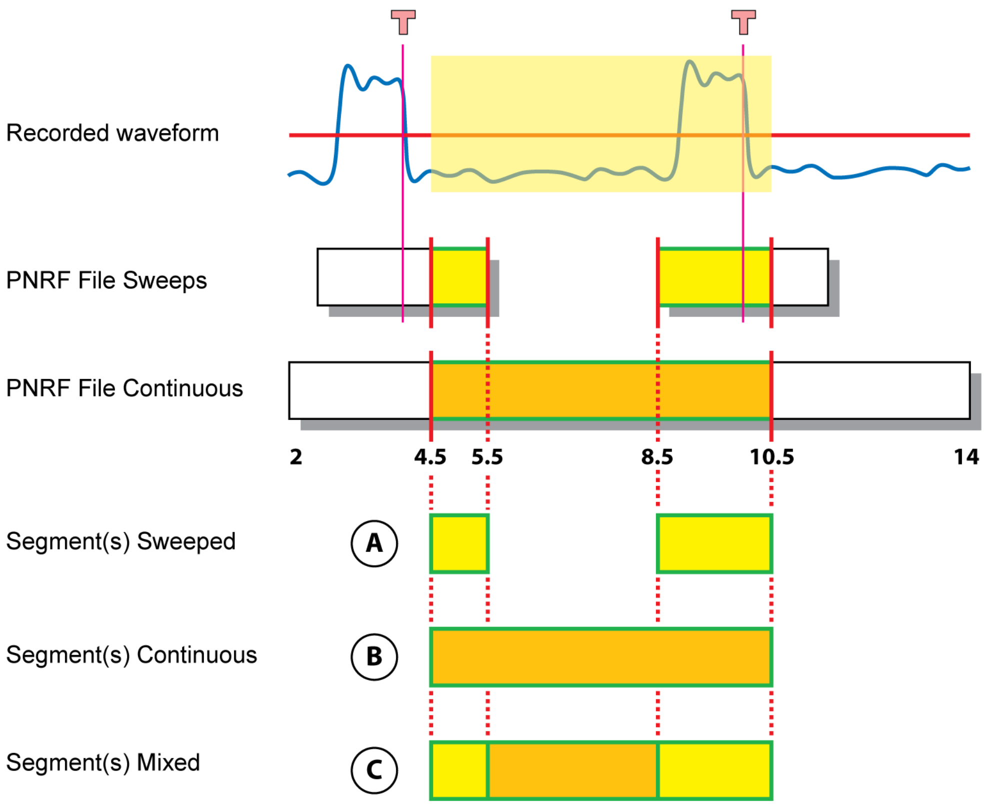


Figure E.5: Recording with continous data, sweeps and mixed mode

Assume we want to select the data between t=4.5 and t=10.5.

A Use sweeps: you will get two segments, one ranging from 4.5 to 5.5 seconds, the other one starting at 8.5 seconds and ending at 10.5 seconds.

B Select continuous. The result will be a single segment that comprises the complete requested data.

C Select mixed mode: now you will get three segments. The first one ranges from 4.5. to 5.5 seconds, containing (high speed) sweep data. The second segment has (slow speed) continuous data, starting at 5.5 seconds* and ending at 8.5 seconds. The last segment again has sweep data up to 10.5 seconds.

* Actually the time between the last sample of the segment and the first sample of the next segment will be somewhere between **IDataSegment.SampleInterval** of the segment and the **IDataSegment.SampleInterval** of the next segment. Due to the nature of HBM digitizing equipment however, the exact timing of these samples is correct and they are synchronized.

F Using C++

F.1 Introduction (Using C++)

C++ (pronounced "see plus plus") is a general-purpose, high-level programming language with low-level facilities. It is a statically typed free-form multi-paradigm language supporting procedural programming, data abstraction, object-oriented programming, generic programming and RTTI. Since the 1990s, C++ has been one of the most popular commercial programming languages.

In this appendix we will give an example of how to use the PNRF API with C+ +. The code is also included "as-is" with the PNRF Reader SDK and developed in the Microsoft Visual Studio development environment (2005 edition).

F.2 Initialization

Initialize the program. Depending on your requirements include the various libraries. Include the Loader and Recording interface by ID rather than by actual name.

```
// PNRFLoadExample.cpp: Defines the entry point
// for the console application.

#include"stdafx.h"              // standard stuff
#include<stdio.h>
#include<conio.h>
#include<iostream>              // basic console I/O
#include<atlcomcli.h>
using namespace std;
// #import "percRecordingInterface.olb" no_namespace
#import"libid:8098371E-98AD-0070-BEF3-21B9A51D6B3E"
no_namespace
// #import "percPNRFLoader.dll" no_namespace
#import"libid:8098371E-98AD-0062-BEF3-21B9A51D6B3E"
no_namespace

char cAnyKey;                   // key input
double dStart, dEnd;            // start and stop time
BSTR myUnits;                   // units
VARIANT myCompany;              // company name
```

F.3 Main

The main program uses **CoInitialize** to initialize the COM library on the current thread. Applications must initialize the COM library before they can call COM library functions.

After this the actual program **ProcessFile()** is called.

Once done **CoUninitialize** closes the COM library on the current thread, unloads all DLLs loaded by the thread, frees any other resources that the thread maintains, and forces all RPC connections on the thread to close.

```
int _tmain(int argc, _TCHAR* argv[])
{
   CoInitialize(NULL);
   ProcessFile();
   cout << endl << "Done. Press any key to quit." << endl;
   _getch();
   CoUninitialize();
   return 0;
}
```

F.4 Process the data

The procedure **ProcessFile** is used to process the data. We start with some initialization and also fetch some information. DataValues are not always available in a PNRF file. Therefore you must check if these values exist. Usually three values are available by default: comment, user and company. You can access these by index number or actual name.

After the information is displayed key input is used to continue or quit.

```
void ProcessFile()
{
   IRecordingLoaderPtr itfLoader;
   itfLoader.CreateInstance(__uuidof (PNRFLoader));
   // Enter the name of the pnrf recording here
   IRecordingPtr itfRecording = itfLoader->LoadRecording
        ("d:\\Temp\\pnrf_example_runup.pnrf" );
   // print recording name
   _tprintf(TEXT("Recording title: %s\n"),
        (wchar_t *)itfRecording->Title);
   // print company name when data values are available
   if (itfRecording->DataValues != NULL)
   {
        myCompany = itfRecording->DataValues->
             Item["Company"]->GetValue();
        _tprintf(TEXT("Company: %s\n"),
             (wchar_t*)myCompany.bstrVal);
   }
   // connect to data source channel 1
   IDataSrcPtr MySource = itfRecording->Channels->
        Item[1]->DataSource[DataSourceSelect_Mixed];
   // fetch YUnits
   MySource->get_YUnit(&myUnits);
   _tprintf(TEXT("Y Unit: %s\n", (wchar_t*)myUnits);
   // fetch start and stop time
   MySource->Sweeps->get_StartTime(&dStart);
   MySource->Sweeps->get_EndTime(&dEnd);
   _tprintf(TEXT(
   "Start time: %lf s, End time: %lf s\n\n"),
        dStart, dEnd);
   cout <<
   "Press any key to continue, or Q to quit\n\n";
   cAnyKey = _getch();
   if ((cAnyKey == 'Q') || (cAnyKey == 'q'))
             return;
```

The result could look like this:

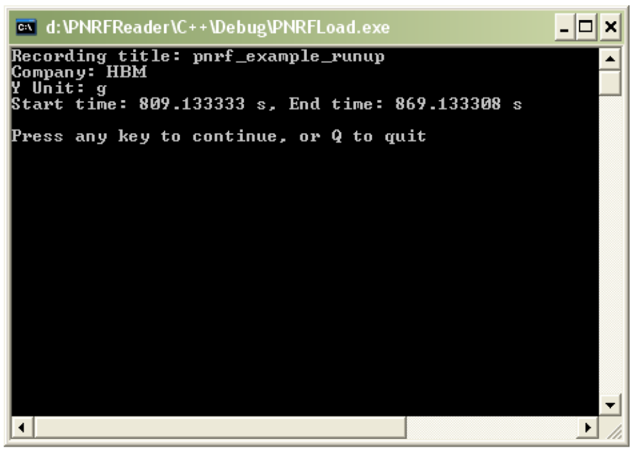

```
d:\PNRFReader\C++\Debug\PNRFLoad.exe
Recording title: pnrf_example_runup
Company: HBM
Y Unit: g
Start time: 809.133333 s, End time: 869.133308 s

Press any key to continue, or Q to quit
```

Figure F.1: DOS window of the PNRFLoad.exe

We continue by connecting to the data an investigating the number of segments. Quit when there is no data, or when no segments are found.

```
// create data array as variant
CComVariant myData;

// Get data between start and stop time
MySource->Data(dStart, dEnd, &myData);

// if object is empty: quit
if (myData.vt == VT_EMPTY)
{
      _tprintf(TEXT("No Data"));
      return;
}

// create segments pointer
IDataSegmentsPtr itfSegments = myData.punkVal;

int iSegIndex = 1;           // segment index
int iCount = itfSegments-    // number of
>Count;                      segements
if (iCount < 1)
{
      _tprintf(TEXT("No Segments found\n"));
      return;
}
```

At this point we can loop through all available segments and display the data. Before we actually display the data we display the number of data points and give the option to continue or quit.

```
// loop through all available segments
for (iSegIndex = 1 ; iSegIndex <= iCount ;
iSegIndex++)
{
     // pointer inside segment data
     IDataSegmentPtr itfSegment = NULL;
     itfSegments-
     >get_Item(iSegIndex,);&itfSegment);
     int lCnt = itfSegment->NumberOfSamples;

     // display info before continuing
     _tprintf(TEXT("Segment %d: %d samples\n"),
     iSegIndex,
          lCnt);
     cout << "Press any key to continue, or Q to
     quit" ;
     cAnyKey = _getch();
     if ((cAnyKey == 'Q') || (cAnyKey == 'q'))
          return;
     // variant data array for segment data
     CComVariant varData;
     // fetch data
     itfSegment-
     >Waveform(DataSourceResultType_Double64, 1,
          lCnt, 1, &varData);
     //If there is no data, process next segment
     if (varData.vt == VT_EMPTY)
          continue;
     //If it isn't an array, something is wrong here
     if (!(varData.vt & VT_ARRAY))
          continue;
     // Get data out through the use of the safe
     array
     // and store locally
     SAFEARRAY* satmp = NULL;
     satmp = varData.parray;

     if (satmp->cDims > 1)
     {
          // It's a multi dimensional array
          _tprintf(TEXT("Too many dimensions.\n"));
          continue;
     }
     double *pData;
```

```
                    SafeArrayAccessData(satmp,(void**)&pData);
                    double X0 = itfSegment->StartTime;
                    double DeltaX = itfSegment->SampleInterval;
                    double X, Y;

                    for (int i = 0; i < (int)satmp-
                    >rgsabound[0].cElements;
                         i++)
                    {
                         X = X0 + i * DeltaX;
                         Y = pData[i];
                         _tprintf(TEXT("%d (X, Y) = (%g, %g)\n"), i
                         +1, X,
                              Y);
                    }
                    SafeArrayUnaccessData(satmp);
          }
}
```

This concludes our C++ example.

G Using C#

G.1 Introduction (Using C#)

C# (pronounced see sharp) is an object-oriented programming language developed by Microsoft as part of their .NET initiative, and later approved as a standard by ECMA and ISO. C# has a procedural, object-oriented syntax based on several other programming languages (most notably Delphi and Java) with a particular emphasis on simplification.

The de facto standard implementation of the C# language is Microsoft C# compiler, included in every installation of .NET Framework. The original .NET Framework distributions from Microsoft included several language-to-IL compilers, including the two primary languages:
C# and Visual Basic. The bulk of the differences between C# and VB.NET from a technical perspective are syntactic sugar. That is, most of the features are in both languages, but some things are easier to do in one language than another.

It should be noted that all .NET programming languages share the same runtime engine, and when compiled produced binaries that are seamlessly compatible with other .NET programming languages, including cross language inheritance, exception handling, and debugging.

In this appendix we will give an example of how to use the PNRF API with C#. The code is also included "as-is" with the PNRF Reader SDK and developed in the Microsoft Visual Studio development environment (2005 edition).

G.2 Initialization

Start a new project and include the correct references. To do this select the ***Add Reference...*** command in the ***Project*** menu. You are now presented a list of registered references. Use the ***Browse...*** button to open the Select Component dialog. In this dialog browse to the **C:\Program Files\Common Files\HBM\Components** folder. In this folder select **percPNRFLoader.dll** and **percRecordingInterface.olb** and click Open to add these files to the list of References. Select OK. Now the RecordingLoaders and RecordingInterface are added to your list of included references.

Depending on your installation you can also include the COM reference **Perception PNRF Loader** when available. This will load the two references in one go.

Now you can initialize the program.

```
using System;
using System.Collections.Generic;
using System.Text;
using RecordingLoaders;
using RecordingInterface;

namespace Ex1_CSharp_PNRF_Load
{
  classProgram
  {
    static void Main(string [] args)
    {
         object myCompany; // company name
         string myUnits;   // units
         double dStart;    // start time
         double dEnd;      // stop time

         ConsoleKeyInfo cki = newConsoleKeyInfo ();
```

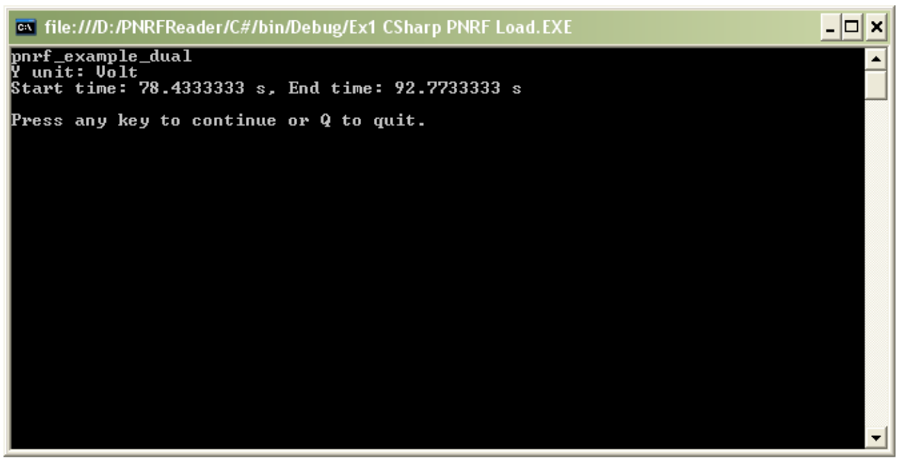
G.3 Process the data

We start with some initialization and also fetch some information. DataValues are not always available in a PNRF file. Therefore you must check if these values exist. Usually three values are available by default: comment, user and company. You can access these by index number or actual name.

After the information is displayed key input is used to continue or quit.

```
// create a pnrfloader
PNRFLoader FromDisk = new PNRFLoader();
// enter the correct name of the pnrf file here
String WithFileName = "D:\\Temp\
\pnrf_example_dual.pnrf" ;
// load recording
IRecording myData =
FromDisk.LoadRecording(WithFileName);
// print recording name
Console.WriteLine(myData.Title);
if (myData.DataValues != null)
{
   // only when data values are available
   myCompany =
   myData.DataValues["Company"].Value;
   Console.WriteLine("Company: " + myCompany);
}
// connect to the first channel as data source
IDataSrc mySource =
myData.Channels[1].get_DataSource
(DataSourceSelect.DataSourceSelect_Mixed);

// display the y-units
myUnits = mySource.YUnit;
Console.WriteLine("Y unit: " + myUnits);

// get start and stop time
dStart = mySource.Sweeps.StartTime;
dEnd = mySource.Sweeps.EndTime;
Console.WriteLine("Start time: {0} s, End time:
{1} s\n",
   dStart, dEnd);
Console.WriteLine("Press any key to continue or
Q to quit.\n");
cki = Console.ReadKey(true);
if (cki.Key == ConsoleKey.Q)
   return;
```

The result at this point could look like this (no data values present):



file:///D:/PNRFReader/C#/bin/Debug/Ex1 CSharp PNRF Load.EXE

```
pnrf_example_dual
Y unit: Volt
Start time: 78.4333333 s, End time: 92.7733333 s

Press any key to continue or Q to quit.
```

Figure G.1: DOS window of the CSharp PNRF Load.exe

We continue by connecting to the data an investigating the number of segments. Quit when there is no data, or when no segments are found.

```
// create data array as object
object mySegmentData = null;
// fetch data
mySource.Data(dStart, dEnd, out
mySegmentData);
if (mySegmentData == null)
   Console.WriteLine("No Data.");
   return;
}

// convert object into segment information
IDataSegments mySegments = mySegmentData as
   IDataSegments;

int iSegIndex = 1;              // segment index
int iCount = mySegments.Count; // number of
                                // segments
if (iCount < 1)
{
   Console.WriteLine("No segments found.\n");
return;
}
```

At this point we can loop through all available segments and display the data. Before we actually display the data we display the number of data points and give the option to continue or skip the segment.

```
// loop through all available segments
for (iSegIndex = 1; iSegIndex <= iCount;
iSegIndex++)
{
     // create a single segment
     IDataSegment mySegment =
     mySegments[iSegIndex];
     int iCnt = mySegment.NumberOfSamples;
Console.WriteLine("\nSegment {0}:
     {1} samples\n",
     iSegIndex, iCnt);
Console.WriteLine("Press any key to continue or
     S to skip");
cki = Console*.ReadKey(true);
if (cki.Key == ConsoleKey.S)
     continue;

// create object to hold segment data
object varData;
// fetch data
mySegment.Waveform(DataSourceResultType.
     DataSourceResultType_Double64, 1, iCnt,
     1,
     out varData);

if (varData == null)
{
     Console.WriteLine("No valid data
     found.");
     return;
}

// convert object to actual double values
double[] dSamples = varData as double[];

double X0 = mySegment.StartTime;
double DeltaX = mySegment.SampleInterval;
double X, Y;

for (int i = 0; i < dSamples.Length; i++)
{
     X = X0 + i * DeltaX;
     Y = dSamples[i];
     Console.WriteLine("{0}: X = {1}, Y =
     {2}",
        i+1, X, Y);
}
```
```
}
```

```
            Console.WriteLine("\nDone. Press any key to
            quit.");
            Console.ReadKey();
        }
    }
}
```

This concludes our C# example.

H NRF Files

H.1 Introduction

The PNRF Reader SDK also supports reading Odyssey/Vision NRF files. Reading an NRF file requires changing a single line of code in the various code examples.

Visual Basic

```
FromDisk = New RecordingLoaders.NRFLoader
```

MATLAB

The Progid for the NRF loader is ‘Perception.Loaders.NRF’.

```
>> FromDisk = actxserver('Perception.Loaders.NRF')
```

C#

```
   // create a nrfloader
   IRecordingLoader FromDisk = new NRFLoader()  ;
```

C++

```
   IRecordingLoaderPtr itfLoader;
   itfLoader.CreateInstance(__uuidof (NRFLoader));
```

I LRF Files

I.1 Introduction

The PNRF Reader SDK also supports reading Dimension LRF files. Reading an LRF file requires one or two changes in the various code examples depending on the programming tools in use.

Visual Basic

Add a reference to C:\Program Files\Common Files\HBM\Components \LRFLoader.dll

Use the following code to create an instance of the CLRFRecordingLoader

```
FromDisk = New PerceptionLrfLoader.CLRFRecordingLoader
```

MATLAB

The Progid for the NRF loader is ‘Perception.Loaders.LRF’.

```
>> FromDisk = actxserver('Perception.Loaders.LRF')
```

C#

Add a reference to C:\Program Files\Common Files\HBM\Components \LRFLoader.dll

Use the following code to create an instance of the CLRFRecordingLoader

```
   // create a nrfloader
   IRecordingLoader FromDisk = new CLRFRecordingLoader();
```

C++

To reference the LRF loader the following #import statement must be added.

```
// #import "LRFLoader.dll" no_namespace

#import"libid:8098371E-98AD-0069-BEF3-21B9A51D6B3E"
no_namespace
```

Use the following code to create an instance of the CLRFRecordingLoader

```
   IRecordingLoaderPtr itfLoader;
   itfLoader.CreateInstance(__uuidof
(CLRFRecordingLoader));
```

Index

Head Office
HBM
Im Tiefen See 45
64293 Darmstadt
Germany
Tel: +49 6151 8030
Email: info@hbm.com

France
HBM France SAS
46 rue du Champoreux
BP76
91542 Mennecy Cedex
Tél:+33 (0)1 69 90 63 70
Fax: +33 (0) 1 69 90 63 80
Email: info@fr.hbm.com

Germany
HBM Sales Office
Carl-Zeiss-Ring 11-13
85737 Ismaning
Tel: +49 89 92 33 33 0
Email: info@hbm.com

UK
HBM United Kingdom
1 Churchill Court, 58 Station Road
North Harrow, Middlesex, HA2 7SA
Tel: +44 (0) 208 515 6100
Email: info@uk.hbm.com

USA
HBM, Inc.
19 Bartlett Street
Marlborough, MA 01752, USA
Tel : +1 (800) 578-4260
Email: info@usa.hbm.com

PR China
HBM Sales Office
Room 2912, Jing Guang Centre
Beijing, China 100020
Tel: +86 10 6597 4006
Email: hbmchina@hbm.com.cn

measure and predict with confidence

HBM

I2697-1.1 en