DSP Master for MtxVec v5

Users Guide for Delphi Win32

DSP Master for MtxVec version v5, rev 2.0

www.dewresearch.com

1 Displaying the signal

Objective: We have a signal stored in the file and would like to display its contents on the chart.

1.1 Simple start

1. Start a new project and put TSignalRead and TSignalChart on the Form.
2. Add a TFastLineSeries to the Chart, you can do that by double clicking the chart and select Add Series.
3. Expand the TSignalChart.Signals property.
4. Add a new Item to the list and specify SignalRead1 as the Input and Series1 for the series property. (Figure 1) Once complete close the popup window.

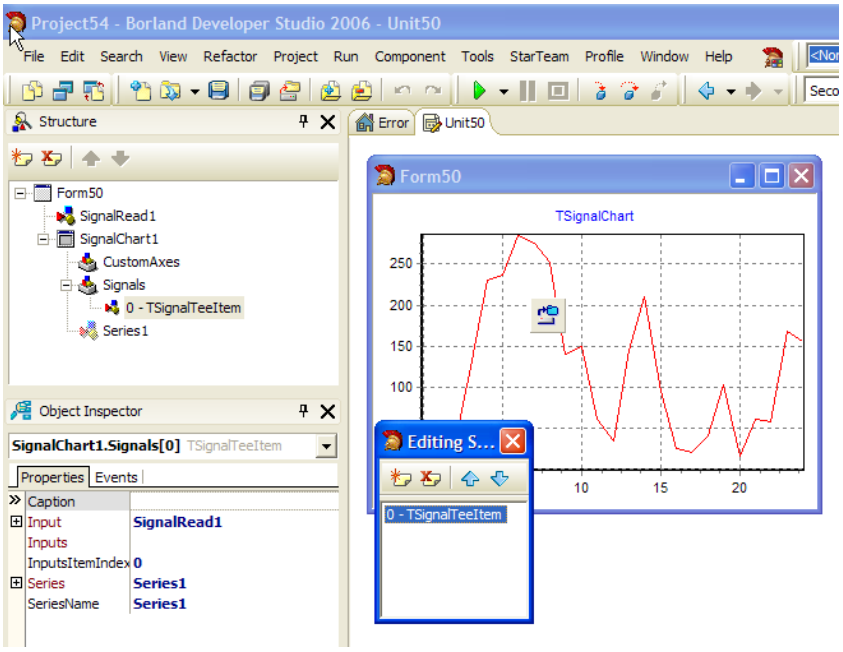


Figure 1 Connecting TSignal with TSignalChart

5. Set TSignalRead.FileName property to the bz.sfs filename in the DSP Master examples folder.
6. Add the following line to the Forms on create event:

```
procedure TForm50.FormCreate(Sender: TObject);
begin
    SignalRead1.Update;
end;
```

6. Press F9 to run application. Once started, the chart would be showing the first 128 samples or so of the signal.

Discussion:
The program has read data from the file and displayed it on the chart. SignalRead1.Data TVec object holds that data. You can access SignalRead1.Length individual values via SignalRead1.Data[i] property.

1.2 Browsing the signal

Now we will add the application the ability for the user to scroll or browse through the signal.

1. Allign the SignalChart1 to alTop, add a TPanel to form and align it to alBottom. Then align SignalChart1 to alClient. This gives us space to put user controls on the form.
2. Put two TMxFloatEdit controls on the new panel and name them PositionEdit and SamplesEdit. Add labels in front of them. One should say Position and the other Samples.
3. Set TMtxFloatEdit.IntegerIncrement to True and TMtxFloatEdit.ReFormat to 0 for both. Set SamplesEdit.Value to 256.

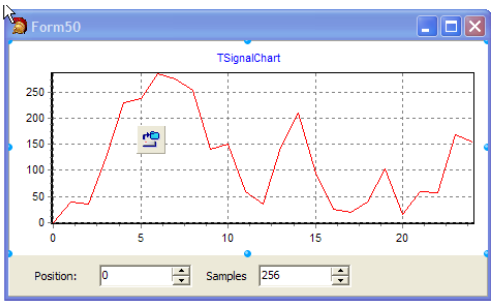


Figure 2 Adding browse controls

4. Now definie PositionEdit.OnChange and SamplesEditOnChange events like this:

```
procedure TForm50.PositionEditChange(Sender: TObject);
begin
    SignalRead1.RecordPosition := PositionEdit.IntPosition;
end;

procedure TForm50.SamplesEditChange(Sender: TObject);
begin
    SignalRead1.Length := SamplesEdit.IntPosition;
    Signalread1.Update;
end;
```

5. Press F9 to compile and run the project.
6. Try out the new controls. Put the cursor inside of them and use the Up and Down buttons.
7. Press and Hold CTRL and double click inside the edit control. A window will be displayed:

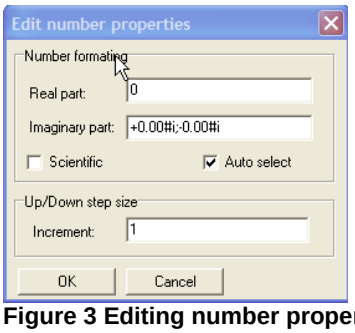


Figure 3 Editing number properties

Here we can specify the step and number formatting. Set Increment to 10, press OK and try to use the Up/Down buttons on the Keyboard.

8. Change the OnFormCreate event like this:

```
procedure TForm50.FormCreate(Sender: TObject);
begin
    SignalRead1.Update;
    Caption := FormatFloat('Sampling frequency = 0.00Hz',SignalRead1.SamplingFrequency);
end;
```

8. Double click on the chart and specify a Title for the bottom Axis: Time [s]

Discussion:
SignalRead1 contains many properties that describe the signal beside the SamplingFrequency property. See the help file for their description. Note that setting SignalRead1.RecordPosition automatically calls SignalRead1.Update.

1.3 Reading multi-channel files

1. Find a two channel (stereo) wav file on your disk and assign it to SignalRead1.FileName property.
2. Place TSignalDemux component on the Form and set its Input property to SignalRead1.
3. Expand the SignalChart1.Signals property and change the Input of the first Item from SignalRead1 to SignalDemux1.
4. Modify the events like this:

```
procedure TForm50.FormCreate(Sender: TObject);
begin
    SignalRead1.Update;
    SignalDemux1.Update;
    Caption := FormatFloat('Sampling frequency = 0.00Hz',SignalRead1.SamplingFrequency);
end;

procedure TForm50.PositionEditChange(Sender: TObject);
begin
    SignalRead1.RecordPosition := PositionEdit.IntPosition;
    SignalDemux1.Update;
end;

procedure TForm50.SamplesEditChange(Sender: TObject);
begin
    SignalRead1.Length := SamplesEdit.IntPosition;
    Signalread1.Update;
    SignalDemux1.Update;
end;
```

5. Press F9 and run the application. The application is now showing the left channel of the two channel wav file.

Discussion:
We could easily place two SignalDemux1 components on the Form. SignalDemux1.Channel property specifies the channel number to demultiplex from the stream. In general however, the channel count can be any number and for that reason we need a list whose item count can adjust automatically to SignalRead1.ChannelCount.

1.4 Showing two channels

1. Delete SignalDemux1 component and Place TSignalDemuxList component on the Form.
2. Assign SignalRead1 to SignalDemuxList1.Input property.
3. Add a second TFastLineSeries to the SignalChart1.
4. Expand the SignalChart1.Signals property and add a new (second) item to the list:

a.) Set the Inputs of the first item to SignalDemuxList1.
Set the InputsItemIndex property of the second Item to 0. This means channel 0 in the list of channels.
b.) Set the Inputs of the second item to SignalDemuxList1.
Set the InputsItemIndex property of the second Item to 1. This means channel 1 in the list of channels.
Assign Series2 to the Series property.

5. Set SignalDemuxList1.Count to 2.

6. Modify events like this:

```
procedure TForm50.FormCreate(Sender: TObject);
begin
    SignalDemuxList1.Pull;  //match the list count to the channel Count
    Caption := FormatFloat('Sampling frequency = 0.00Hz',SignalRead1.SamplingFrequency);
end;

procedure TForm50.PositionEditChange(Sender: TObject);
begin
    SignalRead1.RecordPosition := PositionEdit.IntPosition;
    SignalDemuxList1.Update;
end;

procedure TForm50.SamplesEditChange(Sender: TObject);
begin
    SignalRead1.Length := SamplesEdit.IntPosition;
    Signalread1.Update;
    SignalDemuxList1.Update;
end;
```

7. Press F9 to run the application. You can now see two channels displayed in the chart.

Discussion:
The Pull method will call Update for all the components chained together with their Input/Inputs properties. In our case it will first call SignalRead1.Update and then adjust the SignalDemuxList.Count property to match the Signalread1.ChannelCount. It will then call first SignalDemuxList[0].Update and then SignalDemuxList[1].Update.

Assuming you have a graph connected components, the picture shows how the Pull method will progress.

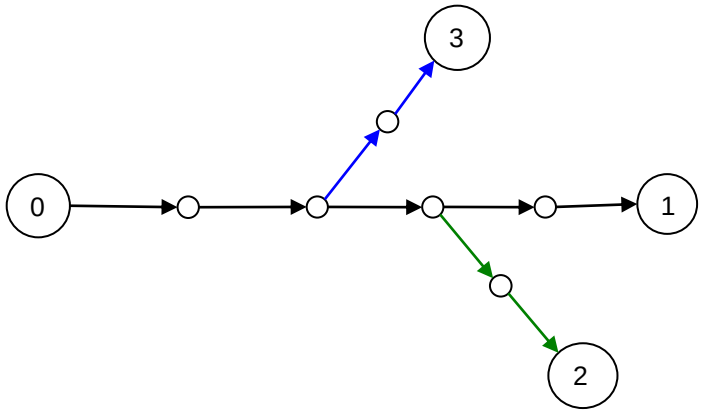


Figure 4 Pull progression graph

Each circle represents one component and each arrow shows that previous component in line is assigned to the Input property of the next component in line. When you call Pull method of component "1", this will cause recalculation of all components from 0 to 1, starting at 0 (black arrows). When the Pull method is then called for components "3" and "2", their recalculation request (blue and green) will not go pass the black arrows, because those components have already been updated.

If however, component "2" would call Pull once more, the recalculation request would progress until component "0" and Pull request from components "1" and "3" would only update those parts of the graph that has not yet been updated.

The graph can consist from lists of components at each node. This means that you can concurrently process arbitrary number of channels without knowing in advance how many channels the source will have.

1.5 Showing two channels second example

In the previous example, we had to manually set the SignalDemuxList1.Count. This was necessary to allow the SignalChart1 to know at the time when the Inputs property was assigned the count of channels. If that would not be the case, we have to let the SignalChart1 know that something has changed:

```
procedure TForm50.FormCreate(Sender: TObject);
begin
   SignalDemuxList1.Pull;  //match the list count to the channel Count
   SignalChart1.Signals.UpdateInputs;
   SignalDemuxList1.UpdateNotify;
   Caption := FormatFloat('Sampling frequency = 0.00Hz',SignalRead1.SamplingFrequency);
end;
```

UpdateInputs will check SignalDemuxList1.Count and reconnect individual items to the SignalChart1 by using the SignalChart.Signals[i].InputsItemIndex property. The UpdateNotify method can be called always when we know that the component is connected to some chart and when we would only like to update the chart and not also trigger recalculation. The Update method namely triggers recalculation of the component and updates any associated charts.

1.6 Summary

- The Pull method will call Update for all components in the direct chain, but not in the branches.
- The Update method will fetch whatever data the component connected to Input property has, recalculate that data according with its function, place the result in its own Data property and notify any associated charts to update the display.
- The UpdateNotify method will not trigger recalculation, but only notify associated charts, that new data is to be displayed.
- If we would like to call the Pull method, but not update the charts, there is a special SuspendNotifyUpdate property that can be set to false.

2 Frequency analyzer

When dealing with signals one of the first things we would like to look at is the frequency spectrum. We continue with the project from the previous chapter.

2.1 Dual channel frequency analyzer

1. Take the project from the previous chapter and add TSpectrumChart at the top and one TSplitter between TSignalChart and TSpectrumChart. TSpectrumChart is alTop aligned, TSignalChart is alClient and the splitter is alTop.
2. Add two TFastLineSeries to the TSpectrumChart.
3. Insert TSpectrumAnalyzerList in the form.
4. Add two items to SpectrumChart.Spectrums property and make the following connections:

a.) Set the Inputs of the first item to SpectrumAnalzyerList1.
Set the InputsItemIndex property of the second Item to 0. This means channel 0 in the list of channels.
Assign Series3 to the Series property.
b.) Set the Inputs of the second item to SpectrumAnalyzerList1.
Set the InputsItemIndex property of the second Item to 1. This means channel 1 in the list of channels.
Assign Series4 to the Series property.

See Figure 5 on how the form should be looking now.

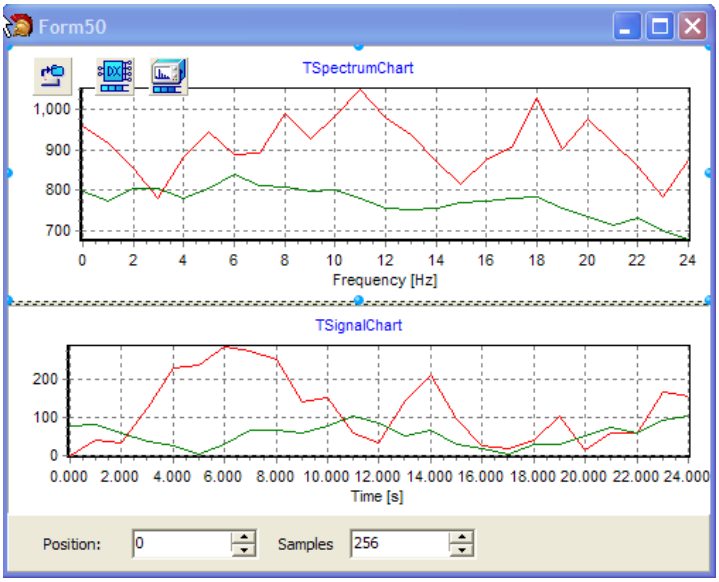


Figure 5 Frequency analyzer window

5. Modify the events like this:

```
procedure TForm50.FormCreate(Sender: TObject);
begin
    SpectrumAnalyzerList1.Pull;  //match the list count to the channel Count

    SignalChart1.Signals.UpdateInputs;
    SignalDemuxList1.UpdateNotify;

    SpectrumChart1.Spectrums.UpdateInputs;
    SpectrumAnalyzerList1.UpdateNotify;

    Caption := FormatFloat('Sampling frequency = 0.00Hz',SignalRead1.SamplingFrequency);
end;
```

```
procedure TForm50.PositionEditChange(Sender: TObject);
begin
    SignalRead1.RecordPosition := PositionEdit.IntPosition;
    SpectrumAnalyzerList1.Pull;
end;

procedure TForm50.SamplesEditChange(Sender: TObject);
begin
    SignalRead1.Length := SamplesEdit.IntPosition;
    SignalRead1.Update;
    SpectrumAnalyzerList1.Pull;
end;
```

4. Press F9, recompile and run the application. Change the Samples and Position edit boxes to see how the signal looks.

Discussion:
We call Signalread1.Update first and then SpectrumAnalyzer1.Pull. Shouldn't Pull also call SignalRead1.Update? Then answer is no, because the Update method notified all connected components that it has already updated.

If the Pull method would be called twice, the second call would reach SignalRead1, which would advance the cursor position in the file and read the next frame. You could Pull until the return value of the function would be equal to pipeEnd, which would signal that the end of the file has been reached.

When we change the number of samples to be displayed however we do not want to advance the read position in the file and that it is why we still call SignalRead1.Update before calling Pull. Figure 6 shows that Pull only progressed until SignalDemuxList1. The green arrow shows the part that was already done when we called SignalRead1.Update. The SignalDemuxList1.Update and SpectrumAnalyzerList1.Update only were called by the Pull method.

In case of PositionEdit remember that setting SignalRead1.RecordPosition also calls Update.

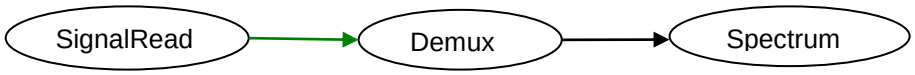


Figure 6 Pull progression graph

2.2 Adding peak marking

1. Double click on the SpectrumChart1 and add a new TPointSeries to the list of the series.
2. Select the tools panel and add a new Spectrum mark tool to the tools panel.
3. Fill in the parameters of the Spectrum mark tool, by setting the Spectrum to Series4 and Mark series to Series5 (TPointSeries).
4. Close the TeeChart editor and run the app.

Although you can see the peak and apply marks, there are is still some fine tunning necessary. Start TeeChart editor again and:

5. TPointSeries: set Style to circle. Width and Height to 3 pixels. Marks->Style panel, check the Visible box, Marks->Format panel, check the Transparent box. Marks->Arrows, set distance to 10.
6. BottomAxis: Labels->Style, set minimum separation to 0% and Style to Text. Minor->Ticks, Visible to false.
7. Close the TeeChart editor and run the app.

Discussion:
Note that you can click the marked peaks again to unmark them and if you want to remove them you can double click the chart. The zoom (click and drag down) and pan (right click and drag) also remain functional.

2.3 Adding user dialogs and editors

1. Insert TMainMenu in the Form and add 6 items: Edit chart > (Top Chart, Bottom Chart) and Edit Spectrum ->(Left channel, Right Channel).
2. Insert TSpectrumAnalyzerDialog and TCharEditor.
3. Add the following OnClick events for each menu item:

```
procedure TForm50.opchart1Click(Sender: TObject);
begin
    ChartEditor1.Chart := SpectrumChart1;
    ChartEditor1.Execute;
end;

procedure TForm50.Bottomchart1Click(Sender: TObject);
begin
    ChartEditor1.Chart := SignalChart1;
    ChartEditor1.Execute;
end;

procedure TForm50.Leftchannel1Click(Sender: TObject);
begin
    SpectrumAnalyzerDialog1.SourceListIndex := 0;
    SpectrumAnalyzerDialog1.Execute;
end;

procedure TForm50.Rightchannel1Click(Sender: TObject);
begin
    SpectrumAnalyzerDialog1.SourceListIndex := 1;
    SpectrumAnalyzerDialog1.Execute;
end;
```

4. Add the OnParameterUpdate event to SpectrumAnalyzerList1. This even is triggered when the users presses OK on the Spectrum Analyzer dialog. The Sender is the TSpectrumAnalyzer object being edited.

```
procedure TForm50.SpectrumAnalyzerList1ParameterUpdate(Sender: TObject);
begin
    TSpectrumAnalyzer(Sender).Update;
end;
```

5. Press F9 and run the application.

Discussion:
Try running the chart editors. You can change the series and peak mark tool parameters. When running the Spectrum Analyzer dialogs, you have to option to specify the dialog to be "Live" from the bottom Options menu. When "Live" all changes to the settings are immediately visible to on the charts. Some parameters displayed in the dialog are not always applicable and that is why TSpectrumAnalyzerDialog.TabsVisible property is there.

3 Recording

Recording is supported for any channel count, sampling frequency and bit depth supported by the underlying hardware.

3.1 Simple monitoring of the recording signal

This example will show how you can display the left and right recorded channel on the same chart in real time. The data is not written to the disk.

1. Start a new project and put one TPanel on the form and align it to the bottom. Put two buttons on it, one labeled Start and one Stop. Then put one TSignalChart on the Form and set it's Align property to alClient.
2. Insert one TSignalIn component , one TSignalList component and one TSignalTimer.

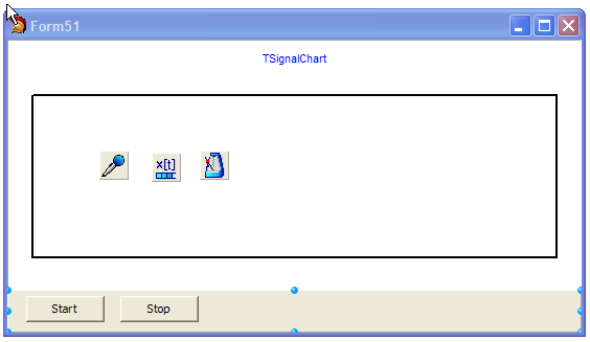


Figure 7 Signal monitor form

3. Set TSignalTimer.Frequency to 10 and fill in its OnTimer even like this:

```
procedure TForm51.SignalTimer1Timer(Sender: TObject);
begin
    SignalList1[0].Length := 1024;
    SignalList1[1].Length := 1024;
    SignalIn1.MonitorData(Signalist1[0], SignalList1[1]);
end;
```

3. Define the OnClick events for the start and stop button:

```
procedure TForm51.Button1Click(Sender: TObject);
begin
   SignalIn1.Start;
   SignalTimer1.Enabled := True;
end;

procedure TForm51.Button2Click(Sender: TObject);
begin
    SignalIn1.StopAtOnce;
    SignalTimer1.Enabled := false;
end;
```

4. Double click the TSignalChart and add two new TFastLineSeries: Series1 and Series2.

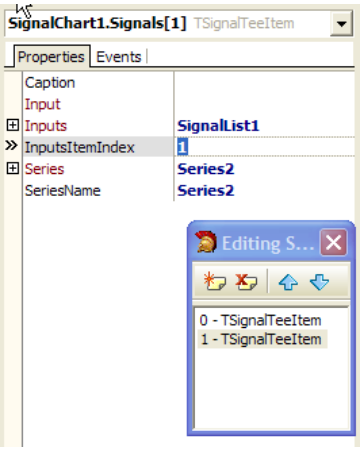


Figure 8 Connecting to chart

Connect the Series to TSignalChart.Signals property. Series2 to SignalChart1.Signals[1] and Series1 to SignalChart1.Signals[0]. SignalChart1.Signals[1].InputsItemIndex should be 1.

5. Set SignalList1.Count to 2 and SignalIn1.ChannelCount to 2.
6. Press F9 and run the application. Press the start button. You should be seeing two wav channels being streamed on the chart.

Discussion:

If your audio hardware allows it, try changing the bit depth to 24bit, by setting SignalIn1.Quantization to 24. When increasing the sampling frequency or the length of the monitored signal (SignalList1[0]) be carefull to also increase the SignalIn1.BufferSize. BufferSize*BufferCount in bytes should be approximately 2 seconds of recording with given channelCount, Bit depth and sampling frequency. If not, the recording could be skipping.

3.2 Monitoring and recording to file

We will upgrade the existing monitoring project by adding the "record to file" feature.

1. Insert TSignalWrite component.
2. Assign SignalWrite1.Input := SignalIn1;
3. Set SignalWrite1.Precision := prSmallInt; We must be carefull that SignalWrite precision matches SignalIn1.Quantization.
4. Specify SignalWrite1.FileName, and choose a file with a .wav extension to record to.
5. Fill-in the SignalIn1.OnBufferFilled event like this:

```
function TForm51.SignalIn1BufferFilled(Sender: TObject): Boolean;
begin
    SignalWrite1.Pull;
    Caption := FormatFloat('Written: 0 [KB]', SignalWrite1.BytesWritten/1024);
    Result := True;
end;
```

6. Modify the OnClick event for the Stop button:

```
procedure TForm51.Button2Click(Sender: TObject);
begin
    SignalIn1.StopAtOnce;
    SignalTimer1.Enabled := false;
    SignalWrite1.CloseFile;
end;
```

We have to close the file once the recording has finished, or the file be written from the point where it stopped each time the Start is pressed again.
7. Press 9 and run the application. Press Start.

Discussion:
The TSignalWrite is now directly connected to the TSignalIn. This does not give us the ability to process the signal, because the signal is multiplexed in two channels. If we would want the recorded signal to pass through a digital filter, we would have insert TSignalDemuxList, TSignalFilterList and TSignalMuxList in between.

4 Playback

This chapter explains how to setup the playback from a file and from the memory and an example on how to vary the playback speed.

4.1 Playback monitoring

1. Start a new project and put one TPanel on the form and align it to the bottom. Put two buttons on it, one labeled Start and one Stop. Then put one TSignalChart on the Form and set it's Align property to alClient.
2. Now insert TSignalOut, TSignalRead, TSignalList and TSignalTimer components.

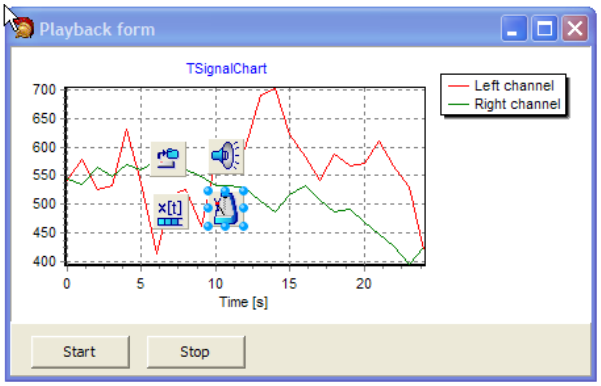


Figure 9 Playback monitoring

3. Set SignalTimer1.Frequency to 10.
4. Set SignalOut1.Input = SignalRead1;
5. Assign a valid stereo .wav file to SignalRead1.FileName
6. Fill-in the OnClick events of Start and Stop buttons and OnTimer event:

```
uses MtxVecTee;

{$R *.dfm}

procedure TForm51.Button1Click(Sender: TObject);
begin
    SignalRead1.Length := 2048;
    SignalOut1.Start;
    SignalTimer1.Enabled := True;
end;

procedure TForm51.Button2Click(Sender: TObject);
begin
    SignalTimer1.Enabled := False;
    SignalOut1.StopAtOnce;
end;
```

```
procedure TForm51.SignalTimer1Timer(Sender: TObject);
begin
    SignalList1.Count := SignalRead1.ChannelCount;
    SignalList1[0].Length := 1024;
    SignalList1[1].Length := 1024;
    SignalOut1.MonitorData(SignalList1[0],SignalList1[1]);
    DrawValues(SignalList1[0].Data,Series1,0,SignalOut1.Dt);
    DrawValues(SignalList1[1].Data,Series2,0,SignalOut1.Dt);
end;
```

Discussion:
Before calling SignalOut1.Start, we set SignalRead1.Length to some rather large value. This number also defines the size of the playback buffer inside TSignalOut. If the buffer is too small, we get clicks or completely distorted sound.

We set SignalList.Count to the number of channels in the file and the length of individual items to the number of samples that we would like to see on the chart. This time the data makes it to the chart via DrawValues routine and not by make use of the SignalChart1.Signals property.

One other parameter that is not automatically transferred is the bit depth of the file being played back. If the file would have had 24bit resolution, the signal would sound distorted. That's why:

```
SignalOut1.Quantization := SizeOfPrecision(Signalread1.Precision,false)*8;
```

is necessary before the playback starts. But, if the hardware does not support 24bit resolution, additional scaling must be performed to prevent clipping. This scaling can be achieved via SignalRead1.ScaleFactor property.

4.2 Variable playback speed, tone generator

1. We start we the project from the previous chapter and add TSignalGeneratorList, TSignalRateConverterList and TSignalBufferList to the form. Delete Signalread1 component and connect the components (at design time):

SignalRateConverterList1.Input := SignalGeneratorList;
SignalBufferList1.Input := SignalRateConverterList1;
SignalMux.Inputs := SignalBufferList1;
SignalOut1.Input := SignalMux;

Signal generator will generate two tones, one on each channel, TSignalRateConverter will change the sampling frequency of the signal and thus the playback speed. TSignalBuffer will make sure that the buffer length after the rate converter has a fixed size that can be used by TSignalOut. The rate converter namely outputs Input.Length*Factor number of samples. This means possibly non-integer sample count on every iteration and consequently varying output Data.Length sample count. TSignalMux will multiplex both channels together and thus prepare the signal for TSignalOut.

TSignalTimer will still perform the same function as before. Namely to update the charts with just to be played back data.

5 Batch file processing

Sometimes you want to apply a specific signal processing operation to a signal in a file and store the result back in to the file. Rate conversion, digital filtering, envelope detection, noise reduction and similar operations come to mind. This chapter shows how you can make such algorithms Channel Count independent, how to automatically preserve the file format and a method to monitor the progress of the processing. All you are left to specify is the actual computing logic itself.

//Unfinished. Work to do…

6 Inner workings

The signal processing components in DSP Master goals:

1. offer pipelined architecture and visual programming of DSP algorithms.
2. simplify the configuration of the pipeline via component editors.
3. offer a quick access to most of the features of the DSP Master to give the user a good overview.
4. simplify the multi-channel processing.

The components can be used in two ways:

- as signal processing blocks connected in to pipes.
- as servers returning data processed according to their property settings.

The components offer a very unique capability to switch between different modes of programming: visual and non-visual. In most classic visual programming enviornments like LabView or Simulink, the programmer has to write its own component in order to be able to use it chained in the pipeline. DSP for MtxVec allows the user to programm different parts of the algorithm in whatever is the most natural and easy way to do it. While working with the components, I must admit that after connecting the components up, it was very appealing to just continue programming in the same manner. There are cases however when there was no way around and some things had to be programmed in a traditional way.

DSP for MtxVec does not offer components for many basics operations like: Multiply, divide etc.... Most components are designed to get the user over the "hard" stuff, and not to completely replace the code editor. (Some things can be become really tedious when streaming is a must).

Perhaps most important: Usually Delphi users dive in the components and try to get everything done just be doing a few clicks. The main strength of DSP Master is its underlaying routine based library. It is recommended that user should first try and write a lines of code using the routine based library and then try to make the best out of components. Most components simply encapsulate the underalying library. Some components are more and others for a less general purpose. One should not try to make a "one fits all" attempt. If the component does fit the need, write your own and use the existing ones as an example. In many cases it is possible to get away by writing just a few lines of code.

6.1 Serializing and streaming

All components are derived from a common ancestor: TMtxComponent. This component provides routines like:

- SaveToStream
- LoadFromStream
- SaveToFile
- LoadFromFile
- Assign
- AssignTemplate
- LoadTemplateFromStream
- SaveTemplateToStream
- LoadTemplateFromFile
- SaveTemplateToFile

First five routines are known to most Delphi/CBuilder developers. What is interesting about TMtxComponent is that all components derived from it have all their published properties streamed, without the need to make any changes to the five routines. Therefore, all components of the DSP for MtxVec have the capability to store their states (properties) to the stream, file or to assign from another object of the same type. The default component streaming mechanism has a bug when it comes streaming properties with a declared default value. This has been fixed for TMtxComponent.

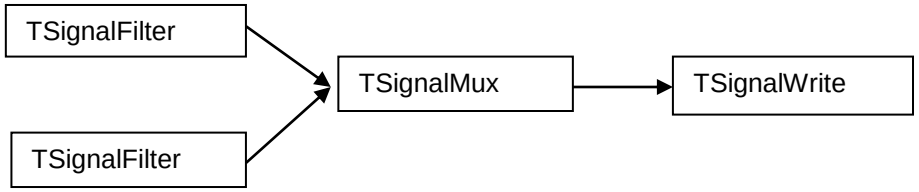
The "template" routines are a bit more advanced. They set a special protected property named BlockAssign to True. Property setter routines can then prevent properties to be changed. This is very usefull when there is a need to save only "parameters" and not the "data" of the component. The parameters will remain the same, while the "data" will be different next time and there is no point in wasting disk space by saving it.

7 Classes

There are two major subclass systems which derive from TMtxComponent:

- TSignal
- TSpectrum

Components derived from TSignal have an Input and/or Inputs property and can be connected in to signal processing chains:



The result of processing of each TSignal component is placed in the Data property. Data property is of TVec type. The Input property points to the first element of the Inputs property (TCollection type). The component can accept many Inputs, but not all components are able to make use of them. An example of a TSignal component accepting many Inputs is TSignalMux, which multiplexes inputs in to one signal.

- To request recalculation of all connected components call the **Pull** method.
- To request recalculation of only the selected component use the **Update** method.
- To request recalculation of only a part of the component chain the, Streaming properties should be set appropriately. (more in "Streaming properties").

7.1 Signal processing properties

Each TSignal component holds the description of the data which it holds. The key signal processing properties are:

- Length – defines the length of Data vector,
- Complex – if True, the data vector is complex,
- ChannelCount – the number of mulitplexed channels stored in the Data vector.
- SamplingFrequency – sampling frequency of the signal

The values of these properties are propagated from the first component in the signal processing chain to the last. Each component in the chain can of course also change the values of these properties, if so required. As it already became obvious, the signal processing chains are block based. This means that

each component will take data from the Data property of the component connected to its input, process it and place the result in its own Data property. The size of the data block is defined with the first component in the chain. Good sizes are from about 50 to about 2000 values. The size of the data block should not exceed the size of CPU cache, or the performance will suffer. Signal processing chains based on block processing are significantly faster then single sample based chains, because they can take advantage of the SSE (P3) and SSE2 (Pentium 4) instructions sets. Single sample processing chains on the other hand are simpler to use. There are several cases when the block size has to be changed on the fly. Cases like this include:

- Multi-rate, multi-stage decimation, Interpolation, demodulation and modulation. When the change of the sampling frequency is large (in case of decimation or demodulation can the sampling frequency change by 1000x), the input buffer has to be large enough, so that the output will result in integer number of samples. The input buffer therefore has to be increased and the output buffer also in order to match the recommended buffer size for further processing.
- Changes in sampling frequency by an arbitrary real factor. (for example from 96kHz to 44.1kHz) In such cases the buffer size again has to be increased, but this time the increase is not fixed. The required input buffer size changes with each iteration, because the input sampling rate is not divisible with output sampling rate.

There is a special component called TSignalBuffer which is designed to help manage the buffering problems. Its Length property defines the desired output block size and until that much data is ready no recalculation requests will be passed to its connected components. If more data is ready, no more data will be fetched until the current buffer has been emptied. If it is not necessary to change the sampling frequency within the processing pipe, the TSignalBuffer component is usually not needed.

7.2 Properties for pipeline control

Another important set of properties are those which control the pipelined streaming of the data. When will a component be updated, when will new data be fetched etc... These properties are:

- Active
- Continuous
- Dirty
- UsesInputs
- SuspendNotifyUpdate

When a recalculation request is received, the request will be not be executed unless Active and Dirty are True. The Active property is to be disabled when it is necessary to disable both:

- fetching of new data
- recalculation of data.

This will be disabled only for the component for which the Active property is false. The Pull method will return true and all subsequent recalculations will perform as usually. The components connected to the Input property will not receive a Pull request. The Dirty property is similar to Active, except that it's purpose is to prevent recalcuation with the same data. For example two TSignalDemux components are connected to the TSignalRead to demultiplex two channels from the TSignalRead.Data property. When the first TSignalDemux passes a recalculate request to TSignalRead, the TSignalRead will read the data from the file. There should be no second call to load the data again for the second channel. That's why the call to recalculate (Update) sets Dirty to false. The Dirty property is set to True by the Pull method.

Continuous should be set to false when only fetching of new data should be omitted, but the component should recalculate (and place the result in its Data property). If the component requires components

connected to its Inputs slot, then the UsesInputs should be set to True. The UsesInputs property is protected.

SuspendNotifyUpdate is similar to the Dirty property. It is to prevent a call to the OnNotifyUpdate event. That event is used to update the Chart's or the form. One such example is averaging. The display should only be updated after the averaging has been completed and not for every iteration. Not setting SuspendNotifyUpdate to false, will unnecessarily drain the CPU. Some components like TCrossSpectrumAnalyzer and TBispectrumAnalyzer automatically set and reset the SuspendNotifyUpdate property when averaging.

7.3 Writing custom TSignal components

In order to be able to include a component in to the pipeline, the component has to be derived from TSignal. If its result depends on the Inputs, the UsesInputs property should be set to True in the constructor. The continuous property should also be set appropriately. A part from that, only the InternalUpdate method should be overridden. The routine should return True to indicate that Data can be further processed and False to stop the processing. The values of all other signal processing and streaming properties are being assigned automatically. The underlying routines also perform the check, if the Inputs are connected.

Example:

```
TSignalScale = class(TSignal)
protected
    function InternalUpdate: boolean; override;
end;

...

constructor TSignalScale.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    UsesInputs := True;
end;

function TSignalScale.InternalUpdate: boolean;
begin
    Data.Copy(Input.Data);
    Data.Scale(2)
    Result := True;
end;
```

The component then still has to be registered with the IDE using standard registration procedure before it can be used.

8 The dialogs

The package includes several component editors (dialogs). This dialogs can be used to give the application a quick user interface to the most of the properties of key components. They also give the user a quick access to the majority of all signal processing algorithms in the package. Each dialog is controled by four key properties:

- Continuous
- Live
- Docking
- StayOnTop

Continuous controls the TSignal.Continuous properties. This property can be very useful when a recalculation with a set of different values for one or more of the parameters is to be tried out on the same data. When Live is checked, the component will be recalculated immediately when a parameter (edit box) changes. If Docking is checked it is possible to dock the window. StayOnTop is self explanatory. A very neat feature is the adjustable increment of up/down boxes. Double click any up/down edit box to change the number formatting or its increment. If StoreInRegistry property of the associated Dialog component is True, the number formatting and the increment of all up/down edit boxes on the current dialog will be saved in to the system registry.

All dialogs also provide commands to save and load the settings of the edited component to and from a file.