

# EG Courses

## Graphical Programming Languages 1

### WORKBOOK

Revision 2.1

**GPL1**



**Graphical  
Programming  
Languages 1**



## Table of Contents

|   |    |
|---|----|
| TABLE OF CONTENTS .....                       | 2  |
| GENERAL INFORMATION.....                      | 3  |
| EXERCISE 1 HELLO, WORLD! .....                | 4  |
| EXERCISE 2 FIRST STEPS IN LABVIEW .....       | 6  |
| EXERCISE 3 FOR, WHILE AND CASE .....          | 8  |
| EXERCISE 4 BASIC DATA TYPES .....             | 11 |
| EXERCISE 5 CLUSTERS .....                     | 15 |
| SPEECH RECORDER 1 EXPRESS VIS.....            | 18 |
| SPEECH RECORDER 2 SOUND INPUT FUNCTIONS ..... | 21 |
| SPEECH RECORDER 3 FILTERING AND FFT.....      | 23 |
| SPEECH RECORDER 4 CODE CLEAN-UP.....          | 26 |
| SPEECH RECORDER 5 CONTINUOUS ACQUISITION..... | 30 |
| SPEECH RECORDER 6 FILE SAVING SUBVI .....     | 32 |
| SPEECH RECORDER 7 INTEGRATING VIS.....        | 36 |
| SPEECH RECORDER 8 STATE MACHINE .....         | 38 |
| SPEECH RECORDER 9 EVENT LOGGING .....         | 43 |



## General Information

This document provides step-by-step instructions for all Graphical Programming Languages 1 (GPL1) course exercises and projects. GPL1 is part of a series of specialized training courses collectively known as EG Courses.

EG Courses are a great opportunity to fully learn the principles of LabVIEW programming. Classes, conducted by certified LabVIEW architects, provide extensive knowledge and develop practical skills in the field of real-time systems, automation test applications, measurement monitoring, broadly understood data acquisition and many other topics related to the construction and maintenance of control systems. The modular structure of the courses allows to adjust their content to the expectations of students.

The GPL1 course provides an introduction to the graphical programming languages and allows to recognize basic code structures and mechanisms specific for graphical programming environments.

All the materials needed to complete the exercises as well as the proposed solutions are available in the GitHub repository: [www.github.com/EGCourses/GPL1](https://www.github.com/EGCourses/GPL1).

## ABBREVIATIONS

|     |                    |
|-----|--------------------|
| BD  | Block Diagram      |
| CP  | Connector Pane     |
| FP  | Front Panel        |
| LMB | Left Mouse Button  |
| RMB | Right Mouse Button |

Words like FOR, WHILE, CASE etc. refer to the corresponding programming object (for example, CASE stands for *Case Structure*, WHILE stands for *While Loop*, and so on).



## EXERCISE 1 Hello, World!

### GOAL

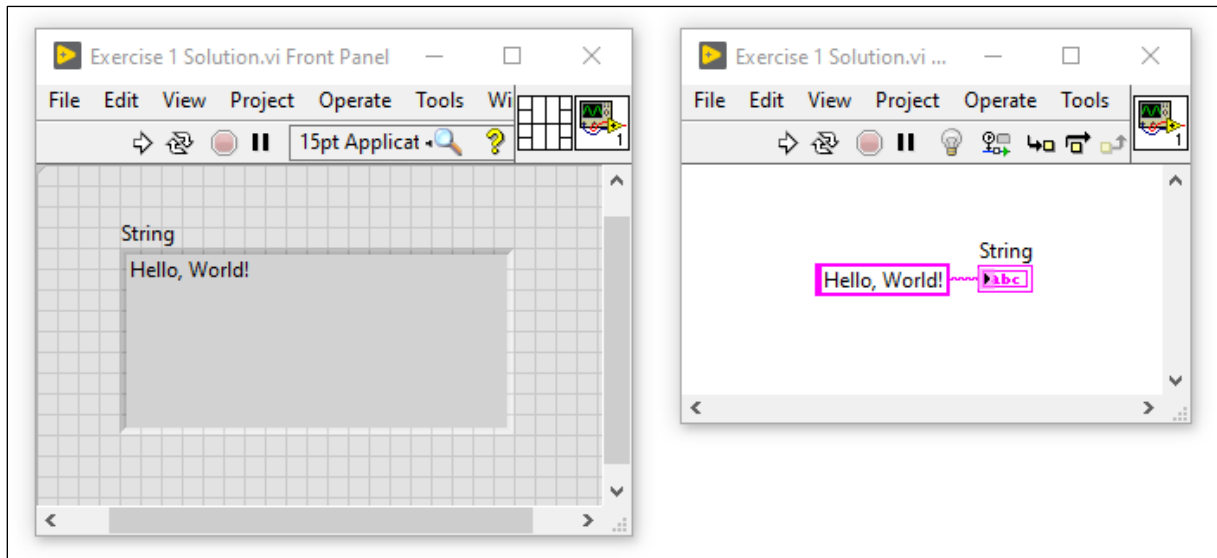
Create an application that displays “Hello, World” string on the FP.

### IMPLEMENTATION

1. Create a new VI.
  - Open LabVIEW.
  - Select *New* → *New VI* from the Menu.
  - Save the VI by selecting *File* → *Save* from the Menu or by using *Ctrl+S* shortcut.
2. Create an indicator and fill it with the content.
  - Press *Ctrl+Space* on the BD (Block Diagram).<sup>1</sup> This action allow to open Quick Drop tool.
  - In the Quick Drop window, start typing *String constant* and once the necessary item shows up, select it by double-clicking it.
  - Double-click on the string constant and fill it with the expected text.
  - Click RMB on the string constant and select *Create* → *Indicator*.
3. Run the VI
  - Click *Run* button on the toolbar of BD (the white arrow, first on the left).
  - See the result on the FP (Front Panel).

---

<sup>1</sup> If you don't see the BD window, select *Window* → *Show BD* on the FP. Similarly you can switch back to the FP from BD. You can also navigate between them by pressing *Ctrl+E*.



**END OF EXERCISE**



## EXERCISE 2 First Steps in LabVIEW

### GOAL

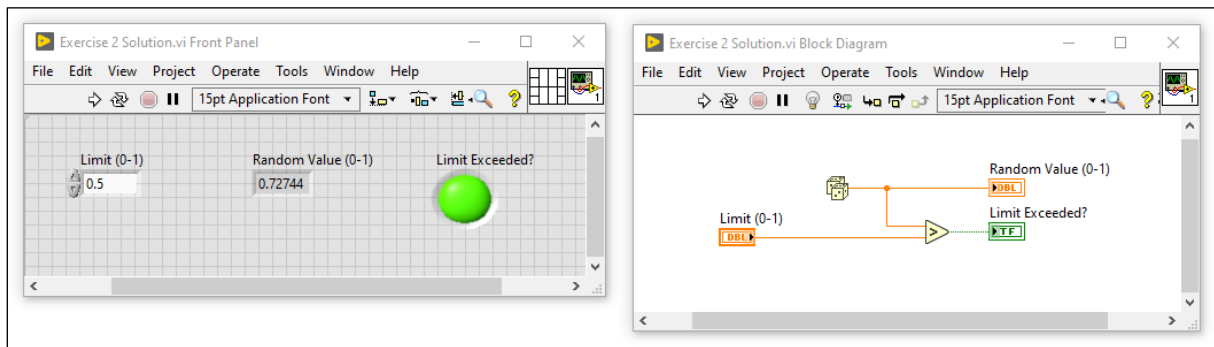
Create an application that compares randomly generated number from 0 to 1 with the value entered by the user on the FP. The application displays the generated number and whether the threshold has been exceeded.

### IMPLEMENTATION

1. Create a new VI.
  - Open LabVIEW.
  - Select *New* → *New VI* from the Menu.
  - Save the VI by selecting *File* → *Save* from the Menu or by using *Ctrl+S* shortcut.
2. Prepare controls and indicators on the FP.
  - Click RMB on the FP to open the controls and indicators palette.
  - Move to the *Numeric* section (first row, first item) and select *Numeric Control* for the threshold value.
  - Open the controls palette once again and move to the *Boolean* section (first row, second item) and select *Round LED* for the *Threshold exceeded?* indication.
  - Similarly, select *Numeric Indicator* from the *Numeric* section for displaying generated value.
  - Organize and name all objects on the FP.
3. Implement application logic.
  - Switch to the BD of your VI and find terminals corresponding to the objects created on the FP.
  - Click RMB on the BD to open the functions palette.
  - Move to the *Numeric* section (first row, last item) and select *Random Number (0-1)* function (look for the icon with two dice).
  - Connect the output of the random function to the corresponding indicator.



- On the functions palette move to the *Comparison* section (second row) and select *Greater?* function.
  - Connect the random value and the threshold value to *Greater?* function inputs.
  - Connect output of the *Greater?* function to the *Threshold Exceeded?* indicator.
4. Run the VI several times and observe the FP.



**END OF EXERCISE**



## EXERCISE 3 FOR, WHILE and CASE

### GOAL

Create an application that continuously generates a 1D array and sums its elements. The application stops execution when the sum of the items is greater than the user-defined limit. In addition, a popup should appear informing that the limit is exceeded. The size of the 1D array is user-defined and the array should contain randomly generated values.





### IMPLEMENTATION

1. Create a new VI.
2. Generate 1D array.
  - Open the functions palette on the BD and select *For Loop* from the *Structures* section (first row, first item). Draw the loop on the BD.
  - Click RMB on the *Loop Count* input (represented by the N symbol in the top left corner of the loop) and select *Create Control*. Change the control name.
  - Select *Random Number (0-1)* function and place it inside the loop.
  - Connect the function output to the loop edge in order to create a tunnel.
3. Compare array sum.
  - Select *Add Array Elements* function from *Numeric* section of the functions palette.
  - Connect generated array to the input of the summing function.
  - Place *Less?* function on the BD (you can find this function in *Comparison* section).
  - Connect array sum to the first input of the function.
  - Move to the second function input and click RMB to open the context menu. Create a control for this input and name it *Limit*.
  - Move to the wire that contains array sum value, so your cursor looks like an arrow. Then click RMB and create an indicator for this wire.




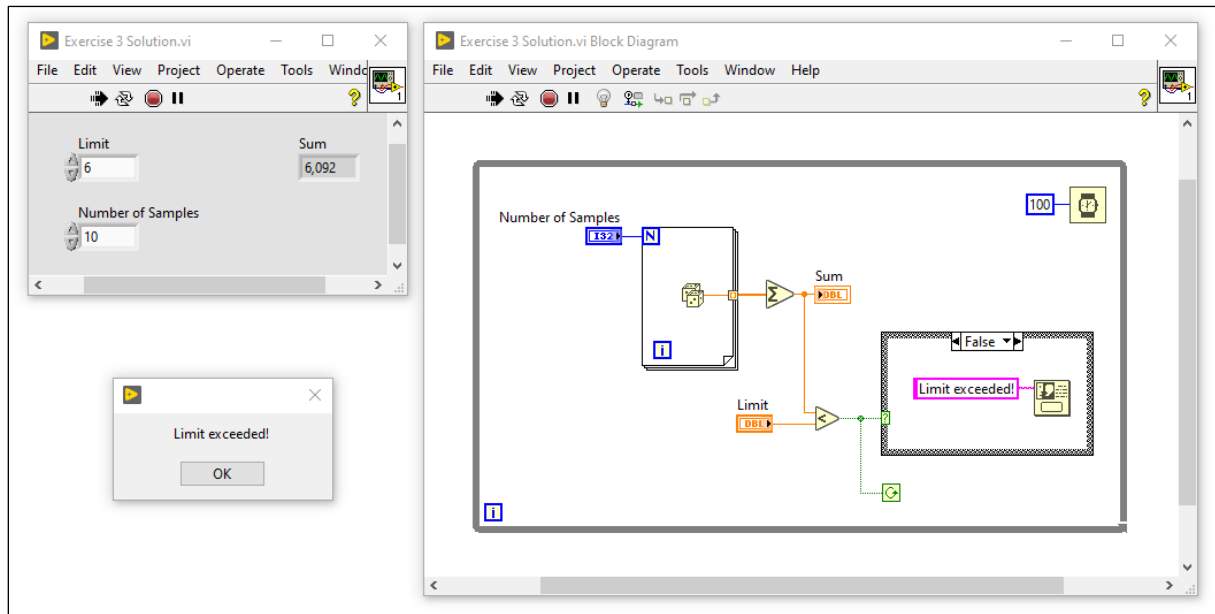


#### 4. Configure *While Loop*.

- Select *While Loop* item from *Structures* section of the functions palette and draw it on the BD. This loop should contain all your previous code in it.
- Connect the output of the *Less?* function to the *Loop Condition* input (  or  ).
- Change the *Loop Condition* node from *Stop if True* mode  to *Continue if True* mode . You can do it by clicking on it once or by selecting appropriate setting from node's context menu (RMB on the item usually opens the corresponding context menu).
- From the functions palette select *Timing* → *Wait (ms)* and place it inside the while loop.
- Create a constant for the input of *Wait (ms)* function and set it to 100.

#### 5. Add a pop-up and clean the VI.

- Select *Structures* → *Case Structure* item and draw it inside the WHILE loop.
- Configure the *Case Selector* input  with the *Less?* function output.
- Inside the *False* case place *Dialog & User Interface* → *One Button Dialog* function.
- Create a constant for *Message* input and fill it with some text.
- Switch to the FP and clean up all controls and indicators.



6. Run the VI and test it. Fix all detected issues.

**END OF EXERCISE**



## EXERCISE 4 Basic Data Types

### GOAL

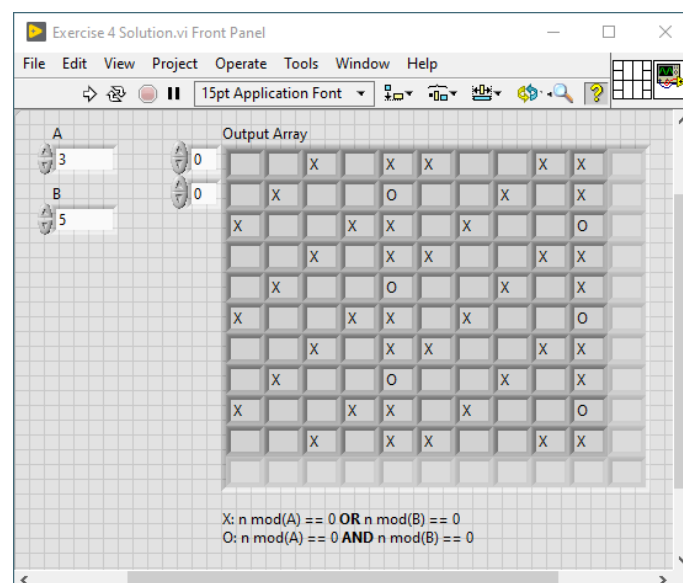
Create an application that checks whether numbers from 1 to 100 divide without remainder by two values entered by the user. The application creates 10-by-10 2D array and each element represents one integer as follows:

|     |     |     |     |
|-----|-----|-----|-----|
| 1   | 2   | ... | 10  |
| 11  | 12  | ... | 20  |
| ... | ... | ... | ... |
| 91  | 92  | ... | 100 |

The array should be filled with symbols indicating whether the corresponding integer divides by user-defined values. There are two symbols available:

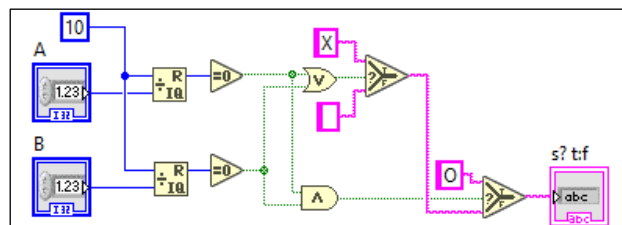
- **X** – indicates that the corresponding integer has non-zero remainder for exactly one of the two values entered by user, and the remainder of the second value is equal to zero,
- **O** – indicates that the corresponding integer has its remainder equal to zero for both values entered by user.

The specified array element remains empty if the corresponding integer has non-zero remainders for both values entered by the user.



## IMPLEMENTATION

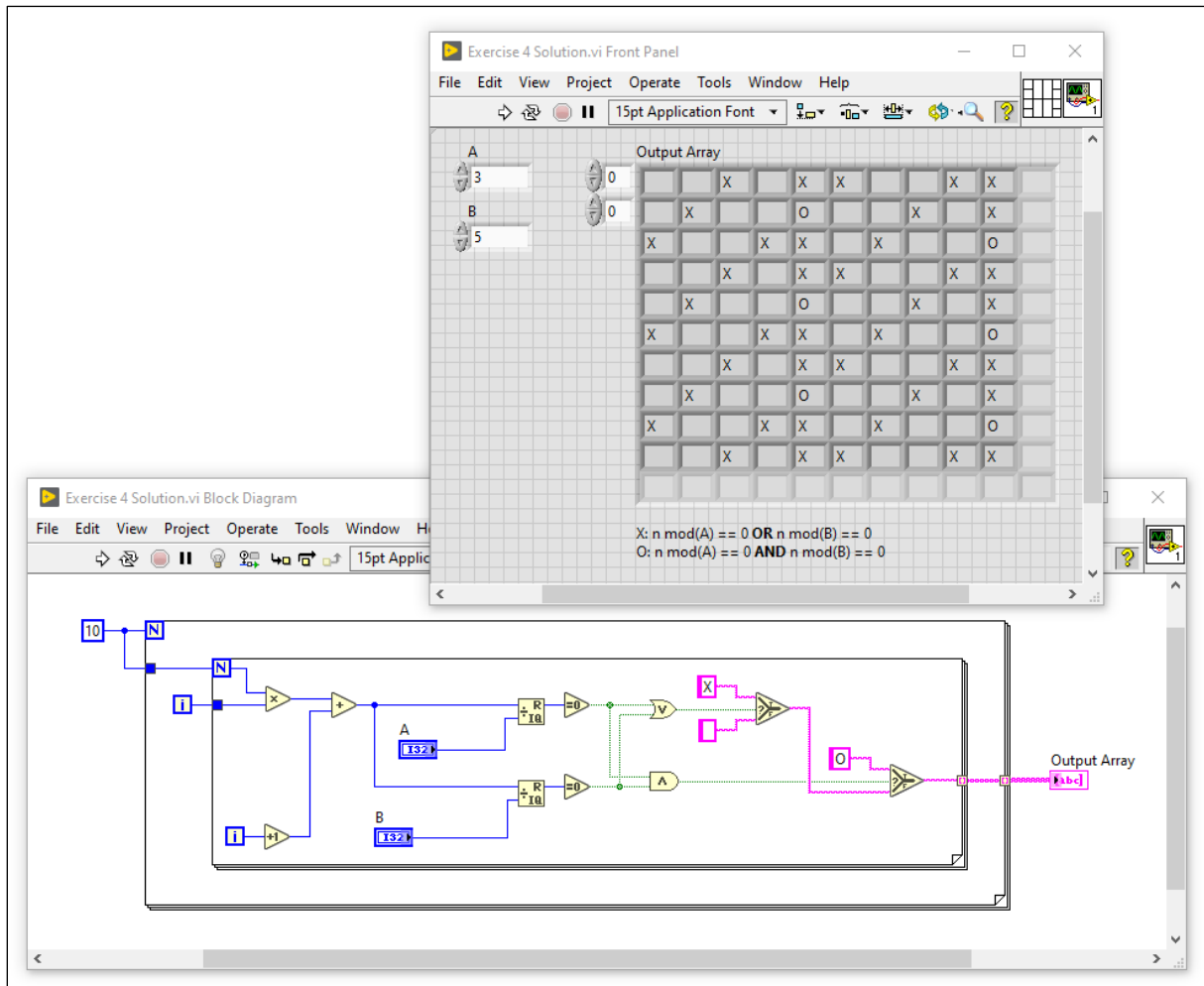
1. Create a new VI.
2. Create symbol selection logic for a single integer.
  - Place *Numeric Control* for the first user-defined divider on the FP. Set the data representation of the control to i32: from the context menu of the item, select *Representation* → *Long*.
  - Place *Numeric* → *Quotient & Remainder* function on the BD.
  - Connect *y* (lower) input of the function to the divider control.
  - Create a constant for *x* (upper) input of the function.
  - Select *Comparison* → *Equal to 0?* function and connect its input with  $x \cdot y \cdot \text{floor}(x/y)$  output of the *Quotient & Remainder* function.
  - Repeat all above steps for the second user-defined divider.
  - Select *Boolean* → *And* function. Connect its inputs to the *Equal to 0?* outputs for both dividers.
  - Configure *Boolean* → *Or* function similarly as *And* function.
  - Select *Comparison* → *Select* function and place it on the BD.
  - Connect *s* input of the *Select* function to the *Or* output.
  - Configure *t* and *f* inputs of the *Select* function, so it passes **X** symbol (formatted as a string constant) if *s* is TRUE and an empty string if *s* is FALSE.
  - Place another *Select* function and connect it to the *And* function output.
  - This node should pass **O** symbol if *s* is TRUE. If *s* is FALSE, it should pass the symbol returned by the other *Select* function connected to the *Or* operator.
  - Create an indicator for the *Select* function output. Test your implementation.



3. Create a 2D array of numbers from 1 to 100.
  - Draw *For Loop* around all of the previous code.
  - Create a constant for *Loop Count* terminal and set it to 10.



- Draw another *For Loop* containing existing loop. Set *Loop Count* value to 10.
  - Multiply the *Loop Iteration* value i of the external loop with the *Loop Iteration* value of the internal loop using *Numeric* → *Multiply* function (do it inside the internal loop).
  - Increment the internal *Loop Iteration* value using *Numeric* → *Increment* function.
  - Calculate the sum of the *Multiply* output and *Increment* the output using *Add* function.
  - Delete the constant connected to the *Quotient & Remainder* functions and replace it with the value calculated in the previous step.
  - Remove the string indicator containing the last symbol and connect this wire to the edge of the inner loop. This output connects to the external loop edge.
  - Create an indicator for the array terminal located on the external loop edge.
4. Organize FP
- Expand the 2D array indicator so it shows 10-by-10 elements.
  - Change controls and indicator names if you haven't already.
5. Run the VI and test it. Fix all detected issues.

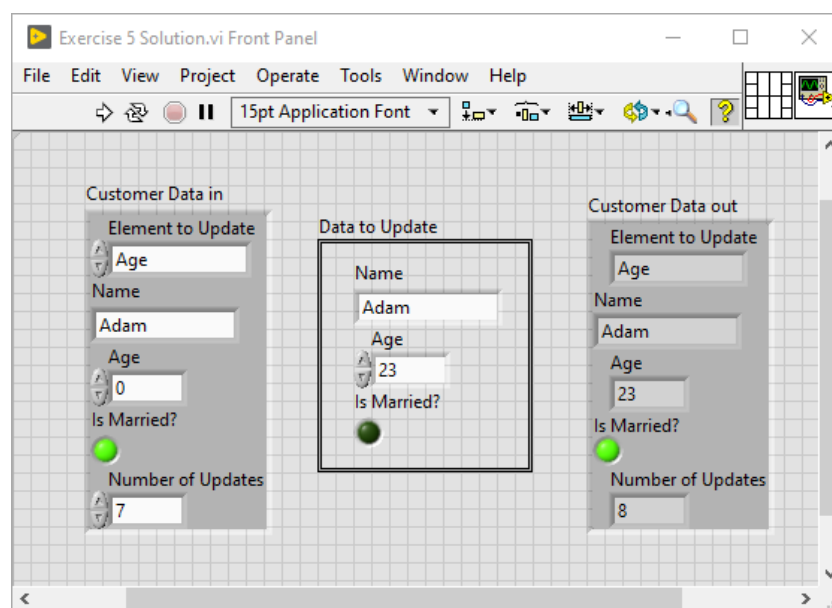


**END OF EXERCISE**

## EXERCISE 5 Clusters

### GOAL

Prepare a VI that takes a personal data cluster as input and modifies the specified element with the value entered by the user. The item to modify is determined by an enumeration data type (enum) inside the cluster named *Element to Update*. This enum should have *<None>* value which allows to execute the VI without modifying the cluster. The VI also increments a *Number of Updates* element if an update is made.



### IMPLEMENTATION


1. Create a new VI.
2. Prepare *Element to Update* enum.
  - Place *Ring & Enum* → *Enum* control on the FP.
  - Open the context menu of the enum control and select *Edit Items...*
  - Double click one the first row of the Items column, type *<None>*, and press the *Enter* key.
  - Fill the enum with other values: *Name*, *Age* and *Is Married?*. Click *OK* to close the properties window.



3. Prepare a personal data cluster.

- Place *Data Containers* → *Cluster* on the FP.
- Drag and drop the enum control to the cluster field.
- Select *String* control from the palette and place it inside the cluster.
- Fill the cluster with other elements: *Numeric* (u8) for *Age*, *Round LED* for *Is Married?* flag, and *Numeric* (i32) for *Number of Updates*.
- Click RMB on the cluster edge to open the context menu and select *AutoSizing* → *Arrange Vertically*.
- Make sure that all elements in the cluster have proper names.

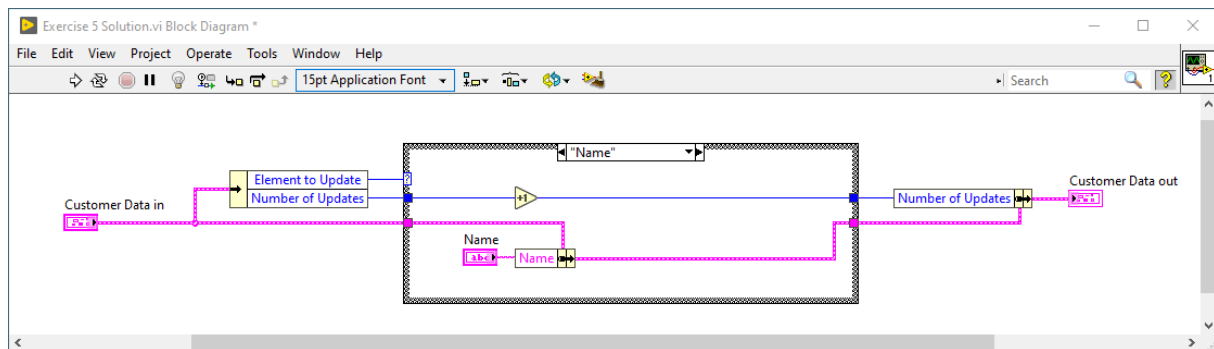
4. Implement update logic.

- Switch to the BD and find the cluster terminal.
- Draw *Case Structure* on the DB.
- Select *Cluster, Class & Variant* → *Unbundle by Name* function. Connect the cluster to its input.
- Expand the *Unbundle by Name* function by dragging the upper or lower function edge if needed.
- Configure the *Unbundle by Name* function to return *Element to Update* and *Number of Updates* fields. You can do it by clicking LMB on one of the function fields and selecting the expected cluster element.
- Connect *Element to Update* value to the *Case Selector* (  ). Connect the cluster and *Number of Updates* element to the structure edge (in order to create necessary tunnels), so they are available in all cases.
- Select *Add Case for Every Value* from the context menu of the CASE. Switch to the *Name* case.
- Place *Cluster, Class & Variant* → *Bundle by Name* function inside this case.
- Connect the customer data cluster to the *input cluster* input located on the top of the *Bundle by Name* function.
- Configure this function to expect a *Name* element as an input.
- Click RMB on the *Name* element of the bundle function and select *Create control* from the context menu.
- Connect the output (updated) cluster of the bundle function to the CASE edge.





- Increment the *Number of Updates* value inside the *Name* case. Connect the updated value to the CASE edge.
- Use the *Bundle by Name* function to set a new *Number of Updates* value in the cluster outputted from the CASE.
- From the context menu, create an indicator for the output cluster.



5. Finish the implementation.

- Implement *Age* and *Is Married?* cases similarly as the *Name* case.
- For the *<None>* case, simply pass the cluster and the *Number of Updates* value unchanged through the CASE.
- Switch to the FP and arrange all controls and indicators.

6. Run the VI and test it. Fix all detected issues.

**END OF EXERCISE**



## SPEECH RECORDER 1 Express VIs

### GOAL

Create a LabVIEW project and a VI that takes an audio signal from the computer's sound card, plays it, and saves it to a file.

### IMPLEMENTATION

1. Create a new project and main VI.
  - In the LabVIEW Hub select *File* → *Create Project...*
  - In the project configuration window select *Blank Project* template and click *Finish*.
  - Save the project and name it *Speech Recorder*.
  - Go to the course materials and find *Exercises/Speech Recorder 1* folder. Copy *SaveAndPlayFile.vi* and paste it to the location of your newly created LabVIEW project file (*Speech Recorder.lvproj*).
  - Go back to the LabVIEW project window and click RMB on the *My Computer* item in the project tree to open the context menu. Select *Add* → *File...* and navigate to the file copied in the previous step. Select it and add to the project.
  - Open the added VI and analyze implemented functionality. Pay attention to the function inputs.
  - From the context menu of the *My Computer* item in the project tree select *New* → *VI*. Save it in the same location as the project file and name it *Main.vi*.
2. Implement logic to acquire and play a sound signal.
  - Open the BD of *Main.vi* and place *Acquire Sound* express VI. You can find it in *Graphics & Sound* → *Sound* → *Input*. In the configuration window you can stick to the default settings – take a look on the possible options and close the window.
  - Create necessary constants: *Device*, *Duration (s)*, *Sample Rate (Hz)*, *#Channels*, *Resolution (bits)*. You can change the duration to two seconds and channels number to 2.



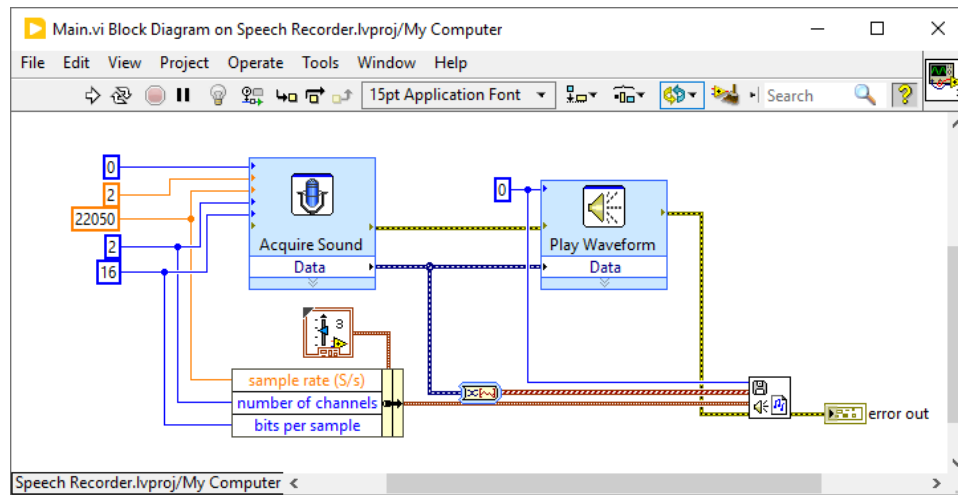
- Place *Play Waveform* express VI. You can find it in *Graphics & Sound* → *Sound* → *Output*. Leave default settings and close the configuration window.
- Connect the *Data* and *error out* outputs of the *Acquire Sound* express VI to the corresponding inputs of the *Play Waveform* express VI.
- Create a constant for the *Device* input and set it to 0.

### 3. Implement file saving feature.

- Go to the project tree. Drag and drop *SaveAndPlayFile.vi* on the BD of *Main.vi*.
- Connect the error wire between the *Play Waveform* and the added subVI. Set the *Speaker Device ID* input to 0.
- Check the data type of the subVI's *Waveform(s) in* input. Compare it with the *Acquire Sound* output data type. You can easily use the Context Help feature (*Ctrl+H*).
- Open Quick Drop tool and type *Convert from Dynamic Data*. Select highlighted function and set resulting data type to *1D array of waveform* in the configuration window. Use this function to convert the acquired sound data into the desired format and connect it to the subVI input.
- Create a constant for the subVI's *Sound Format* input and disconnect it.
- Configure the *Sound Format* cluster by bundling required values to it with a *Bundle by Name* function. You can use the *Sample Rate (Hz)*, *#Channels*, and *Resolution (bits)* constants connected to the *Acquire Sound* function.
- Connect the modified cluster to the appropriate subVI input.
- Create an indicator for the *error out* output.

### 4. Run *Main.vi*

- Run the VI and test it. Fix all detected issues.
- Check if a wave file has been created in the *Measurements* directory of the project.



**END OF EXERCISE**



## SPEECH RECORDER 2 Sound Input Functions

### GOAL

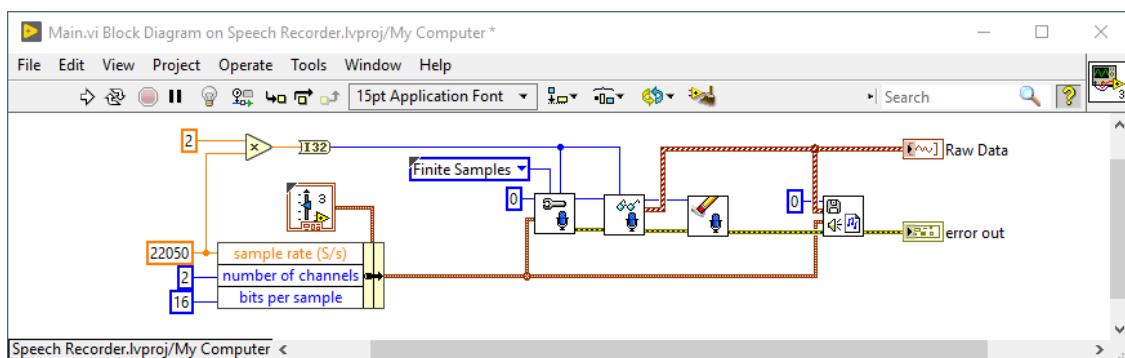
Modify the application created in the previous step so the express VIs are replaced with the low level functions. The application should play the recorded sound only once (from the wave file). Display the acquired signal on the FP.

### IMPLEMENTATION

1. Remove redundant functions.
  - Open the *Speech Recorder* project and *Main.vi*.
  - Remove the *Play Waveform* and *Acquire Sound* Express VIs. Press *Ctrl+B* to automatically clear all broken wires.
  - Leave the code responsible for building the *Sound Format* cluster. Remove all constants that are not connected to anything. Remove the *Convert from Dynamic Data* function.
  - Clean-up the BD if necessary.
2. Rewrite the functionality of acquiring the sound signal.
  - Go to the *Graphics & Sound* → *Sound* → *Input* palette. You can click the pin symbol in the upper right corner of the palette so it stays open.
  - Drag and drop three functions to the BD: *Sound Input Configure.vi*, *Sound Input Read.vi*, *Sound Input Clear.vi*.
  - Connect all VIs together using error and *Task ID* lines so they execute in the proper order. Connect the *SaveAndPlayFile.vi* function with the error output at the end of the operation chain.
  - Create constants for the *device ID* and *sample mode* inputs of the configuration function. Set them to 0 and *Finite Samples* respectively.
  - Multiply the *sample rate (S/s)* constant by the number of seconds you want to acquire. Pass the result to the *number of samples/ch* input of the configuration function. Notice that the coercion dot appeared on this input.



- Convert the number of samples value to the expected data type by clicking RMB on the value wire and selecting: *Insert* → *Numeric Palette* → *Conversion* → *To Long Integer (I32)*.
  - Connect this value also to the *number of samples/ch* input of the *Sound Input Read.vi* function.
  - Connect the *data* output of the read function to the *Waveform(s) in* input of the *SaveAndPlayFile.vi* subVI.
3. Display the acquired sound signal on the FP.
- Switch to the FP and place a *Waveform Graph* object.
  - Go back to the BD and connect created waveform terminal with the acquired sound data wire.
4. Run the VI and test it. Fix all detected issues.



**END OF EXERCISE**



## SPEECH RECORDER 3 Filtering and FFT

### GOAL

Modify the application to calculate the audio signal spectrum (FFT) before saving the audio data to the file. The application should also apply filtering to the signal before computing the FFT. Add the function of saving the signal to the file only when the signal level is higher than the defined threshold.

### IMPLEMENTATION

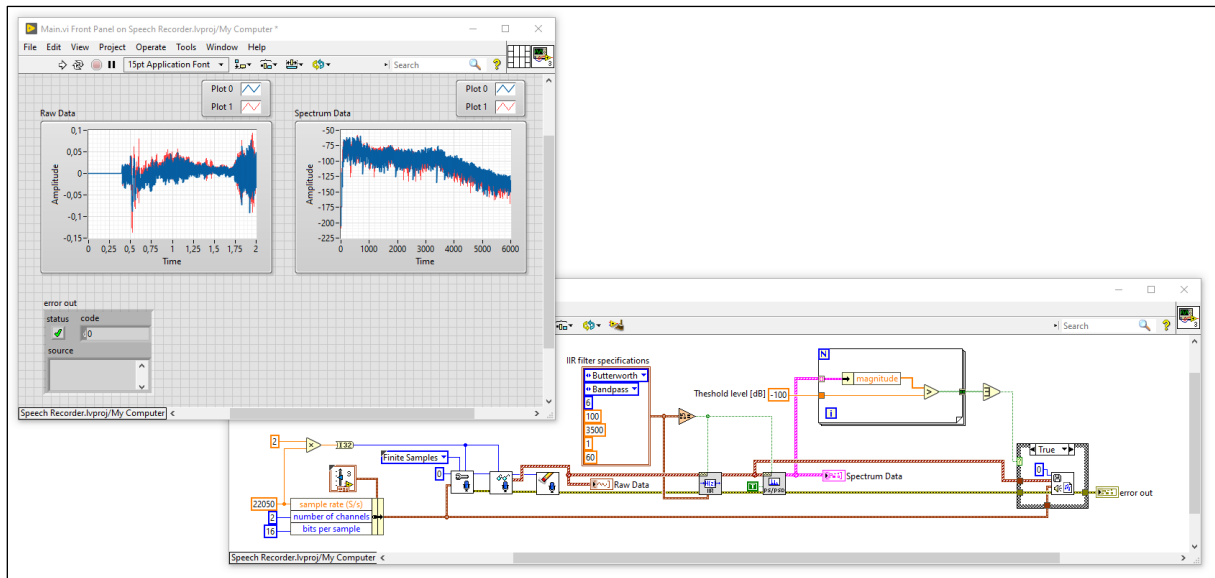
1. Open the *Speech Recorder* project and *Main.vi*.
2. Implement the signal filtering feature.
  - Make some space on the BD for additional functions. You can expand BD by moving mouse in desired direction with LMB and *Ctrl* key pressed.
  - Disconnect the error and sound data wires from the *SaveAndPlayFile.vi* function. Leave the data wire connected to the waveform graph.
  - Place *Digital IIR Filter* function on BD. Connect necessary wires to it (error line and sound data).
  - Create a constant for *IIR filter specifications* input. You can try different settings and observe the results later, but for now you can leave the default values.
  - Place *Is Value Changed* function and connect the constant to it. The output of this function connect to the *reset filters* input.
  - Pass the filtered signal to the saving function input.
3. Implement the FFT calculation.
  - Find and use *FFT Power Spectrum and PSD* function to calculate the FFT. Pass the filtered signal to the *time signals* input. Connect both error tunnels so this function executes between filtering and saving operations.
  - Create a constant for the *dB On (F)* input. Set it to TRUE.
  - Switch to the FP and create a new *Waveform Graph*.
  - On the BD, connect the power spectrum output of the FFT function with the newly created waveform graph.



- Configure the *restart averaging (F)* input using the value utilized for filtering function reset.
4. Implement the threshold detection.
- Place *For Loop* on the BD.
  - Connect the *Power Spectrum/PSD* output of the FFT function to the loop edge. Created tunnel should have indexing mode enabled, so the loop iterates through all of the array elements.
  - Inside the loop, place *Unbundle by Name* function and read the *magnitude* array from the power spectrum data cluster.
  - Inside the loop, place *Greater?* function. Connect the *magnitude* array to it.
  - Create *DBL Numeric Constant* – this constant will represent the requested threshold level in decibels. Connect it to the second input of the comparison function<sup>2</sup>.
  - Pass the output of the comparison function to the loop edge and set *Tunnel Mode* → *Concatenating*. You can do it from the context menu.
  - Connect the newly created tunnel to *Or Array Elements* function.
  - Draw *Case Structure* around the *SaveAndPlayFile.vi* function. Configure the case selector so that the saving routine only executes when the threshold is exceeded. Otherwise the signal should not be saved.
5. Run the VI.
- Clean-up the BD and FP of *Main.vi*: remember to name all controls and indicators. Try to keep the wires organized and not intersecting each other.
  - Run the VI for various filtering settings. Experiment with the threshold values. Fix all detected issues.
- 

<sup>2</sup> If you create an array instead of a scalar, don't worry - you can easily change the array to an element (and *vice versa*) in the target object's context menu.





**END OF EXERCISE**



## SPEECH RECORDER 4 Code Clean-up

### GOAL

Modify the application so that the signal analysis feature is now encapsulated in a subVI. The threshold value and filtering parameters should be defined by the user. Add simple error handling.

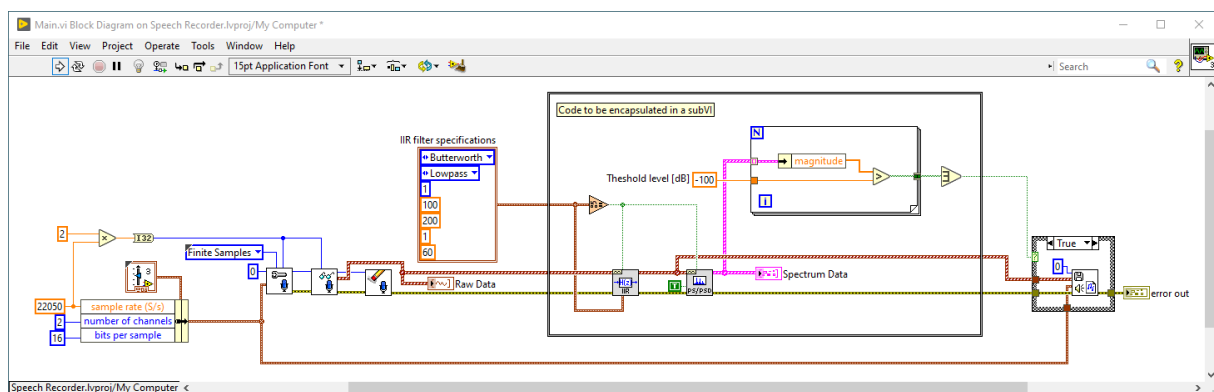
### IMPLEMENTATION

1. Organize project items.
  - Open the *Speech Recorder* project tree. Click RMB on the *My Computer* item and select *New* → *Virtual Folder*. Create a new virtual folder and name it *SubVIs*.
  - Go to the project file location on the hard drive. Create a new folder with the same name (*SubVIs*).<sup>3</sup>
  - Open *SaveAndPlayFile.vi* and remove the *Play Sound File.vi* function. You can use the *Remove and Rewire* option available in the context menu. Remove the redundant *Device ID* input as well.
  - In the project tree, drag and drop *SaveAndPlayFile.vi* to the *SubVIs* virtual folder. Click RMB on *SaveAndPlayFile.vi* and select *Rename...* option. Once the new window pops-up, go to the project *SubVIs* local directory, rename the file to *SaveFile.vi*, and save the VI.
2. Create a subVI responsible for signal analysis.
  - Open the BD of *Main.vi*. Clean-up the code if there are any broken wires.
  - Reorganize the code. The logic you want to include in a subVI should be stored close together and parameters that are expected to be passed to (or outputted

---

<sup>3</sup> You can quickly go to the project item location by selecting *Explore...* from the item's context menu.

from) this subVI should be outside of this area. The SubVI you are about to create should contain the filtering, FFT, and threshold features. This SubVI should also have inputs for error, raw data, filter specifications, and threshold value, and outputs for filtered data, spectrum data, threshold exceeded flag and error.

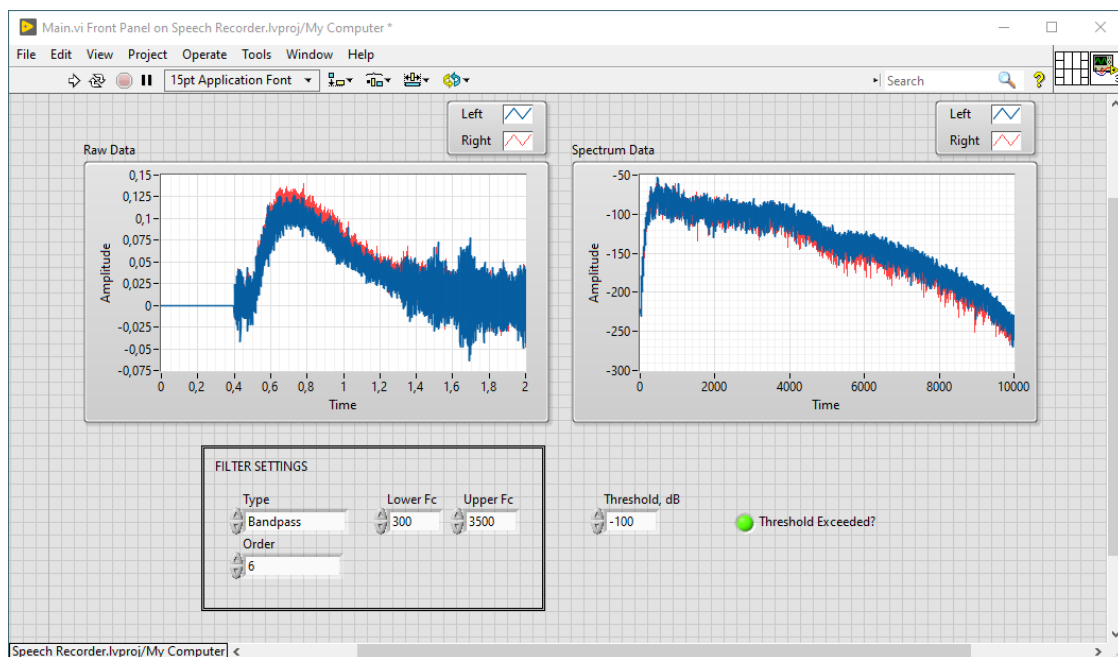


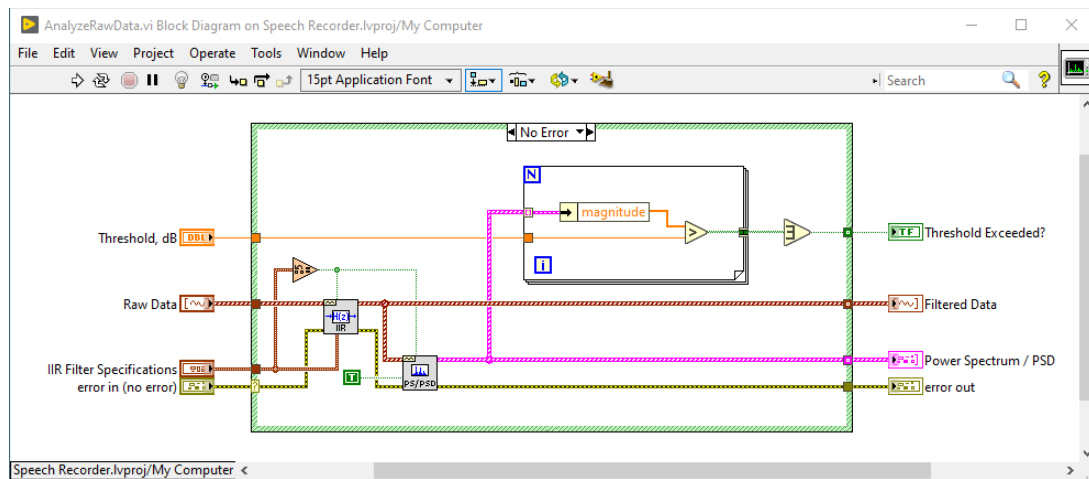
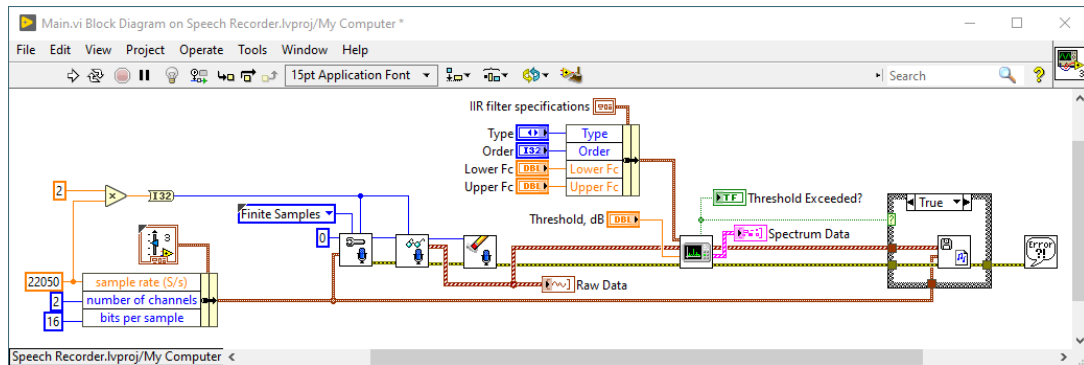
- Select the code that should be included in a subVI, then Select *Edit* → *Create SubVI*.
- Open the newly created subVI and save it in the *SubVIs* directory as *AnalyzeRawData.vi*.
- Go to the project tree. Drag and drop the *AnalyzeRawData.vi* to *SubVIs* virtual folder.
- Double click on the new subVI's icon. Modify the icon using *Icon Editor* tool.
- Clean-up the code inside the subVI, rename controls and indicators if necessary.
- Go back to *Main.vi* and organize wires connected to the newly created subVI.

### 3. Simple error handling.

- Open the BD of *AnalyzeRawData.vi* subVI.
- Draw *Case Structure* in a way that all controls and indicators are outside the structure and all logic is inside the structure.
- Locate the error input tunnel. Open tunnel's context menu and select *Replace with Case Selector*.
- Switch to the *Error* case. Wire error line through the case. For other output tunnels select *Use Default If Unwired* from the context menu – remember, tunnels remain empty if not connected.

- Go to *Main.vi* and remove the *error out* indicator located at the end of the code. Connect the error output to *Simple Error Handler.vi* instead.
4. Controls for filtering and analysis.
- Open threshold constant context menu and select *Change to Control* option.
  - Disconnect the *IIR filter specifications* cluster from the subVI input. Use *Bundle by Name* function to configure needed cluster elements separately. Prepare inputs for following cluster elements: *Type*, *Order*, *Lower Fc*, *Upper Fc*.
  - Create controls for all bundling function inputs. Connect output cluster to the proper *AnalyzeRawData.vi* subVI input.
  - Switch to the FP and organize all controls and indicators. Create an indicator for the *Threshold Exceeded?* value. Add decorations to highlight the groups of controls.
5. Run the VI and test it. Fix all detected issues.





**END OF EXERCISE**

## SPEECH RECORDER 5 Continuous Acquisition

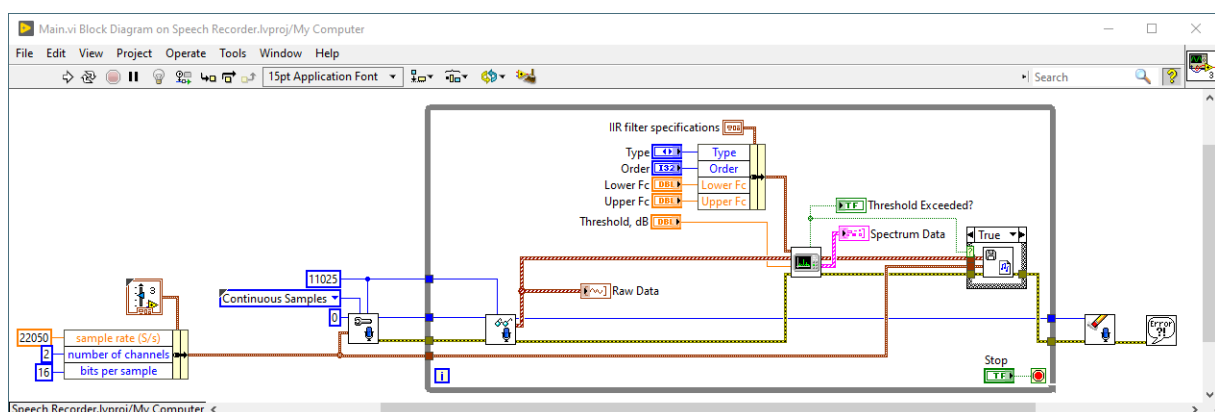
### GOAL

Modify the application by implementing continuous acquisition. The user should be able to stop the application.

### IMPLEMENTATION

#### 1. Implement continuous acquisition.

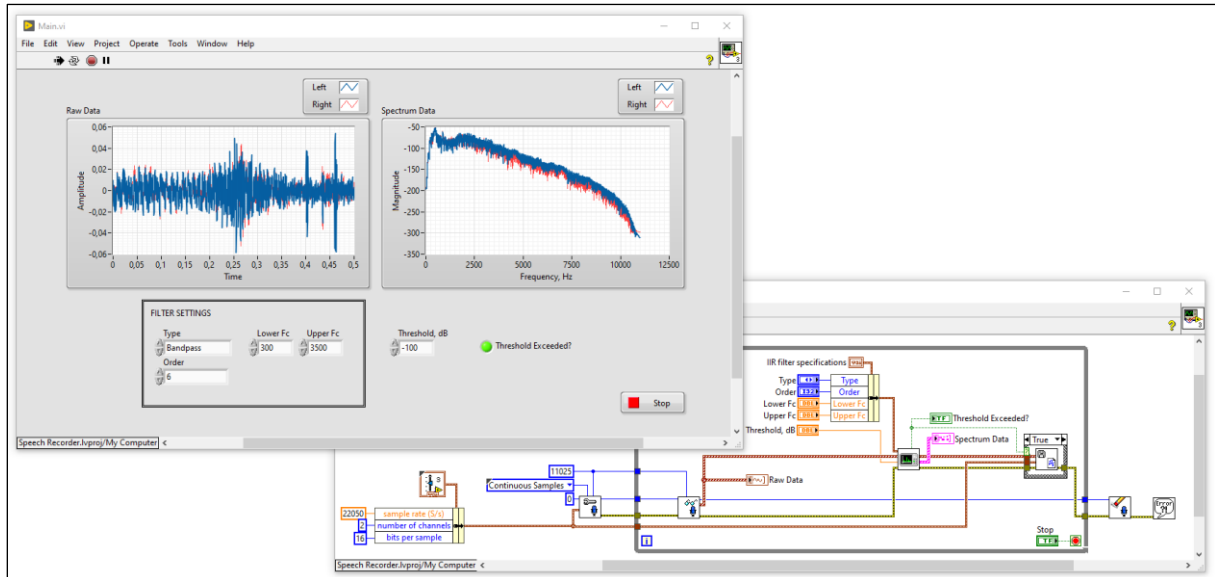
- Open the BD of top-level VI.
- Move *Sound Input Clear.vi* to the end of the operation chain. It should execute as the last function, but before the error handling function.
- Draw *While Loop* in a way that *Sound Input Configure.vi* and *Sound Input Clear.vi* are outside the loop.
- Create a control for *Loop Condition* terminal.
- Change the sample mode of the configuration function from *Finite Samples* to *Continuous Samples*.
- Set the value connected to *number of samples/ch* inputs for *Sound Input Configure.vi* and *Sound Input Read.vi* functions to 11025 – remove unnecessary objects and leave *sample rate (S/s)* as 22050.



- Clean-up the BD.
- Verify tunnel modes for *error* and *Task ID* lines in the loop. If necessary, change them to the tunnels by selecting *Replace with Tunnels* from the context menu.

## 2. Run and test the VI.

- Run *Main.vi* and observe whether an error appears occasionally. What could be causing such behavior?



**END OF EXERCISE**



## SPEECH RECORDER 6 File Saving SubVI

### GOAL

Modify the subVI responsible for the file saving operation. This subVI should create a new file every time the signal exceeds the requested threshold and close the file once the signal drops below the threshold.

### IMPLEMENTATION

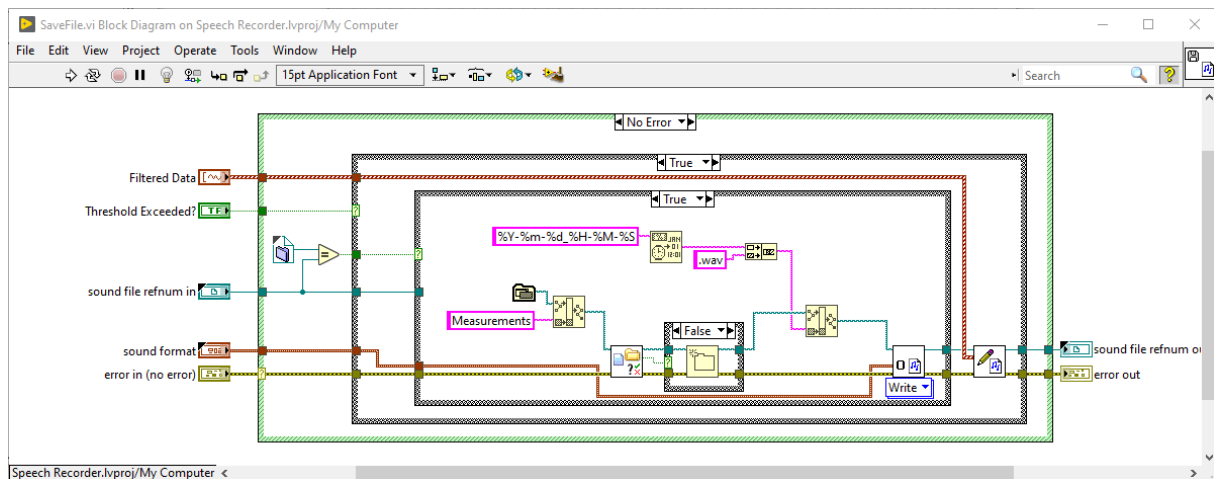
1. Implement file streaming feature.
  - Open *SaveFile.vi*.
  - Create a control for *Threshold Exceeded?* flag.
  - Draw *Case Structure* inside the *No Error* case in such a way that it covers all the code. Connect *Threshold Exceeded?* value to the case selector.
  - Leave open and write operations in the *True* case. Move *Sound File Close.vi* to the *False* case and connect the error wire.
  - Create a control and an indicator for *sound file refnum* input and output, and move them outside the CASEs. Make sure both terminals are connected to the file refnum line and the file operation flow remains uninterrupted. You can create *sound file refnum* objects using proper terminal of *Sound File Write.vi* or *Sound File Close.vi* function – simply create desired object from the context menu.
  - In the *False* case, create a constant for empty file refnum tunnel. Pass the file refnum wire through the *Error* case.
  - In the *No Error* case and outside the inner CASE, place *Equal?* function. Connect the sound file refnum control to one of its input. Create a constant for the other input, so it compares incoming refnum to a default (null) value. Connect the comparison result to the inner CASE.
  - In the *True* case of the inner CASE, add another *Case Structure*. Use it to ensure that a new file is opened only if the incoming file refnum is null. Otherwise, only the writing to the file operation should be performed.

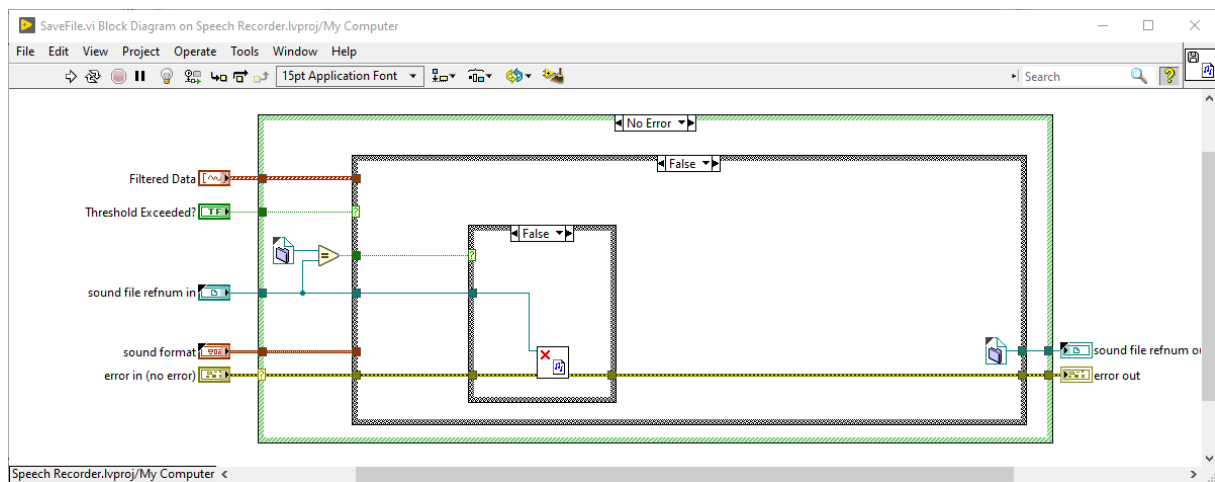


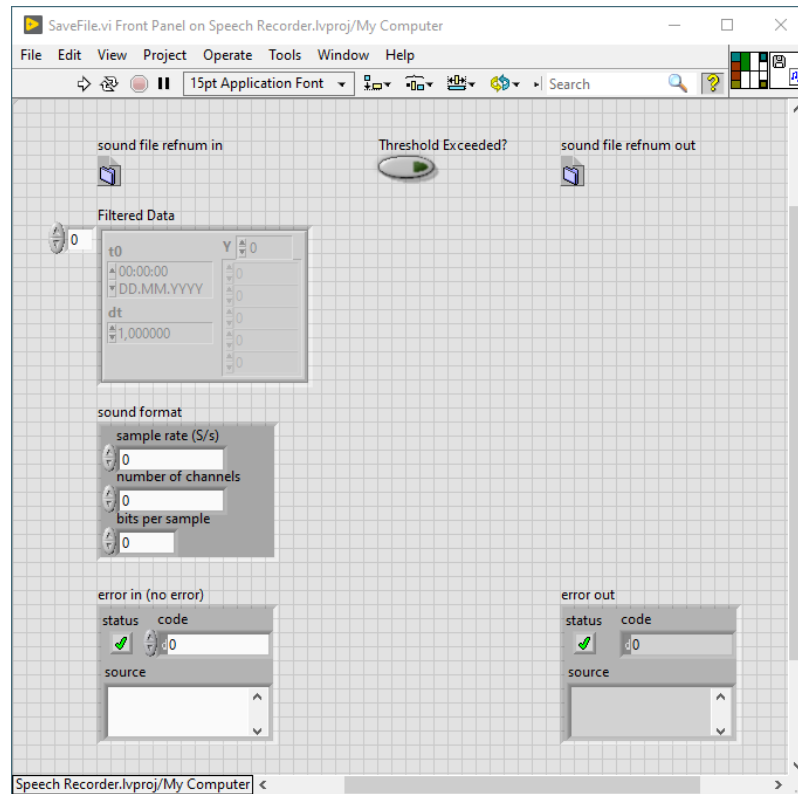
- Draw another *Case Structure* around the *Sound File Close.vi*. This VI should only be executed if the input *sound file refnum* is not equal to the default value – it simply means that the open file should be closed.
- Clean-up the subVI and ensure that all tunnels of all cases are connected – if needed, pass the data through certain cases by simply connecting corresponding pairs of tunnels.

## 2. Modify the inputs.

- Switch to the FP.
- Click LMB on the upper left connector on the CP (Connector Pane), then select the *sound file refnum in* control. The connector should turn blue.
- Connect the *sound file refnum out* indicator to the upper right connector.
- Connect the *Threshold Exceeded?* control to the top left connector which remained unused.
- Clean-up the FP – you can do it by pressing *Ctrl+A*, then *Ctrl+Space*, then *Ctrl+F*. This operation should arrange all controls and indicators accordingly to the CP connections.







**END OF EXECERISE**



## SPEECH RECORDER 7 Integrating VIs

### GOAL

Modify *Main.vi* so it utilizes the subVI created in the previous step. Add logic to prevent memory leaks when the application is closed.

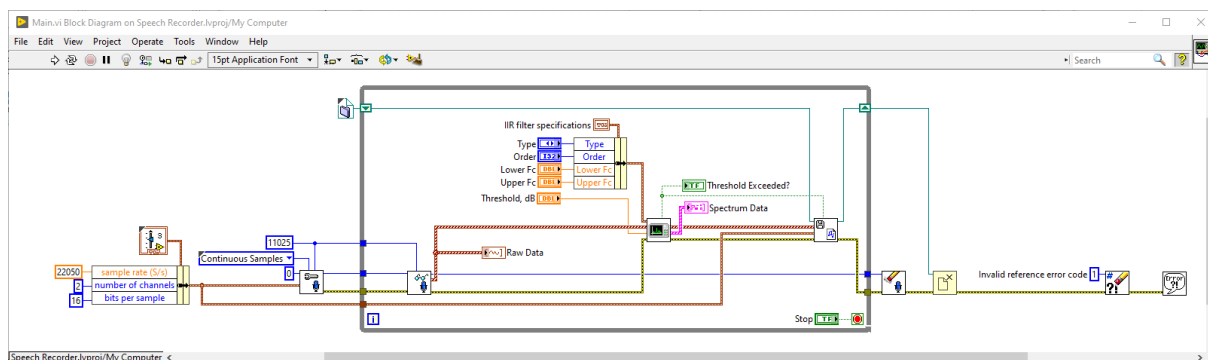
### IMPLEMENTATION

#### 1. SubVI configuration.

- Open the BD of *Main.vi*.
- Remove the CASE around *SaveFile.vi*. You can do it without deleting the content by selecting *Remove Case Structure* from the CASE's context menu.
- Connect *Threshold Exceeded?* value to the subVI input.
- Connect *sound file refnum* output to the WHILE edge and modify the tunnel by selecting *Replace with Shift Register* in its context menu. The input shift register node will be created automatically.
- Connect the input shift register to the subVI's *sound file refnum* input. Create a constant to initialize the shift register.

#### 2. Prevention of memory leak.

- Add *Close File* function after the WHILE. Connect the file refnum line returned by the shift register.
- Add *Clear Errors* function right after *Close File* function. Configure it to clear specific error 1 related to an invalid refnum value.
- Connect all VIs executed after the WHILE with the error wire. *Simple Error Handler* should be the last function in the chain.



**END OF EXERCISE**



## SPEECH RECORDER 8 State Machine

### GOAL

Modify the application architecture to follow the state machine pattern.

### IMPLEMENTATION

#### 1. Data structure preparation.

- Create *Typedefs* folder in the project tree and locally on the disk.
- Click RMB on the *Typedefs* virtual folder and create *New* → *Type Definition*. A new window should open automatically.
- In the type definition window place a *Cluster* container control and name it *Application Data*. Save the typedef file in the target directory as *#cluster\_ApplicationData.ctl*.
- The cluster should contain the following items:
  - *Sound file refnum* – you can obtain a control of that type by copying it from the *SaveFile.vi* FP.
  - *Acquisition Task ID* – the data type of this control is *Sound Input Task ID* used by the *Sound Input* functions.
  - *Sound Format* – a cluster used in *SaveFile.vi*.
  - *Number of samples/channel* – i32 numeric.
- After configuring the application internal data cluster, arrange it vertically or horizontally (use proper options from cluster's context menu), save it and close its window.
- Create another type definition in the *Typedefs* folder and name it *#enum\_State*. This one will be used to determine application states and should contain *Enum* control filled with the following items: *Init*, *Acquisition*, *Error Handler*, *Exit*.

#### 2. Implementing *Init* and *Acquisition* states.

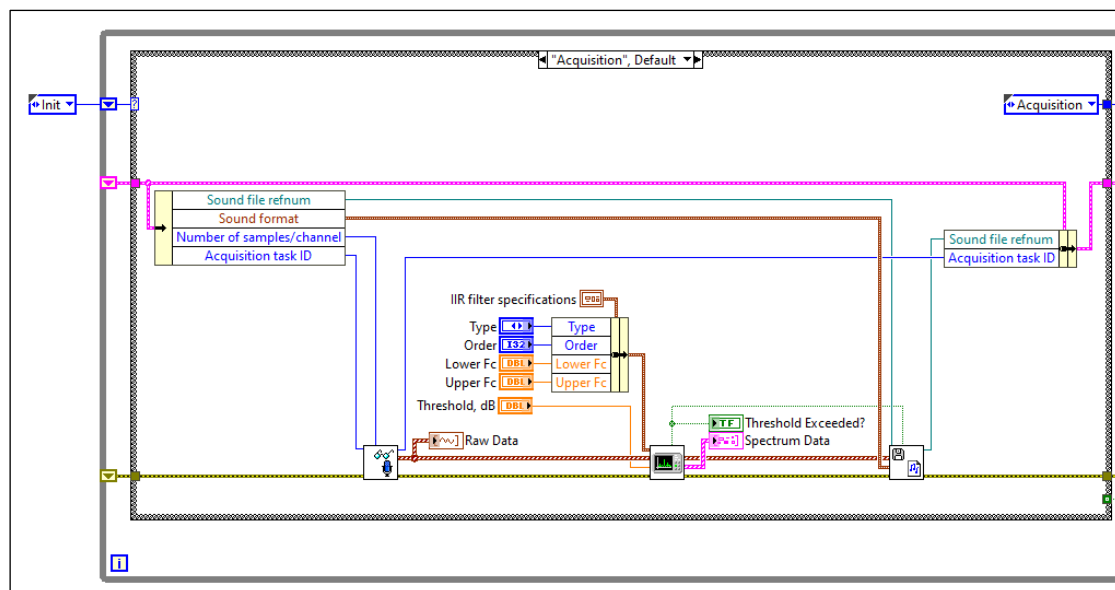
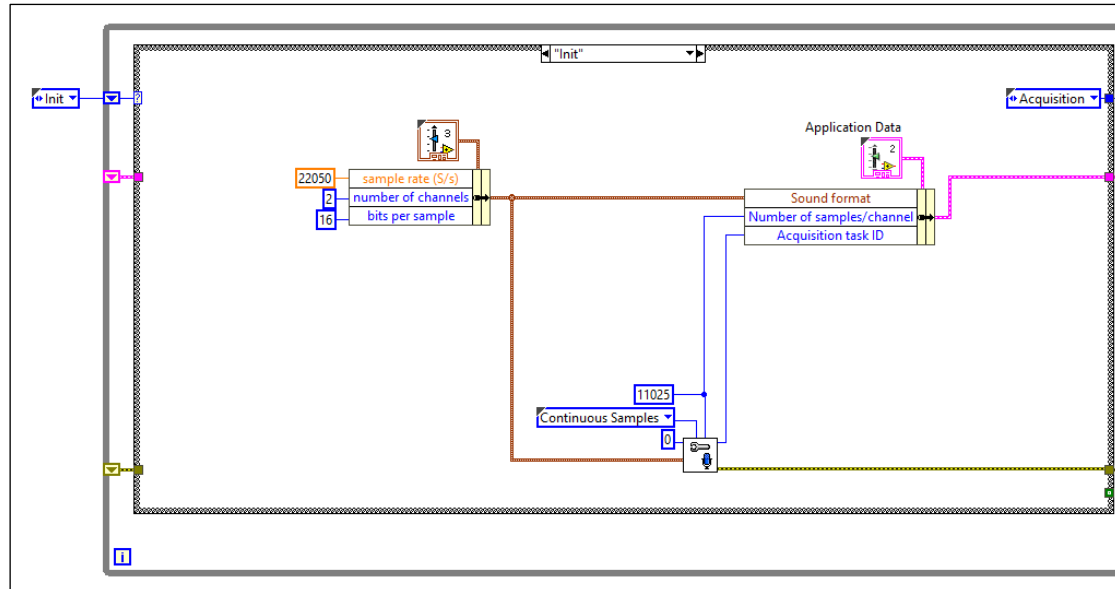
- Go to *Main.vi*.
- Remove the shift register and clear broken wires.



- Draw *Case Structure* around the code inside the WHILE. The *Stop* terminal should stay outside the CASE.
- Drag and drop *#enum\_State* object from the project tree and place it before WHILE. Change the enum to constant and set its value to *Init*.
- Connect the state constant to the WHILE and configure the tunnel as shift register, then Connect the state line to the outer CASE as well.
- Configure the outer CASE to have two cases: *Init* and *Acquisition, Default*. The *Acquisition* case should contain all of the code written up to this point. The *Init* case should be empty for now.
- Inside the *Acquisition* case, place a state enum constant and set its value to *Acquisition*. Connect it to the right shift register – this is the desired transition to the next application state.
- Replace the WHILE's error tunnels with the shift register.
- Move the sound format cluster and the *Sound Input Configure* function to the *Init* case. Remove all disconnected tunnels. No code should remain before the WHILE and it should only have two pairs of shift registers (state and error lines) and no other tunnels.
- Inside the *Init* state, place an instance of *Application Data* cluster as a constant. Use bundling function to configure following elements: *Acquisition Task ID*, *Sound format*, *Number of samples/channel*.
- Connect appropriate values to the bundling function. Connect all inputs of *Sound Input Configure* function – remember about the error line.
- Connect configured application internal data cluster to the WHILE's right edge and change the tunnel to the shift register – application data cluster will be available for all states now.
- Inside the *Init* case, place another state enum constant and set its value to *Acquisition*, and pass it to the corresponding shift register.
- Switch to the *Acquisition* case and clear all broken wires. Use unbundling function to extract application data needed to configure and call *Sound Input Read* and *SaveFile.vi* functions.



- Use another bundling function to update *Sound File Refnum* and *Acquisition Task ID* elements stored in the application data line. Forward the updated cluster to the appropriate tunnel and shift register.



### 3. Implementing *Error Handler* and *Exit* states.

- Open the main CASE's context menu and select *Add Case After* option. Add case for the *Error Handler* state – if necessary, input its name in the new state *Selector Label*.
- Pass the application internal data cluster through this case.
- Call the *Exit* state as the very next state after the *Error Handler* state.



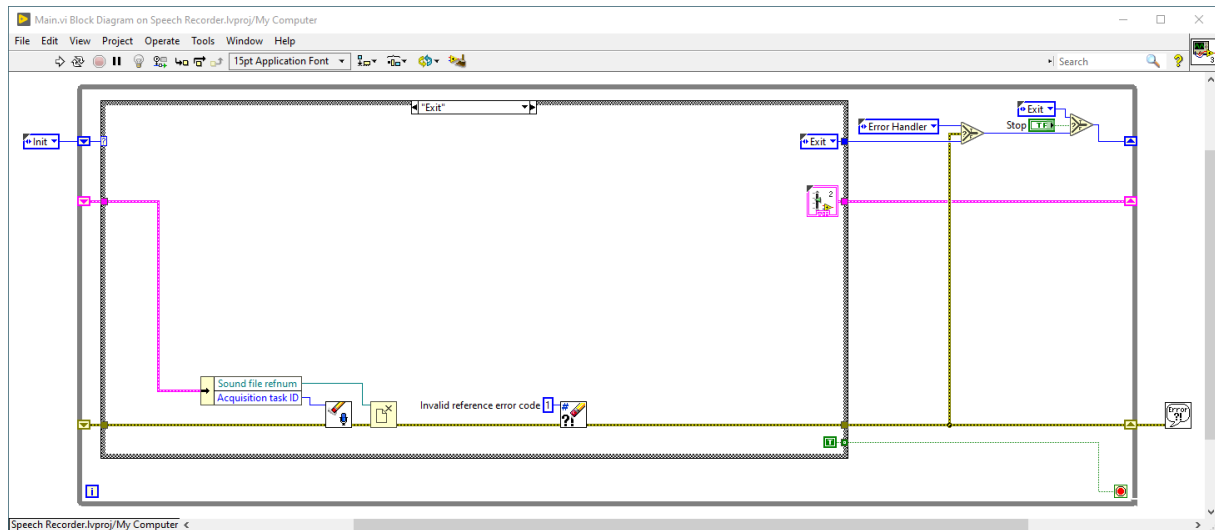


- Add *Clear Errors* function and connect it to the error line, so that every error is cleared – this is the easiest way to handle errors and close the application.
- Add another case for the *Exit* state.
- Move all functions executed after the WHILE into this case (except the *Simple Error Handler* – leave it outside the WHILE). Use the application internal data line to configure moved functions.
- Create constants for empty output tunnels and check if the error line is connected appropriately.

4. Implementing steps navigation.

- Place *Select* function behind the main CASE. Connect the error wire to its selector.
- Connect the next state value outputted from the main CASE to the *f* input of the *Select* function. Create a constant for the remaining *t* input and set it to *Error Handler* value.
- Place another *Select* function. Disconnect the *Stop* terminal from *Loop Condition* and connect it to the second *Select* function. Configure it to pass the *Exit* state if the *Stop* condition is TRUE. Otherwise, the second *Select* function should pass the state outputted from the first *Select* function.
- Disconnect the value connected to the right shift register and replace it with the output of the second *Select* function.
- Create a constant for *Loop Condition* terminal and set it to TRUE. Drag it to the *Exit* case and configure the tunnel to *Use Default If Unwired*.

5. Run and test the application.



42 / 47



## SPEECH RECORDER 9 Event Logging

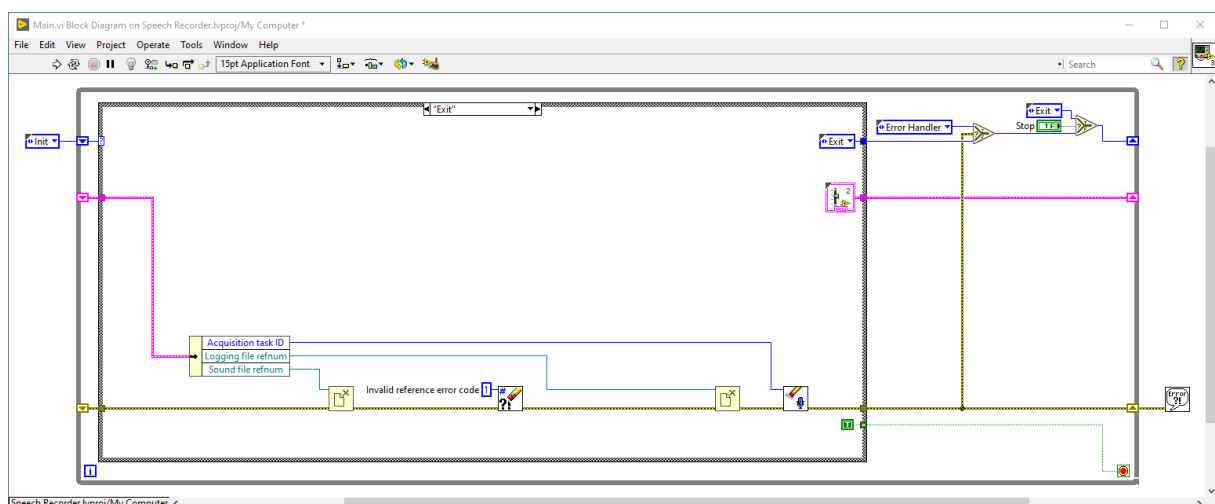
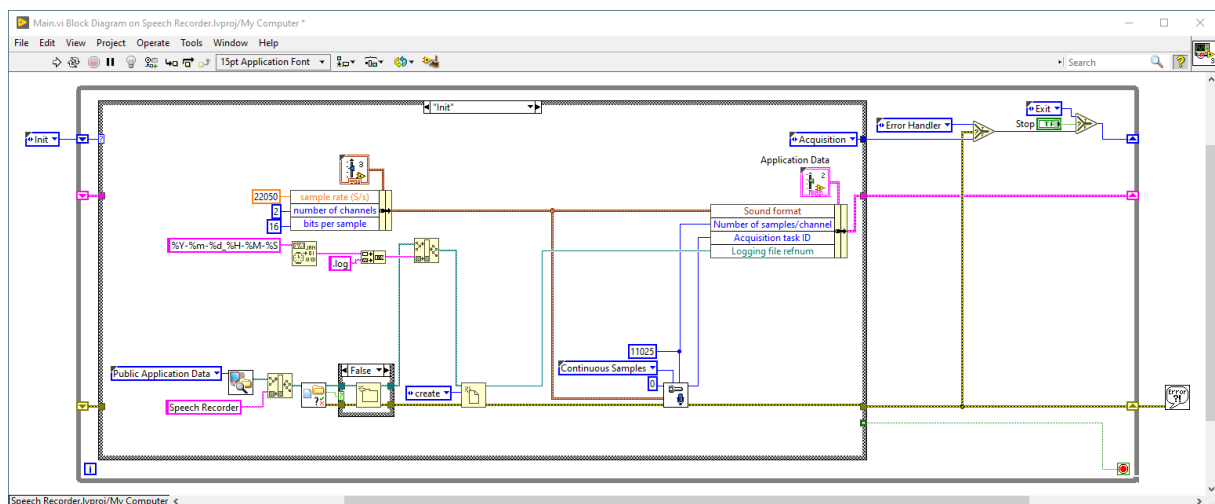
### GOAL

Add event logging to the Speech Recorder project.

### IMPLEMENTATION

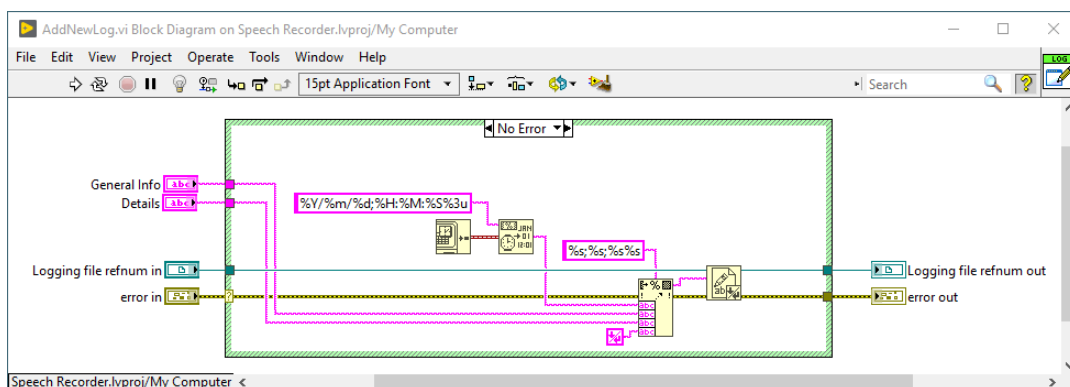
1. Create and close a log file.
  - Go to the *Init* state of *Main.vi*.
  - Choose the location where the logs should be stored – it is best to choose a universal location for Windows computers, e.g. the application data path. Use *Get System Directory* function and configure it to *Public Application Data* to obtain the recommended path.
  - Prepare a subfolder for Speech Recorder application data: connect the output of *Get System Directory* function to *Build Path* function. Create a constant for *Relative Path* input and set it to *Speech Recorder*.
  - Check if the path created in the previous step already exists on the disk: use *Check if File or Folder Exists* function. Create *Case Structure* and connect *file or folder exists?* output to its selector.
  - If the folder does not exist, use *Create Folder* function to create it. Otherwise, do nothing.
  - Prepare a log file name. It should contain the date and time the file was created. Place *Format Date/Time String* function and connect to its *time format string* input the desired formatting scheme: *%Y-%m-%d\_%H-%M-%S* (which stands for 2022-05-22\_10-55-23 format).
  - Place *Concatenate Strings* function on the BD and configure it to have two inputs. The date/time string connect to the first input. Create a string constant, set it to *.log* and connect to the second *Concatenate Strings* input.
  - Place *Build Path* function on the BD. Connect the subfolder path created for Speech Recorder application to the *base path* input. Connect the log file name to the *name or relative path* input.

- Place *Open/Create/Replace File* function and connect the full path to log file to the *file path (use dialog)* input. Set *operation (0:open)* to *Create*.
- Open *#cluster\_ApplicationData.ctl* and add a new element to the cluster: *Logging file refnum*. You can obtain the necessary data type by creating an indicator for the *refnum* out output (*Open/Create/Replace File* function).
- Expand the *Bundle by Name* function inside the *Init* state in order to configure *Logging file refnum* item. Connect the expected file refnum wire to it.
- Remember about the error line – organize error inputs and outputs if needed.
- Go to the *Exit* state. Access the *Logging file refnum* from the application internal data cluster. Use *Close File* function to close and save the log file.



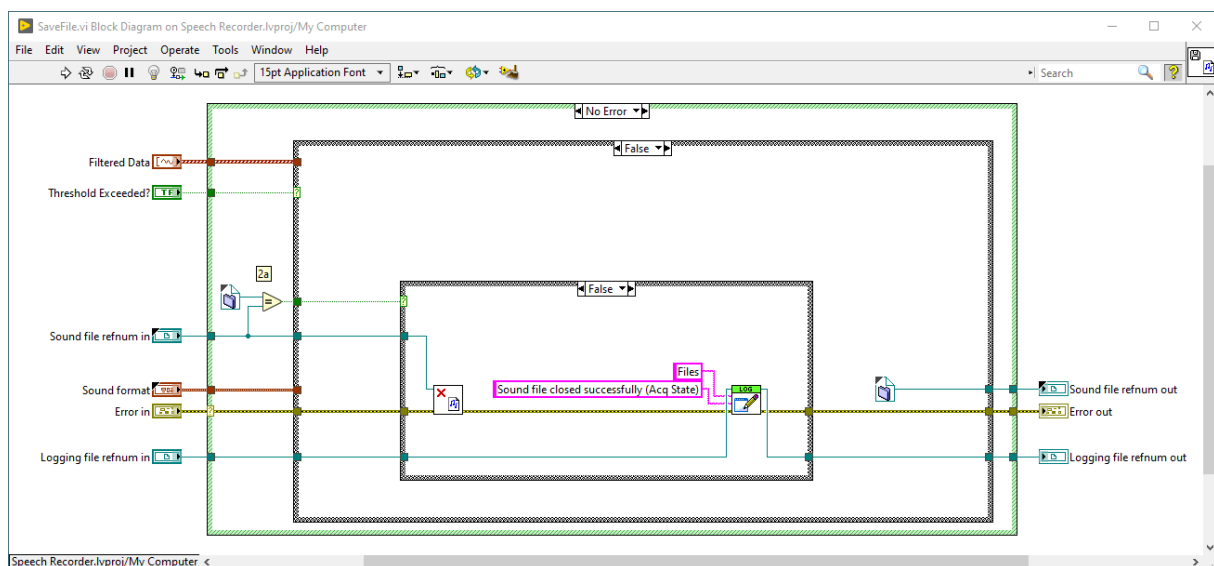
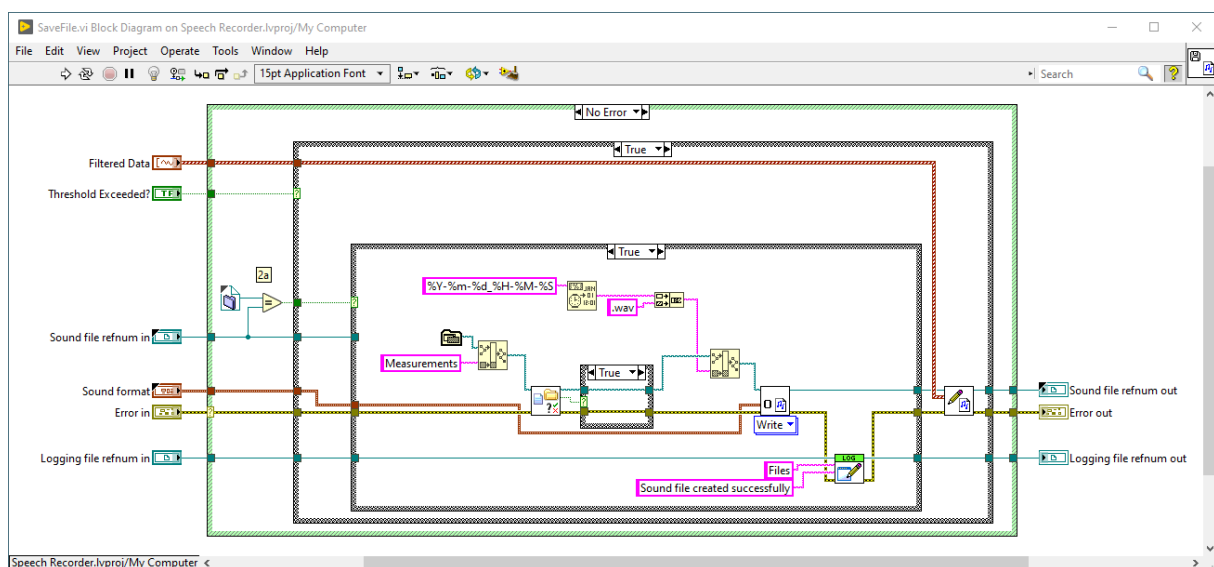
## 2. SubVI for logging events.

- Create a new VI in the *SubVIs* directory. Name it *AddNewLog.vi*. Open it and prepare controls and indicators for the *Logging file refnum* and the error lines. Connect all terminals to the CP.
- Prepare two string controls: one for *General Info* and one for *Details*. Set them up on the CP as well.
- Go to the BD and organize it so that the controls are on the left side and indicators are on the right side.
- Place *Case Structure* and configure it to work with the error input.
- In the *Error* case, simply pass the error and file refnum lines.
- In the *No Error* case, place *Get Date/Time in Seconds* and *Format Date/Time String* functions. Combine functions together and configure the format string to return the current date and time, including milliseconds (`%Y/%m/%d;%H:%M:%S%3u`).
- Place *Format Into String* function. Create a constant for *format string* input and set it to accept three input strings separated by “;” symbol. Then add the input of the fourth string without any separation. The result should be the following regular expression: `%s;%s;%s%s`.
- Expand the *Format Into String* inputs and connect the following values: date/time string, *General Info* and *Details*, and *End of Line Constant*.
- Place *Write to Text File* function and combine it with the error and refnum lines. Connect the *text* input with the corresponding *Format Into String* output.
- Clean-up the code if necessary.



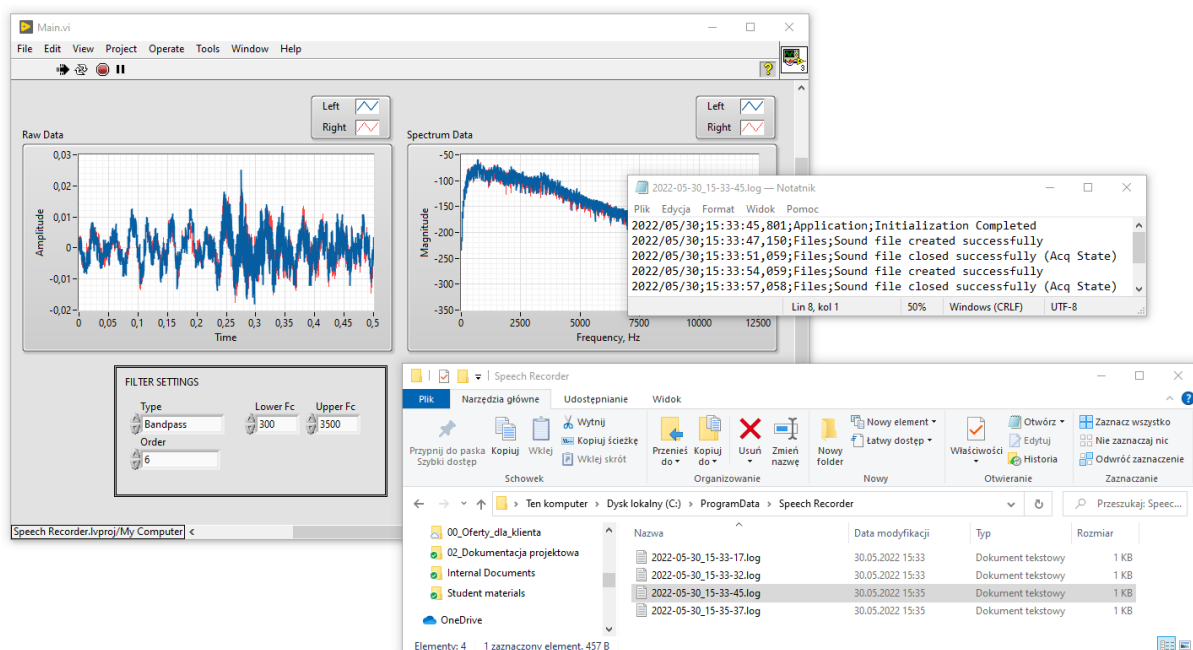
### 3. Log events.

- Open *SaveFile.vi*.
- Create a control and indicator for the *Logging file refnum* line – you can copy needed refnum object from any other place of the code. Update the CP.
- Go to the BD. Go to the case where a new sound file is created.
- After creating a new file, add *AddNewLog.vi* and configure it to log information about the successfully created audio file.
- Go to the case where the file is closed. Likewise, add a new log here stating that the sound file was closed successfully.



- Go to *Main.vi* and *Acquisition* state. Using the application internal data cluster and (un)bundling functions, connect the refnum line added to the *SaveFile.vi* function.
- Go to the *Error Handler* state. After clearing the error, log that this state has been entered and an error occurred.
- Log that application was initialized in the *Init* state.
- In the *Exit* state, log that the sound file has been closed and the application is shutting down.

4. Run the application, resolve all issues, and observe the event logging mechanism.



END OF EXERCISE