EG COURSES
SKILL CENTER

EG Courses

Graphical Programming Languages 2

WORKBOOK

Revision 1.0

GPL2 </>

Graphical
Programming
Languages 2

Table of Contents

General Information

This document provides step-by-step instructions for all Graphical Programming Languages 2 (GPL2) course exercises and projects. GPL2 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 GPL2 is a continuation of graphical programming languages learning path. It focuses on utilizing hands-on software development skills using the LabVIEW programming language.

All the materials needed to complete the exercises as well as the proposed solutions are available in the GitHub repository: www.github.com/EGCourses/GPL2.

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).

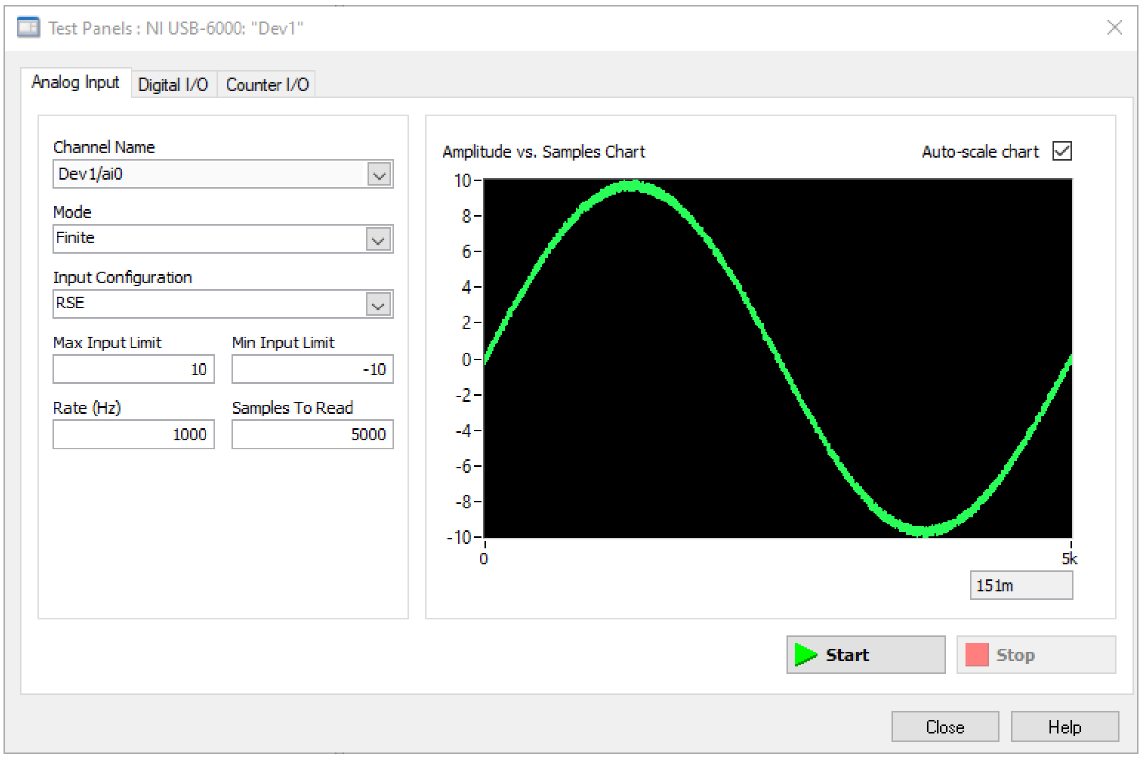
EXERCISE 1 NI MAX

GOAL

Use a device simulated in NI MAX application in order to read and write measurements. Test the device and prepare a simple VI for acquisition.

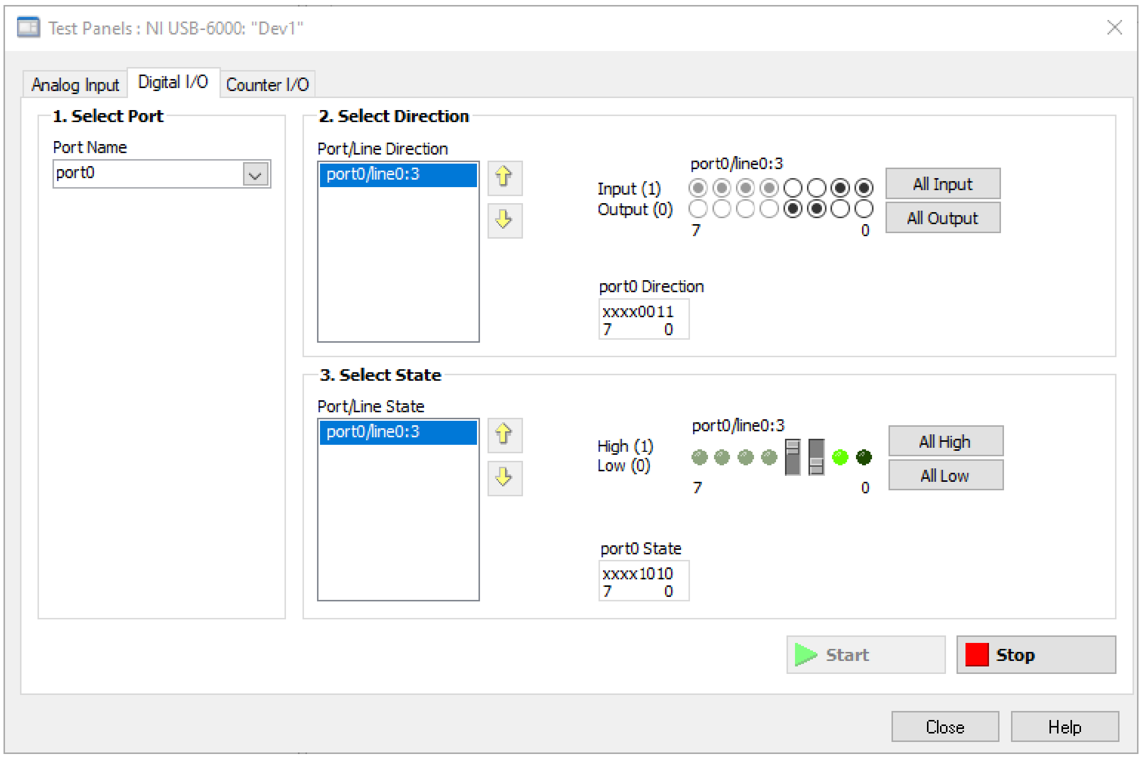
IMPLEMENTATION

1. Simulate a device.
   - Open NI MAX application and click RMB on the *Devices and Interfaces* item. Select *Create New…*
   - Double click on *Simulated NI-DAQmx Device or Modular Instrument*.
   - Search and select *USB-6000* card.
   - A new device should appear in the *Devices and Interfaces* section. From then on, this card can be treated exactly as physical one.
2. Test panels.
   - Test whether the device works properly using *Self-Test* option available in NI MAX in the top menu bar (after selecting the device).
   - Select *Device Pinouts* option to view inputs and outputs available on the card.
   - Select *Test Panels…* option in order to test specific inputs and outputs.
   - In the *Analog Input* tab, select analog channel name from which you want to acquire data.
   - Select an acquisition mode (e.g. *Continuous*).
   - Start measuring from the device. As this is a simulated card, a sine signal should be obtained.
   - Experiment with other settings available on this tab.



Channel Name: Dev1/ai0
Mode: Finite
Input Configuration: RSE
Max Input Limit: 10
Min Input Limit: -10
Rate (Hz): 1000
Samples To Read: 5000

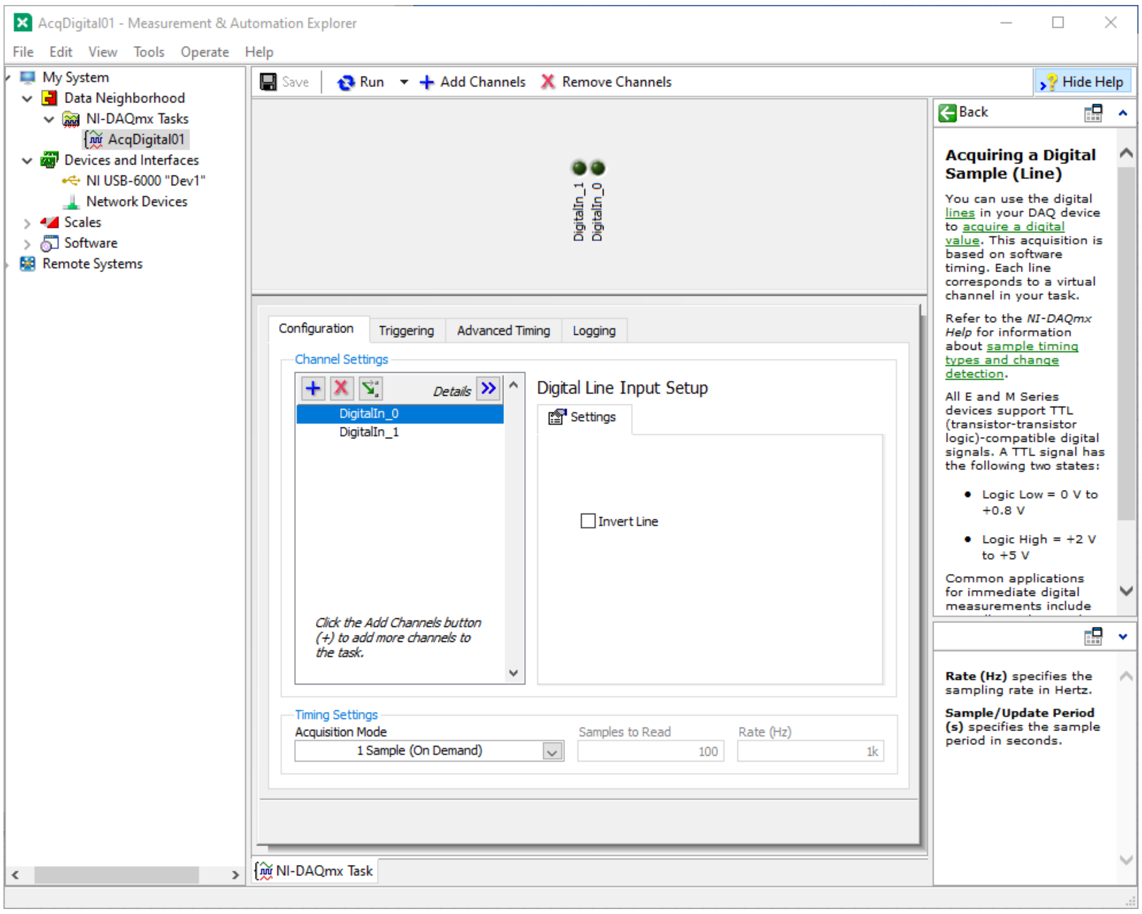
- Go to the *Digital I/O* tab.
- The selected device has only one digital I/O port which contains 4 pins. Each pin can be used as input or output.
- In the *Select Direction* section, you can configure pins direction (input or output).
- In the *Select State* section, you can either observe state of the inputs or set the states of the outputs.
- Try different configurations and start the measurement to see simulated states.



- Close the *Self-Test* window.

3. Create a task.

   - Select *Create Task…* option from the NI MAX toolbar.
   - From *Acquire Signals* section, select *Digital Input* and *Line Input* options.
   - Select the first two lines (0-1).
   - Click *Next* and name your task *AcqDigital01*. Click *Finish*.
   - The created task should appear in *Data Neighborhood* → *NI DAQmx Tasks* section. Once selected, a panel for acquisition configuration for these specific inputs should be displayed. After clicking the *Run* button, measurement should be started.



Digital Line Input Setup

DigitalIn_0
DigitalIn_1

Click the Add Channels button (+) to add more channels to the task.

Invert Line: ☐

Acquisition Mode	Samples to Read	Rate (Hz)
1 Sample (On Demand)	100	1k

Acquiring a Digital Sample (Line)

You can use the digital lines in your DAQ device to acquire a digital value. This acquisition is based on software timing. Each line corresponds to a virtual channel in your task.

Refer to the *NI-DAQmx Help* for information about sample timing types and change detection.

All E and M Series devices support TTL (transistor-transistor logic)-compatible digital signals. A TTL signal has the following two states:

- Logic Low = 0 V to +0.8 V
- Logic High = +2 V to +5 V

Common applications for immediate digital measurements include

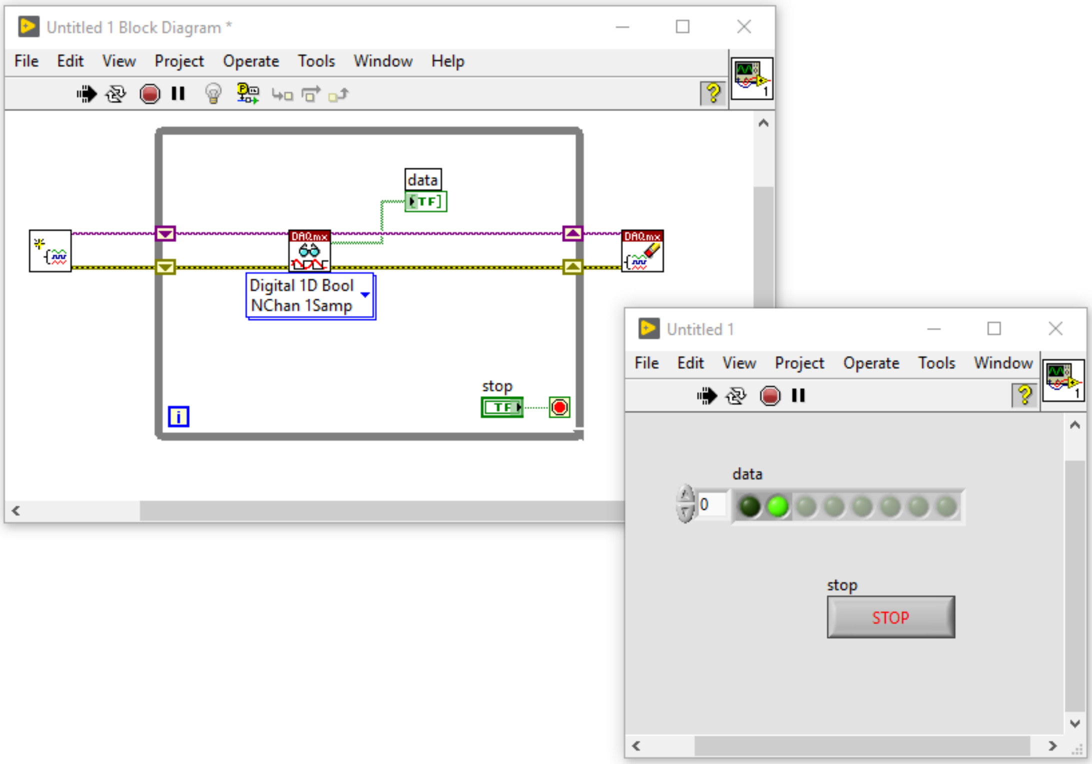
Rate (Hz) specifies the sampling rate in Hertz.

Sample/Update Period (s) specifies the sample period in seconds.

- ➢ Add another task for this device. Select *Generate Signals* → *Digital Output* → *Line Output*.
- ➢ Select two lines with indexes 2-3. Click *Next* and name the task *GenDigital23*.

4. Generate LabVIEW code.

- ➢ Open LabVIEW and create a new VI. Drag and drop the *AcqDigital01* task from NI MAX list to the BD of the VI.
- ➢ Click RMB on the task constant appeared on the BD and select *Generate Code* → *Configuration and Example* option from the context menu. Wait while the code is added to your VI.
- ➢ Find the data indicator on the FP and run the VI to see acquired data.
- ➢ Switch to the BD and draw *While Loop* around the read function.
- ➢ Create a control for stopping the loop.
- ➢ Run the VI and observe how the values are acquired from digital inputs.


Untitled 1 Block Diagram *
File Edit View Project Operate Tools Window Help
data
Digital 1D Bool NChan 1Samp
stop
Untitled 1
File Edit View Project Operate Tools Window
data
0
stop
STOP

➢ Similarly, you can repeat the previous steps to generate sample code for the *GenDigital23* task. In this task, you can set states for defined digital outputs.

END OF EXERCISE

EXERCISE 2 Let’s Talk

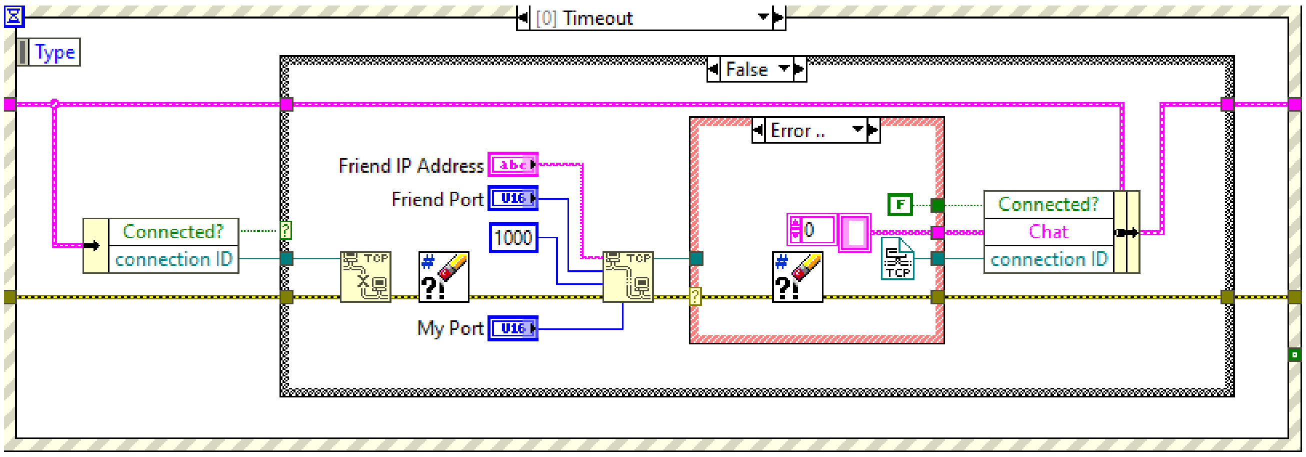
GOAL

Create an application that allows to chat between two users who run the application on different PCs. The application should connect to the specified user based on the IP address and port number via TCP protocol. The application should send and receive messages, and display them on the chat indicator.

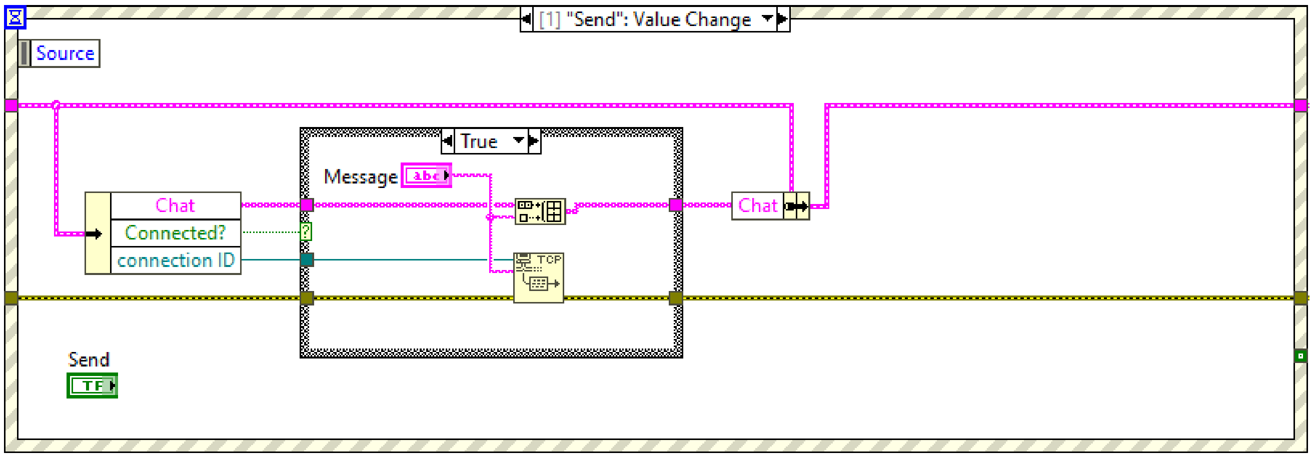
IMPLEMENTATION

1. Familiarize yourself with the exercise starting point.
   - Go to the *...\GPL2\Exercises\Exercise 2* location in course materials directory. Open *Messenger.vi*.
   - Analyze the purpose of GUI controls and indicators:
     i. *Chat* – displays all messages (received and sent),
     ii. *Message* – allows to enter a message to be send,
     iii. *SEND* – sends the message,
     iv. *STOP* – stops the application,
     v. *Friend IP Address* – specifies the IP address of the PC to which the application should connect,
     vi. *My Port* – defines the local port on which the connection should be established,
     vii. *Friend Port* – allows to enter a remote port on which another user has established a connection.
   - Go to the BD of the VI. Find red markers with letters placed in different parts of the code (A.). These markers point to places where some logic is missing.
   - Locate necessary TCP functions in the functions palette: *Functions* → *Data Communication* → *Protocols* → *TCP*. You can pin this palette if you want.
   - On the BD, go to the *[0] Timeout* → *True* case inside the WHILE. This piece of code periodically checks whether any messages were sent by the other user. Analyze how the *TCP Read* function is used here to listen to received messages.

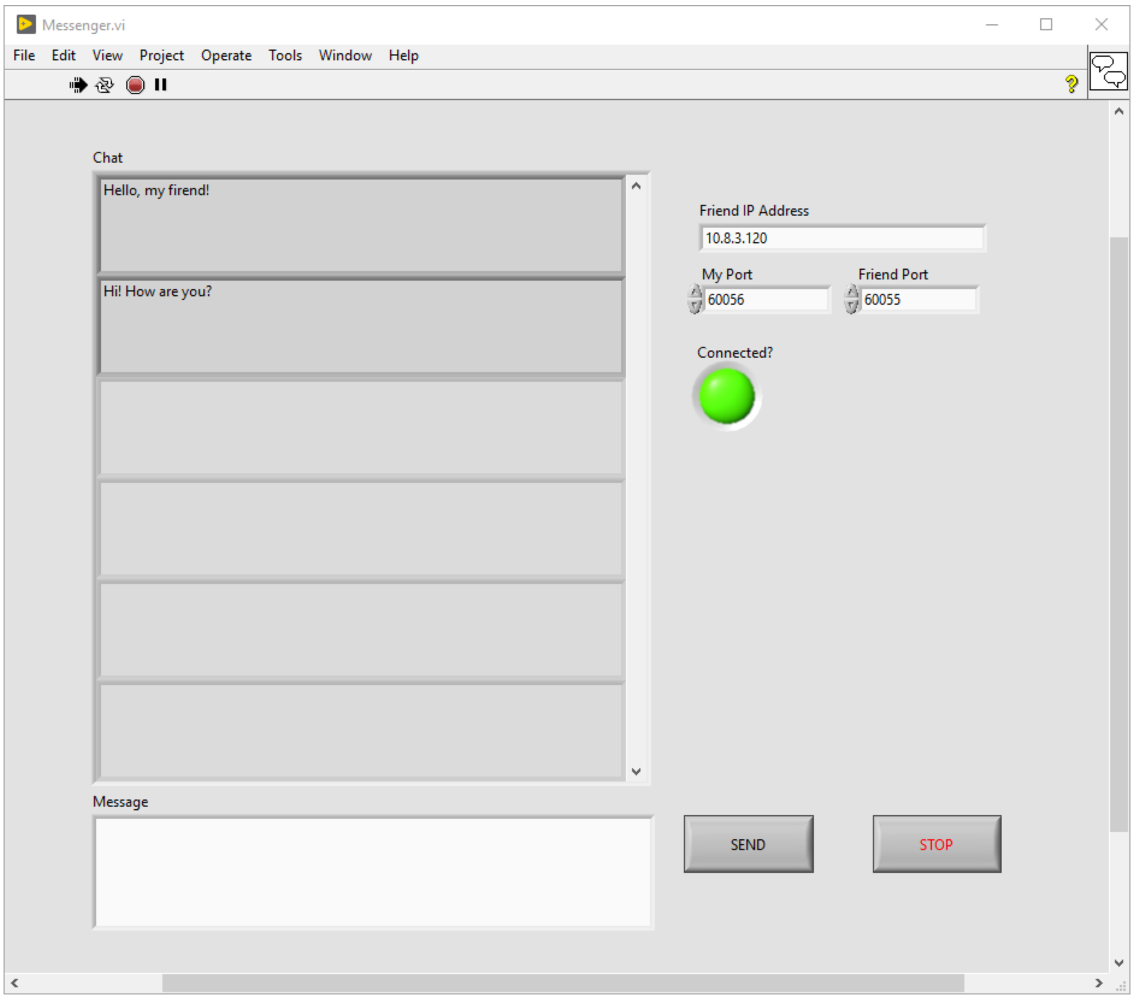
2. Implement a connection with another user.
   - Find A and B markers located in the *[0] Timeout* → *False* case. This code runs periodically, but only if the connection with the user has not yet been established (*Connected?* value is *False*).
   - Remove A marker and disconnect error wire before the *Clear Errors* function.
   - Use *TCP Close Connection* function in order to correctly close the previous connection handled by the *connection ID* reference. This part of code ensures that before opening a new connection, the last one is always closed on both TCP ends.
   - Connect the error wire.
   - Remove B marker and place *TCP Open Connection* function between the *Clear Errors* function and the inner case structure. Configure it with *Friend IP Address*, *Friend Port* and *My Port* data. Set the function timeout to 1000 ms.
   - Go to the *No error* case. Pass the *Connection ID* output through this case and connect it to the proper input of the bundle function located behind the inner case structure.
   - Go to the *Error* case and remove C marker. There should be an empty tunnel for the *connection ID*. Create a constant for this tunnel (remember: RMB opens proper context menu, where you can find all necessary options and items).



3. Implement a message sending mechanism.
   - Go to the *[1] "Send": Value Change* event, locate and delete D marker.
   - Inside the *True* case, use *TCP Write* function in order to send the *Message* string. Remember to use the *connection ID* reference.



4. Implement TCP connection closing.
   - Go to the *[2] "Stop": Value Change* event, locate and delete E marker.
   - Inside the *True* case, close the TCP connection handled by the *connection ID* reference (use *TCP Close Connection* function).
5. Test the application and fix all the issues, if needed.
   - Find another user to chat with via the application.
   - Check the IP address of your PC and give it to your partner (this value should be entered in the *Friend IP Address* control).
   - Similarly, enter the IP Address of your partner's PC in your application instance.
   - Configure both *My Port* and *Friend Port* values with your partner.
   - Run the VI and chat.



Hello, my firend!

Hi! How are you?

Friend IP Address: 10.8.3.120

My Port: 60056

Friend Port: 60055

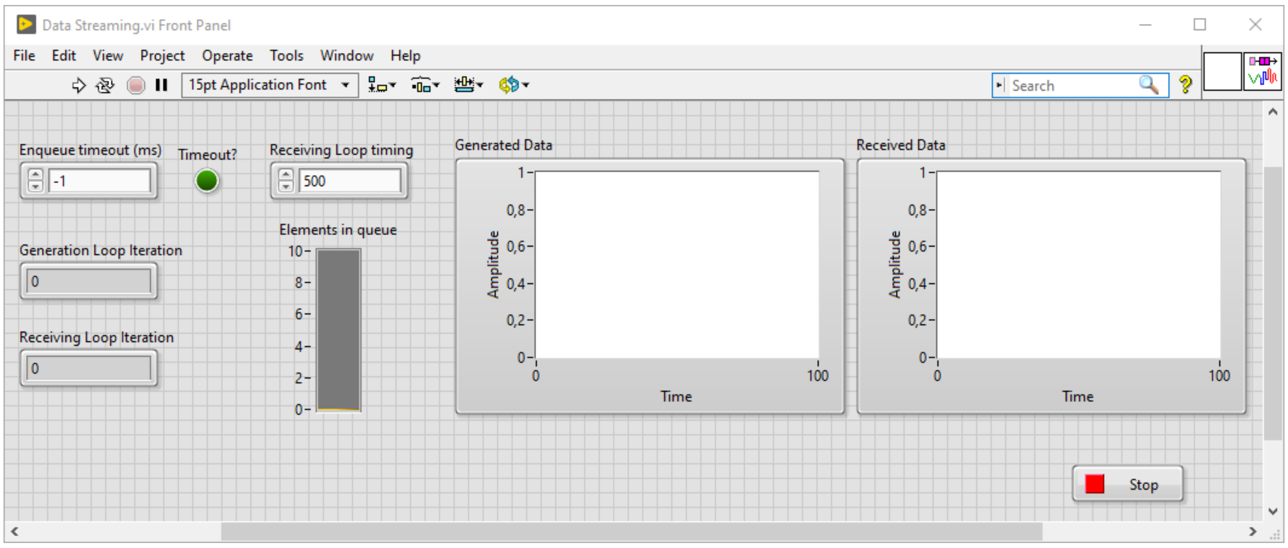
Connected?

END OF EXERCISE

EXERCISE 3 Streaming

GOAL

Create an application that streams data between two separate threads using a queue. One thread should be responsible for writing numeric data to the buffer and the other thread should receive this data. Add the ability to control the timing of the receiving loop. Observe the queue status to see possible overflow.



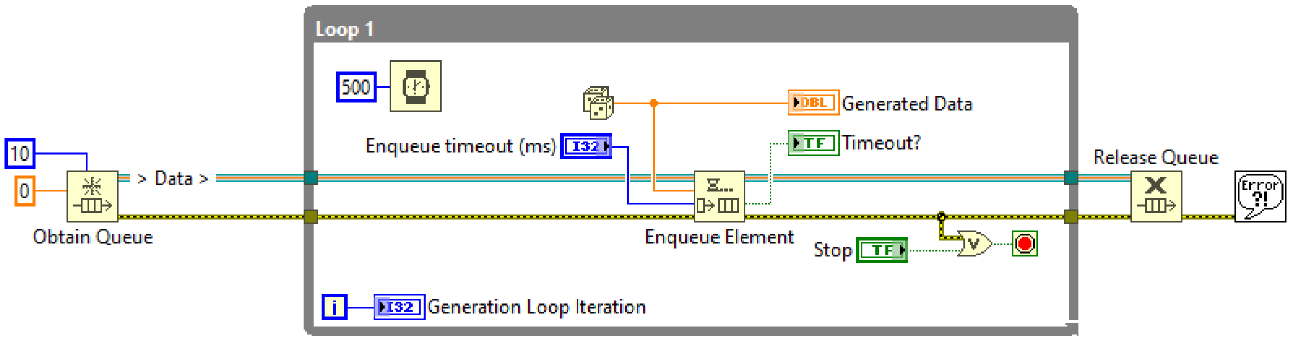
IMPLEMENTATION

1. Prepare main controls and indicators.
   - Create a new VI and save it.
   - Go to the FP and create a button for stopping the VI.
   - Place two *Waveform Chart* objects (*Controls* → *Graph* → *Waveform Chart*) and name them *Generated Data* and *Received Data*.
   - Create *Numeric Control* for controlling the timing of the receiving loop. You can name it *Receiving Loop Timing*. From the context menu, set its representation to U32.
   - From the controls palette, select *Numeric* → *Vertical Progress Bar* and name it *Elements in Queue*. Open the context menu for this indicator and go to *Scale* →

Style option and select one of the scale display settings. Set the maximum scale value to 10.

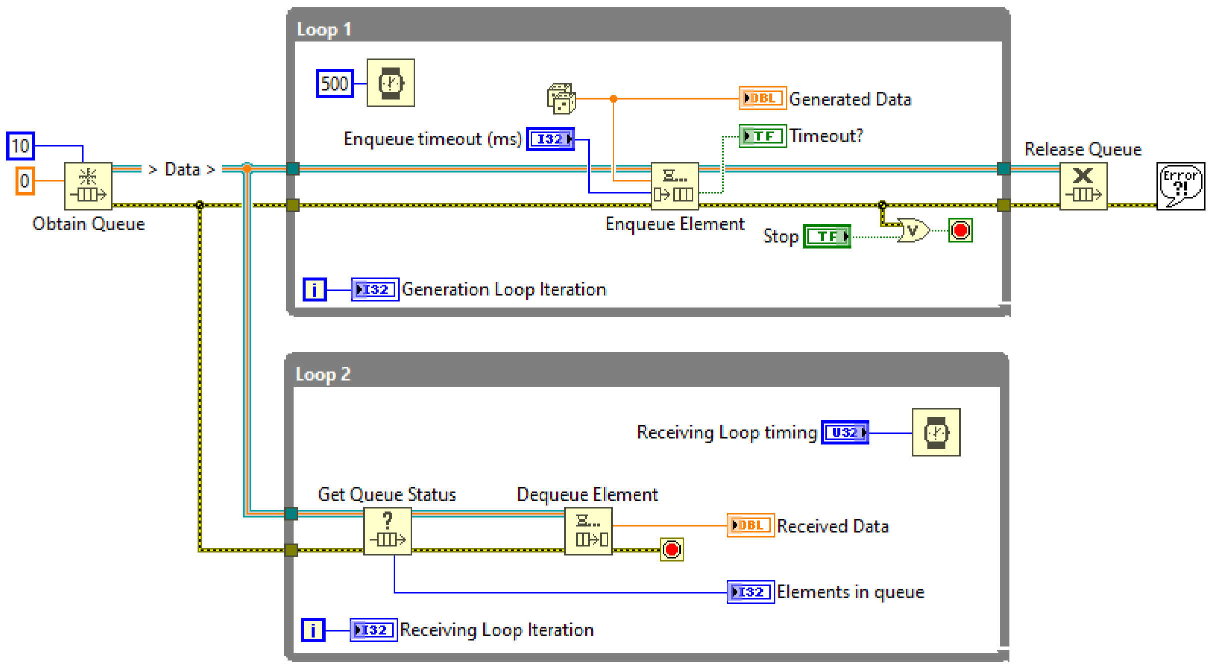
2. Implement data generation.
   - Switch to the BD and organize terminals: inputs should be placed on the left, and outputs should be placed on the right.
   - Draw *While Loop*. Use *Wait (ms)* function to ensure that the loop is executed every 500 ms.
   - Create an indicator to observe the *Loop Iteration* value (i) and name it *Generation Loop Iteration*.
   - Open the function palette and go to *Data Communication* → *Queue Operations*. Pin the palette.
   - Select *Obtain Queue* function and place it just before the WHILE.
   - Select *Enqueue Element* function and place it inside the WHILE.
   - Select *Release Queue* function and place it just after the WHILE.
   - Connect all new functions from left to right using *queue* and *error* lines.
   - Configure the *Obtain Queue* function by setting the *max queue size* value to 10 and connecting *DBL Numeric Constant* to the *element data type* input.
   - Configure the *Enqueue Element* function. Create a control for the *timeout in ms (-1)* input and name it *Enqueue timeout (ms)*. Create an indicator for the *timeout?* output.
   - Switch to the FP and find the *Enqueue timeout (ms)* control. Set its value to -1 and select *Data Operations* → *Make Current Value Default* from the context menu. Go back to the BD.
   - Generate a random number using *Numeric* → *Random Number (0-1)* function. Add it to the queue (connect the *element* input of the *Enqueue Element* function).
   - Find the terminal for *Generated Data* graph and move it inside the WHILE. Connect the random number generated in the previous step to it.
   - Configure the *Loop Condition* terminal (●) so the loop stops if the *Stop* button is clicked or the *Enqueue Element* function returns an error (use *Or* function and connect both error wire and *Stop* button output to it).

- Open the functions palette and select *Dialog & User Interface → Simple Error Handler* function. Place it after the *Release Queue* function and connect the error wire to it.

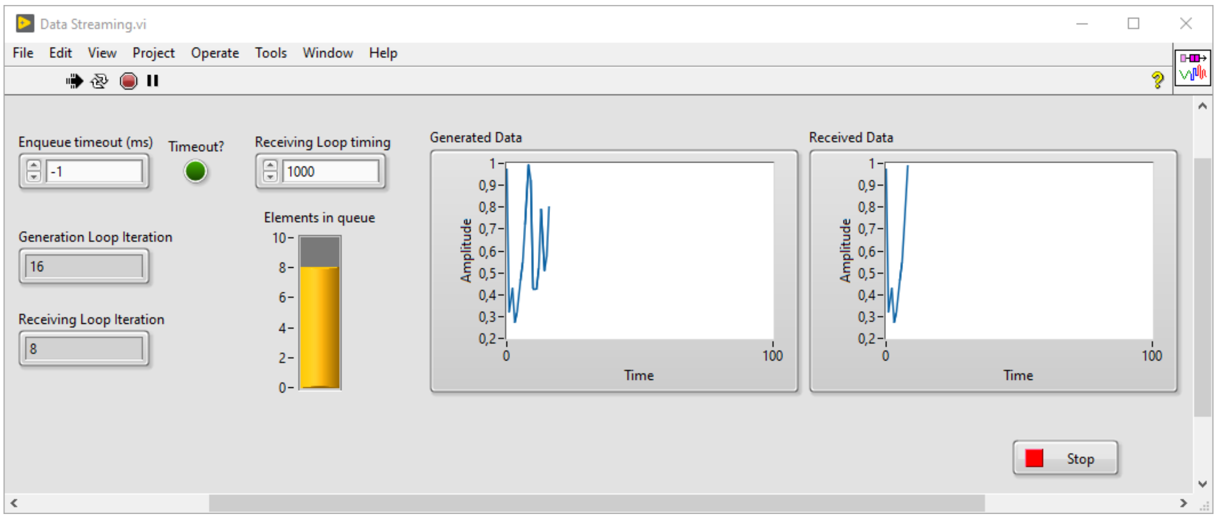


3. Implement data receiving mechanism.
    - Draw another *While Loop* below the previous one. This thread will be responsible for reading the data from the queue.
    - Connect *queue* reference and *error* wires from the *Obtain Queue* function to the edge of the new WHILE.
    - Select *Get Queue Status* function from the functions palette and place it inside the bottom WHILE. Connect function's inputs with wires created in previous step.
    - Find the *Elements in Queue* terminal created in the very first step and move it to the inside of the bottom loop. Connect to it *# elements in queue* output of *Get Queue Status*. If the coercion dot appeared, use proper debug techniques to check and resolve the reason. Set the correct representation of the indicator, if needed.
    - Place *Dequeue Element* function inside the loop. Connect it with the outputs of *Get Status Queue* function using both queue and error wires.
    - Find the *Received Data* terminal and move it to the inside of the bottom loop. Connect the *element* output of *Dequeue Element* function to it.
    - Configure the loop to stop if error occurs – connect the *error out* output from *Dequeue Element* function to the *Loop Condition* (🔴) terminal.
    - Place *Wait (ms)* function inside the second WHILE. Configure it with the control prepared in the very first step (*Receiving Loop timing*). Set control's representation to U32.

- Create an indicator for the *Loop Iteration* terminal (i) and name it *Receiving Loop Iteration*.



4. Clean up and test the VI.
   - Clean-up the code on the BD.
   - Switch to the FP and organize all controls and indicators. Make sure that all of them have meaningful names.
   - Run the VI and test it. Fix all detected issues.
   - Experiment with different *Enqueue timeout (ms)* and *Receiving Loop timing* settings. Observe the graphs and queue overflow.



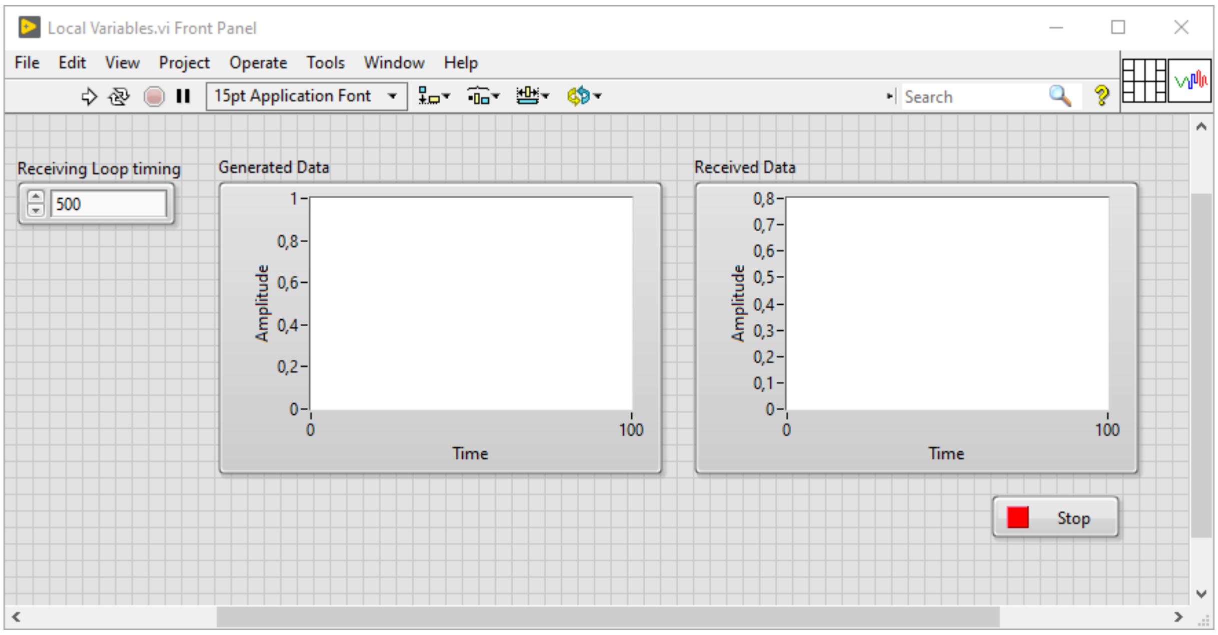
Enqueue timeout (ms): -1
Timeout?
Receiving Loop timing: 1000
Elements in queue
Generation Loop Iteration: 16
Receiving Loop Iteration: 8
Generated Data
Received Data

END OF EXERCISE

EXECRISE 4 Local Variables

GOAL

Prepare a VI that transfers data from one loop to another using a local variable. Both threads should stop after clicking one common button.

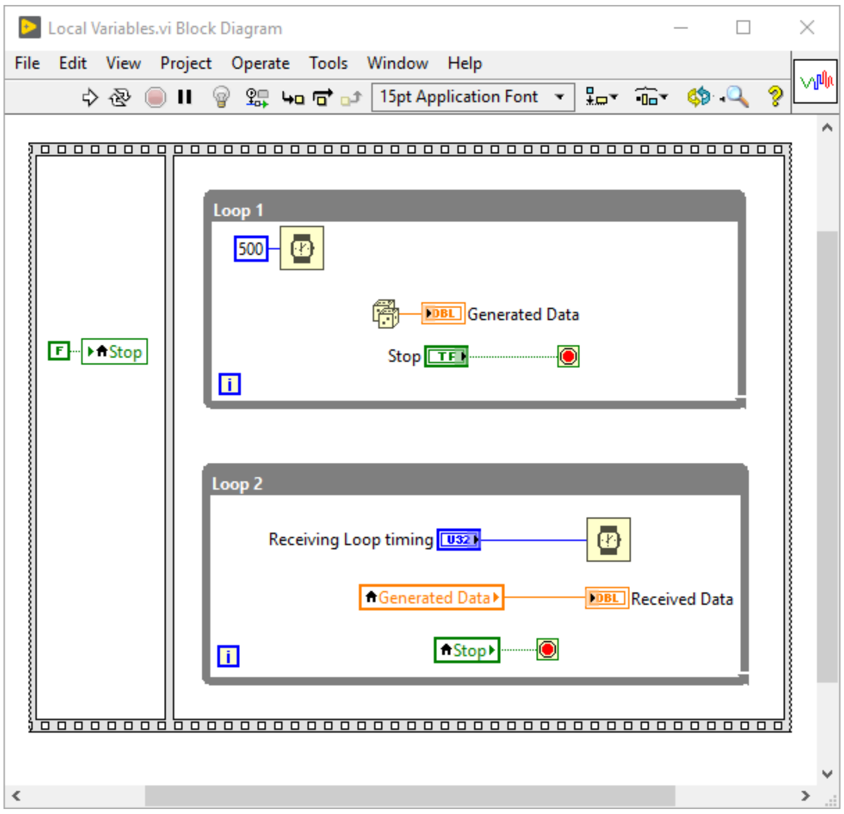


IMPLEMENTATION

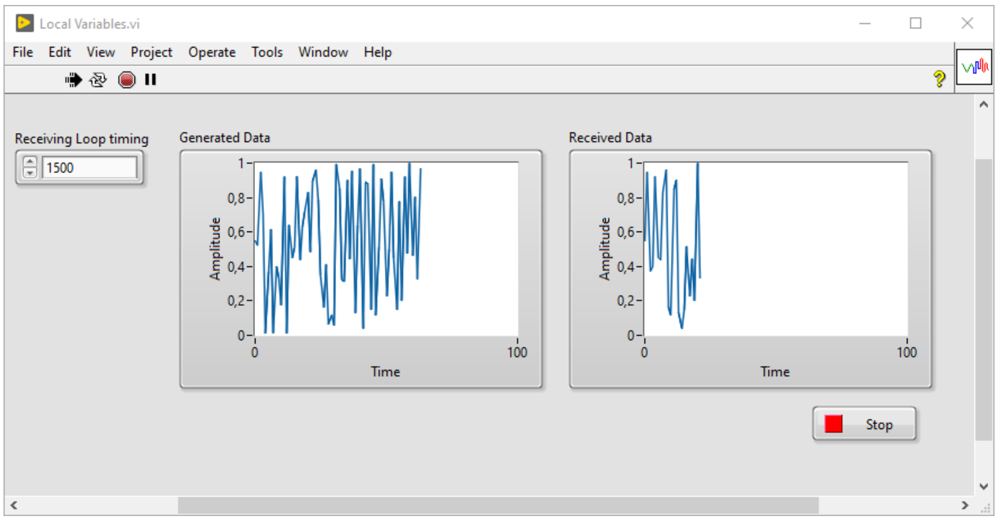
1. Familiarize yourself with the exercise starting point.
   - ➢ Go to the *...\GPL2\Exercises\Exercise 4* location in course materials directory. Open *Local Variables.vi*.
   - ➢ Open the BD and analyze it.
2. Implement data receiving mechanism.
   - ➢ Create a local variable for *Generated Data* object – open the functions palette and select *Data Communication* → *Local Variable*. Place it inside the receiving loop.
   - ➢ Click LMB on the local variable and select *Generated Data* as the source.
   - ➢ Open variable's context menu and select *Change to Read* option. Connect created local variable with the *Received Data* graph.

- Create a control for configuring receiving loop timing – place *Wait (ms)* function and create a control for its input. Name it *Receiving Loop timing*.

3. Implement application shutdown procedure.
    - Inside the data generating loop, create a control for the *Loop Condition*.
    - Create a local variable for the new button – open the context menu on the *Stop* control and select *Create → Local Variable* option. Place it inside the receiving loop.
    - Change the mode of the newly created local variable to the read mode. Connect it to the *Loop Condition* of the second loop.
    - Notice the broken *Run* arrow. Click on it to see the list of errors. The major error should state that the mechanical action of the *Stop* button is incompatible with local variables.
    - To fix the error, open the *Stop* button's context menu and select *Mechanical Action → Switch when Released* setting.
4. Initialize local variable.
    - Run the VI. Click *Stop* button and run the VI again – notice that the application was shut down immediately. Why?
    - Open the functions palette and select *Structures → Flat Sequence Structure*. Draw it around both loops.
    - Click RMB on the sequence structure edge and select *Add Frame Before* from the context menu. Make more space inside the new frame using LMB and *Ctrl* combination.
    - Create another local variable for the *Stop* button. Place it inside the first sequence frame. Create a constant for it and set it to *False*. This way the variable is properly initialized before the main code executes.



5. Clean-up and test the VI.
   - ➢ Organize both BD and FP to meet good development practices and standards.
   - ➢ Run the VI and experiment with receiving loop timing. Observe data loss when the receiving loop runs slower than the generating loop.

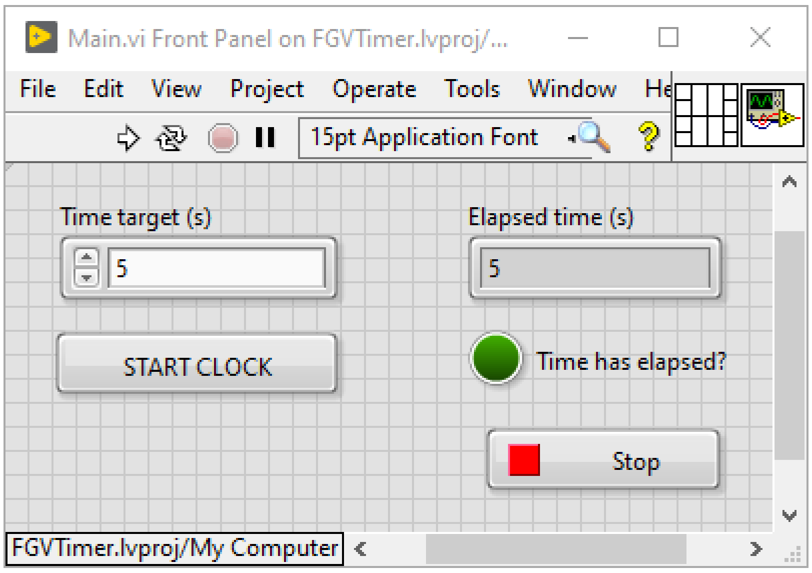


END OF EXECRISE

EXERCISE 5 Timer

GOAL

Create a timer application that counts the elapsed time since the timer started and displays if a specific time target has been reached.


Main.vi Front Panel on FGVTimer.lvproj/...
File Edit View Project Operate Tools Window He
15pt Application Font
Time target (s)
5
Elapsed time (s)
5
START CLOCK
Time has elapsed?
Stop
FGVTimer.lvproj/My Computer

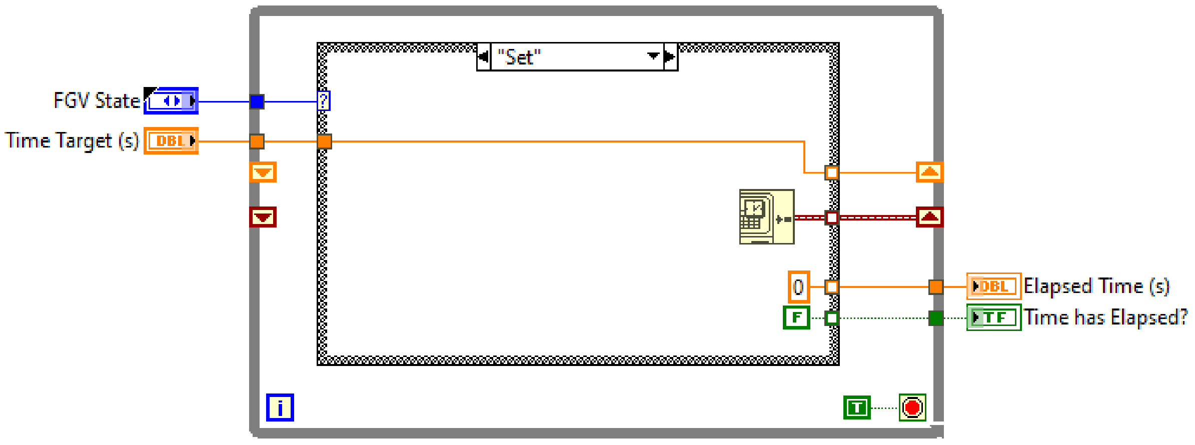
IMPLEMENTATION

1. Familiarize yourself with the exercise starting point.
   - Go to the ...\GPL2\\Exercises\\Exercise 5 location in course materials directory. Open *FGVTimer* project.
   - Open *Main.vi* and recognize the purpose of all controls and indicators placed on the FP.
   - Switch to the BD and analyze it.
2. Prepare an enum for FGV State.
   - Go to the project tree and open the *Controls* virtual folder context menu. Select *New* → *Type Definition*.
   - Place *Enum* control on the newly opened FP (*Controls palette* → *Ring & Enum* → *Enum*). Name it *FGV State*.

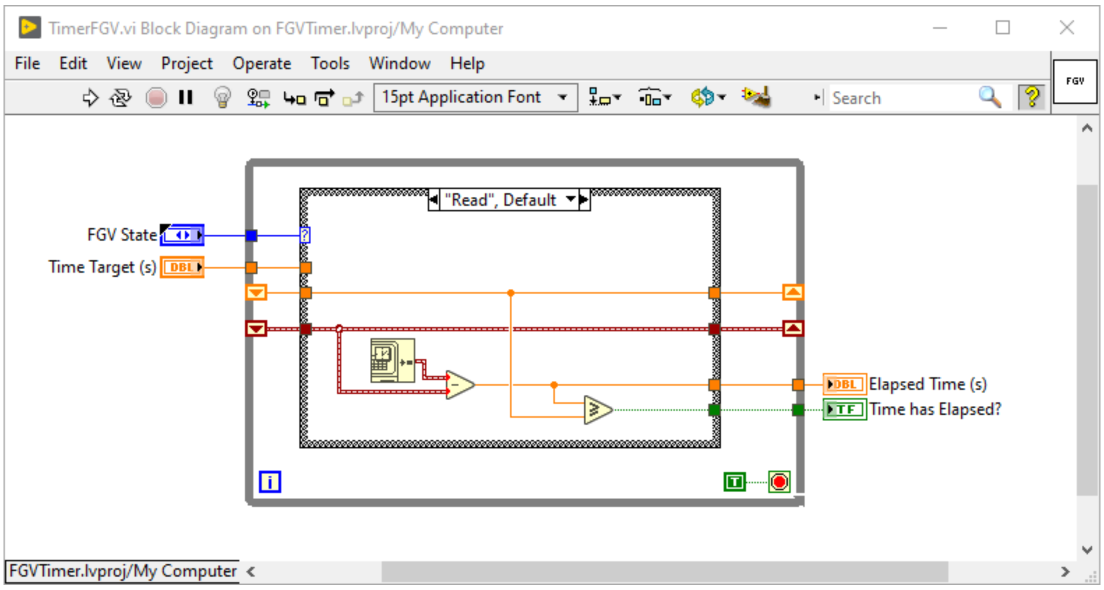
- From the enum context menu, select *Edit Items…* and define two new items: *Read* and *Set*. Apply the changes.
- Save the typedef to disk in the *Controls* directory in the project location. Name it *FGV_State.ctl*.

3. Crete Functional Global Variable (FGV).
    - Create a new VI under *SubVIs* virtual folder in the project tree. Save it to disk in the corresponding *SubVIs* directory in the project location. Name it *TimerFGV.vi*.
    - Drag *FGV_State.ctl* object instance to the FP.
    - Create *Numeric Control* for setting time target. Name it *Time Target (s)*. The numeric representation should be DBL.
    - Copy the control created in the previous step and rename it to *Elapsed Time (s)*, and change it to the indicator by selecting appropriate option from the context menu.
    - Place *Round LED* indicator on the FP. Name it *Time has Elapsed?*.
    - Configure the CP (Connector Pane) by connecting created controls and indicators to it. Modify the icon as well.
    - Switch to the BD. Move all controls to the left and all indicators to the right. Draw *While Loop* in the center.
    - Place *True Constant* and connect it to the *Loop Condition* terminal – the loop will execute indefinitely.
    - Draw *Case Structure* inside the WHILE. Configure it with the *FGV State* enum (connect this enum to *Case Selector*).
    - Open the context menu on WHILE’s edge and select *Add Shift Register* option. A shift register for an undefined data type should be created.
    - Inside the *Set* case, place *Get Date/Time In Seconds* function. Connect the outputted timestamp to the right shift register.
    - Connect the *Time Target (s)* control to the CASE’s left edge.
    - Add another shift register to the WHILE.
    - Inside the *Set* case, wire the *Time Target (s)* value through it, then connect it to the right shift register.
    - Notice that both of the left shift register inputs should remain uninitialized.

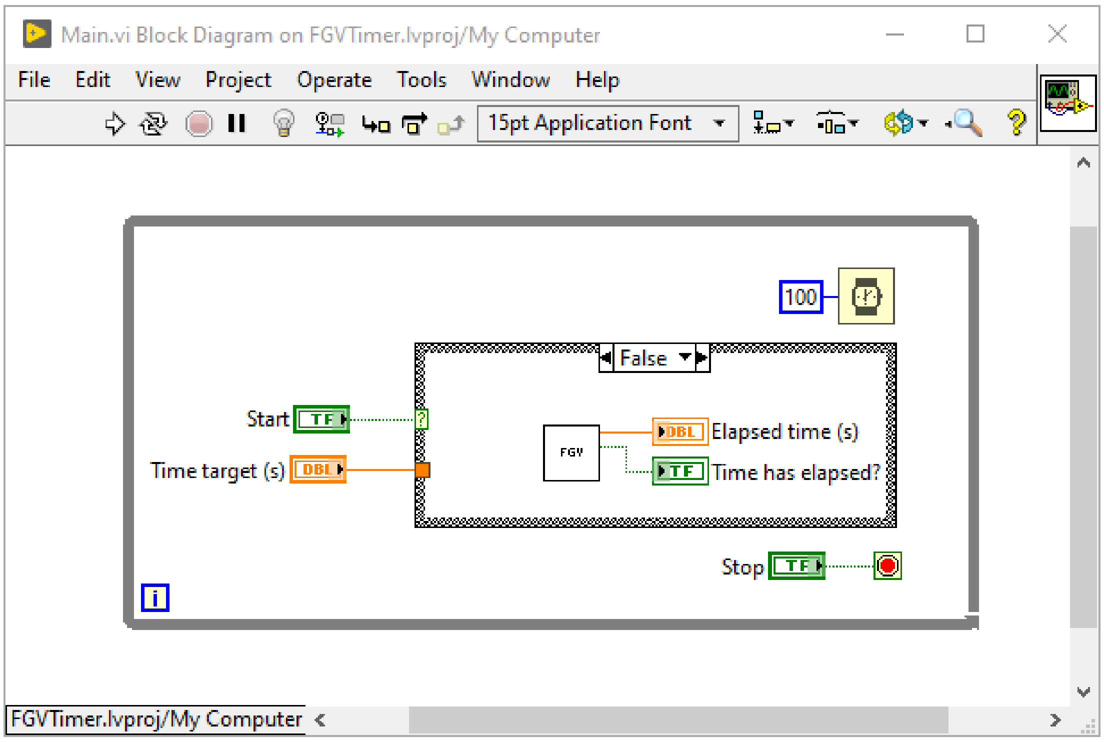
- Inside the *Set* case, place *DBL Numeric Constant* and connect it to the *Elapsed Time (s)* indicator.
- Similarly, place *False Constant* and connect it to the *Time has Elapsed?* indicator.



- Go to the *Read* case and pass the timestamp value from the left shift register through the case and connect it to the empty tunnel of the correct type.
- Similarly, do the same with another shift register that holds time target value.
- Inside this case, place another *Get Date/Time In Seconds* function.
- Calculate a difference between the timestamp acquired in the *Read* state and the timestamp kept by the shift register (use *Subtract* function). This difference represents how many seconds elapsed from the current moment and the moment of starting the timer.
- Connect the time difference calculated in the previous step to the empty tunnel prepared for the *Elapsed Time (s)* indicator.
- Use *Grater or Equal?* function in order to compare whether the elapsed time is greater than time target. Connect the comparison output to the tunnel prepared for the *Time has Elapsed?* flag.


TimerFGV.vi Block Diagram on FGVTimer.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
"Read", Default
FGV State
Time Target (s)
Elapsed Time (s)
Time has Elapsed?
FGVTimer.lvproj/My Computer

4. Finish the implementation of *Main.vi*.
   - Open *Main.vi* and go to the *True* case (executed when the *Start* button is clicked).
   - Place *TimerFGV.vi* prepared by you inside this case. Create a constant for *FGV State* enum and set it to *Set*.
   - Connect the *Time Target (s)* control to the appropriate FGV input.
   - Go to the *False* case and place *TimerFGV.vi* there as well.
   - Create necessary FGV outputs and connect them with the prepared indicators.


Main.vi Block Diagram on FGVTimer.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
100
False
Start
Time target (s)
FGV
Elapsed time (s)
Time has elapsed?
Stop
i
FGVTimer.lvproj/My Computer

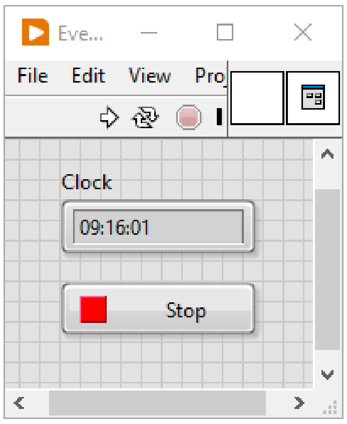
5. Test the application and fix all detected issues.

END OF EXERCISE

EXERCISE 6 Events

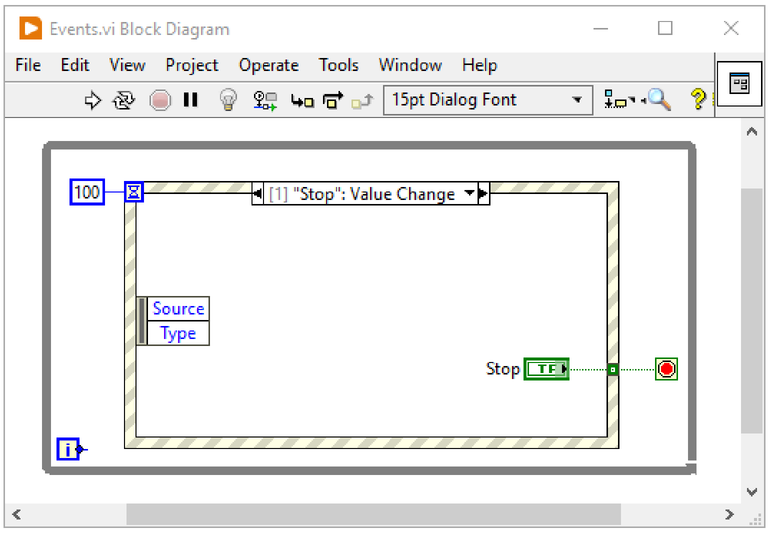
GOAL

Create an application that displays current time and stops at the click of a button. The application should not allow to close the panel with the widow's native X-button, and should display pop-up in this case. The application architecture should be event driven.

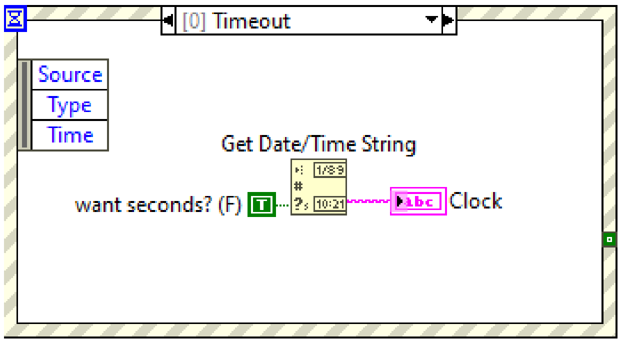


IMPLEMENTATION

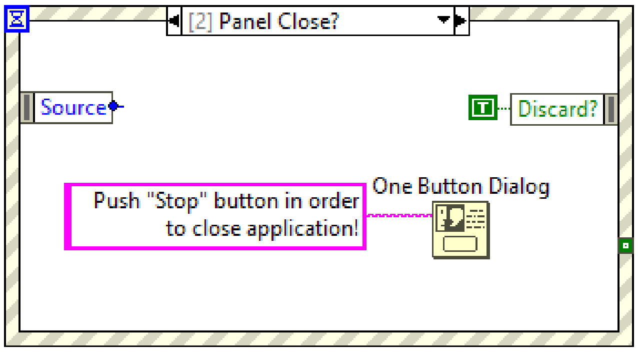
1. Create a new VI and save it as *Events.vi*.
2. Prepare the application architecture.
   - On the FP, prepare *Stop* button and *String* indicator for the clock display.
   - Go to the BD and draw *While Loop*.
   - Inside the WHILE, draw *Event Structure* (we will refer to it later in this manual using the word EVENT).
   - Open the EVENT context menu and select *Add Event Case* option.
   - Inside the *Edit Events* window, select *Controls* → *Stop* from *Event Sources* list. On the right panel, select *Value Change* option. Click *OK* to add a new event.
   - Move the *Stop* button inside the created event case. Place there *True Constant* and connect it to the *Loop Condition* terminal.
   - Find the *Event Timeout* terminal (⌛) and create a constant for it. Set the timeout to 100 ms.



3. Events implementation.
   - Go to the *Timeout* event of the structure and place *Get Date/Time String* function in order to get current time. Create a constant for *want seconds? (F)* input and set it to *True*.
   - Move the string indicator prepared for the clock display to the event and connect it with the *time string* output of the function.



   - Add another event case to the EVENT by using proper context menu. Inside the *Edit Events* window, select <*This VI*>. Then, select *Panel Close?* option on the right panel and click *OK*.
   - Create a constant for *Discard?* terminal and set it to *True*. This allows the application to ignore the default behavior programmed for this event – in other words, the window will not close.
   - Inside this event case, use *One Button Dialog* function in order to display a message that the application can be closed only by clicking the *Stop* button.



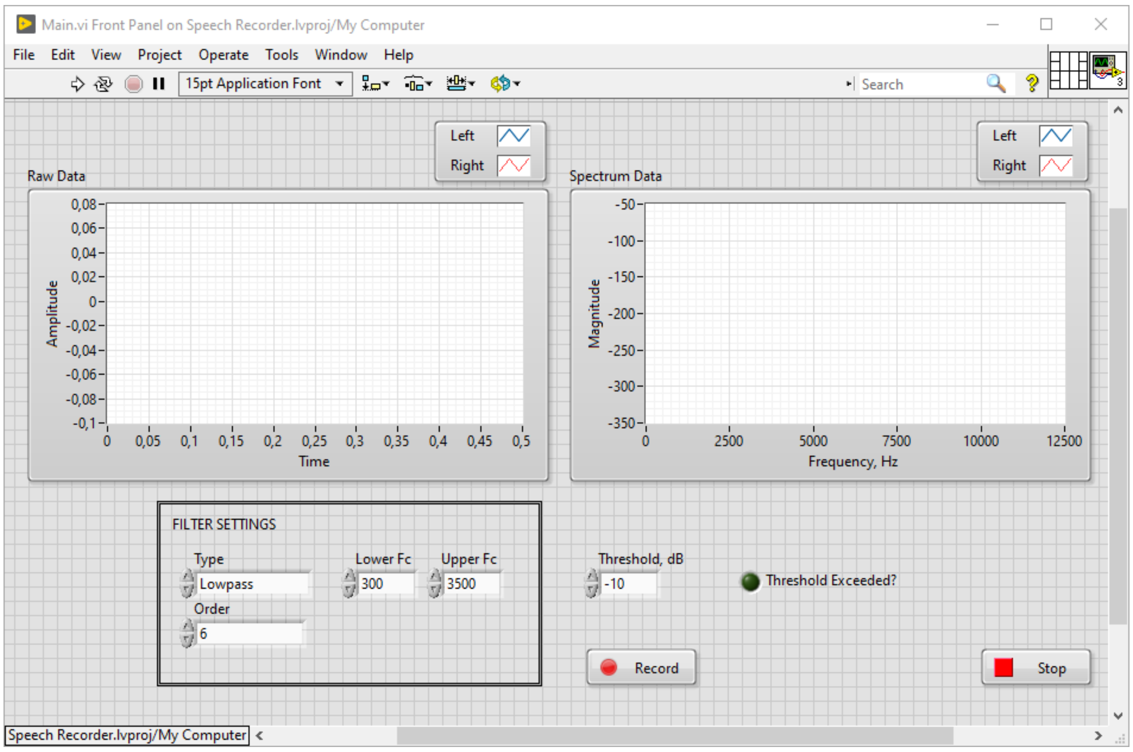
4. Test the VI and fix all detected issues.

END OF EXERCISE

EXERCISE 7 State Machine with Events

GOAL

Modify the *Speech Recorder* project by adding some events to its architecture: add the ability to start and stop the measurement by the user. The application should be idle until the user launches the recording.

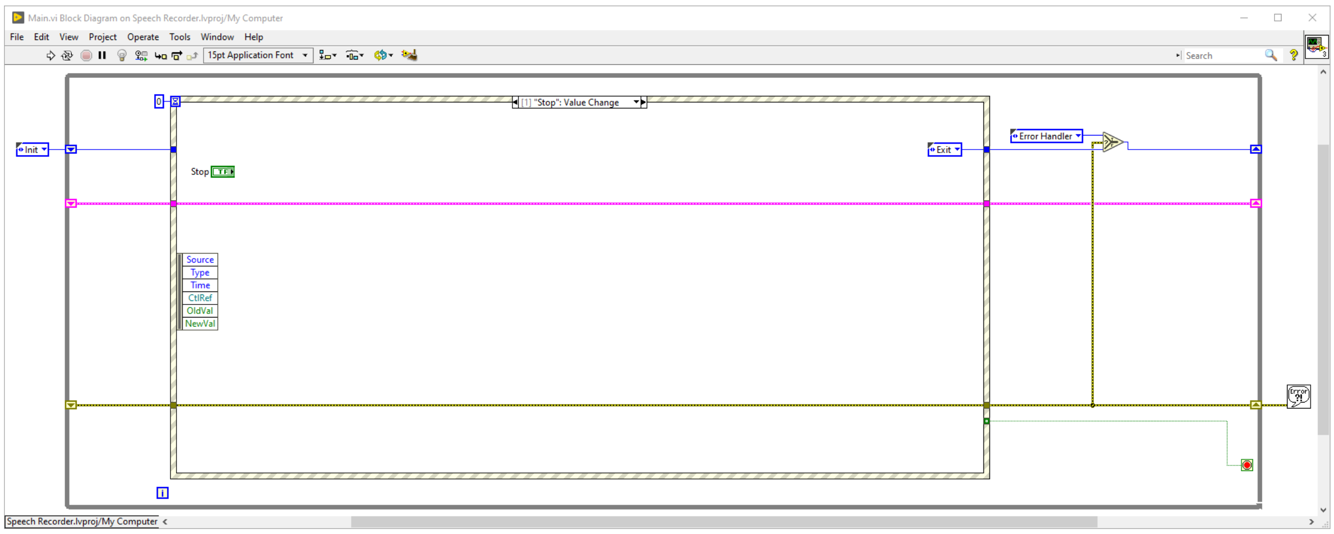


IMPLEMENTATION

1. Go to the *...\GPL2\Exercises\Exercise 7* location in course materials directory. Open *Speech Recorder* project and explore it.
2. Open *Main.vi* and recognize the purpose of all controls and indicators placed on the FP.
3. Implement an event driven shutdown procedure for the application.
   - In *Main.vi*, go to the BD.
   - Disconnect the *Stop* button from the *Select* function located on the right side of the WHILE and remove this *Select* function. Wire the value returned by the

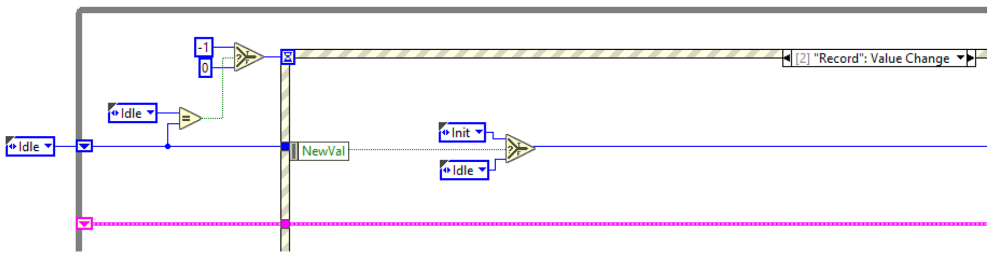
remaining *Select* function (the one configured with the error wire) to the shift register. Remove any redundant constants and broken wires.

- Make more space around the CASE and draw *Event Structure* so the CASE is inside the event. Clean-up the code.
- Create a constant for *Event Timeout* (⌛) and set it to 0.
- Add a new event case for the *Stop* button on *Value Change*.
- Inside the new event case, pass the application data cluster and the error wires through it.
- Create a constant for the next requested application state and set it to *Exit*. Connect it to the appropriate tunnel on the right edge of the structure.


Main.vi Block Diagram on Speech Recorder.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
[1] "Stop": Value Change
Init
Stop
Exit
Error Handler
Source
Type
Time
CtlRef
OldVal
NewVal
Speech Recorder.lvproj/My Computer

4. Implement starting and stopping the measurement.
   - Open *#enum_State.ctl* type definition. Add a new state to the enum – select *Add Item After/Before* or *Edit Items…* from its context menu. The new state should be named *Idle*. Save and close the type definition.
   - Return to *Main.vi* and go to the FP.
   - Open the controls palette and select *Silver* → *Boolean* → *Buttons* → *Record Button*.
   - Change the mechanical action of the newly placed button to *Switch When Released*.
   - Go to the BD and add a new event case for the *Record* button *Value Change*.
   - Pass the application data and error wires through the new case.

- Find *Event Data Nodes* located on the left edge of the EVENT. Check whether *NewVal* property is visible. If not, you can click LMB on the node and select desired property. You can also drag the upper or lower edge to expand the nodes list, if needed.
- Use *Select* function in order to request desired application state depending on the *NewVal* value. If it is True, *Init* state should be passed. If it is False, *Idle* state should be passed. Connect the output value to the appropriate tunnel on the right edge of the EVENT.
- Change the initial value of the application state (it can be found near the left shift register used to transmit application state data) from *Init* to *Idle*.
- Make more space before the EVENT. Disconnect the current timeout value (0) from the terminal.
- Staying before the EVENT, use *Equal?* function to check if the current state is equal to *Idle*. Place *Select* function and connect the result of the comparison to it.
- Set up the *Select* function to return -1 if the current state is equal to *Idle* and 0 otherwise. Connect the result of this logic to the *Event Timeout* terminal. As a result, an application in the Idle state will wait indefinitely for strictly defined user action.



5. Test the application and fix all detected issues.

END OF EXERCISE

EXERCISE 8 Application Distribution

GOAL

Prepare builds for the *Speech Recorder* project. Create an executable version of the application (*.exe file) and an installer package.

IMPLEMENTATION

1. Go to the *...\GPL2\Exercises\Exercise 8* location in course materials directory. Open *Speech Recorder Event* project and explore it.
2. Prepare an executable.
    - ➢ In the project tree, open the context menu on *Build Specifications* item and select *New → Application (EXE)* option. The build configuration window should appear.
    - ➢ View the *Category* section located on the left side of the build creator. Take a look on basic categories like: *Information*, *Source Files*, *Icon*, *Version Information*, etc.
    - ➢ Open the *Information* category. Change the *Build specification name* and *Target filename* settings at your discretion (e.g. *Speech Recorder* and *SpeechRecorder.exe*).
    - ➢ Define the *Destination directory* – this is a physical location where the application executable will be created.
    - ➢ Open the *Source Files* category. Click on *Main.vi* and add it to the *Startup VIs* section by clicking appropriate arrow button. This is the first VI that will be called when the * .exe file is run.
    - ➢ Go to the *Icon* category. Here you can create a custom icon for the application or leave the default one.
    - ➢ Go to the *Version Information* category. Here you can manually change the application version. You can also leave *Auto-increment version on build* option enabled.

- Click the *Build* button and wait for the application build to complete. Click the *Explore* button to open the location in which the executable version of the application was created.
- Run the application executable and test it.

3. Create an installer package.
    - In the project tree, add a new installer specification by selecting *New* → *Installer* option on the *Build Specifications* item. The installer configuration window should appear.
    - On the *Product Information* category, change the build specification and product name. Define the *Installer destination* path.
    - Go to the *Source Files* category. Click on the *Speech Recorder* build and move it to the *Destination View* by clicking the arrow button. This operation specifies for which executable the installer should be created.
    - Click the *Build* button and wait for the installer package to be created.
    - Optionally, run the installer once it is created and test the installation process.

END OF EXERCISE

PULSE OXIMETER 1 Project Requirements

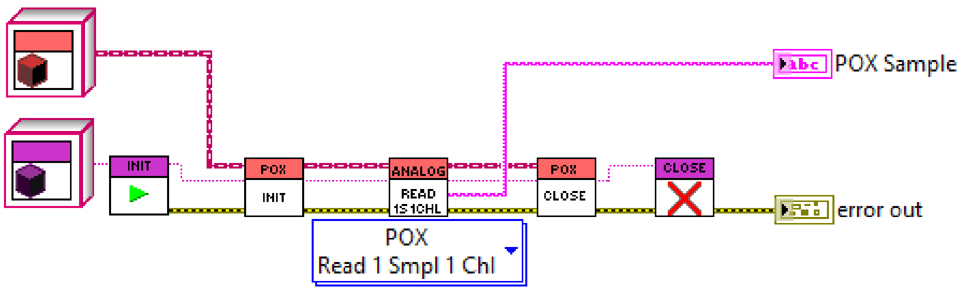
GOAL

Read the *Pulse Oximeter – Project Specification* document carefully. Check the hardware requirements with the devices provided. Plan the architecture and development of the application.

IMPLEMENTATION

1. Go to the *...\GPL2\Exercises\ Pulse Oximeter\Step 1* location in course materials directory. Open the project specification and read it. Try to answer following questions:
   - Is the specification clear?
   - Do you have access to all hardware described in the specification?
   - Do you have access to the *Magic Box* palette used for communication with devices?
   - How many threads should the application have?
   - Which method of the parallel threads communication would be appropriate?
   - What design pattern should you use?
   - How to handle the mechanism of asynchronous display of measurement files in separate windows?
2. Test the hardware.
   - Place the Base Shield connector on the Arduino Uno device.
   - Connect the POX device to the any I2C input on the board.
   - Use the USB cable to connect the Arduino to the PC. The LED on the Arduino should light up.
   - Go to the NI MAX application and check if the new device appears in the *Devices and Interfaces* section.
3. Prototype the POX measurement.
   - Create a new VI.
   - On the BD, open the functions palette, find the *Magic Box* palette and pin it.
   - From this palette, select *Arduino* → *Init* and *Arduino* → *Close* functions.

- Similarly, select *Init*, *Read* and *Close* functions from the *POX* subpalette.
- Connect all five VIs with the error wire so they execute in following order: *Arduino Init*, *POX Init*, *POX Read*, *POX Close*, *Arduino Close*.
- Create a constant for the *Arduino in* input of the *Arduino Init* function. Connect the *Arduino Out* output to the appropriate input of the next function.
- Create a constant for the *POX in* input of the *POX Init* function.
- Note that both constants created in the previous steps are classes, which are not discussed in this course. For the purposes of the project, classes can be treated as protected clusters - we cannot modify or read their elements, but we can pass them between functions.
- Configure all functions inputs so the single measurement can be acquired. Display the value outputted from the *POX Read*.



- Switch to the FP and run the VI. The device initialization process may take several seconds.
- Check that there is no error and whether a POX sample has been read. Most likely, the *POX Sample* will stand *Invalid Sample*. This means that the measurement was successful, only no heartbeat was detected.

END OF EXERCISE

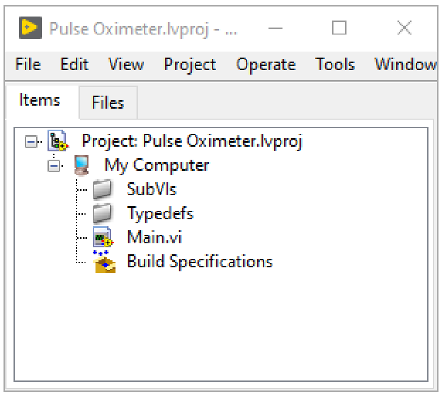
PULSE OXIMETER 2 Application Architecture

GOAL

Prepare the project structure and application architecture, including all anticipated threads and communication methods. Prepare the application GUI. Implement an application error handling and shutdown mechanism. The implementation of data acquisition, data display, and data storage logic is not the focus of this exercise.

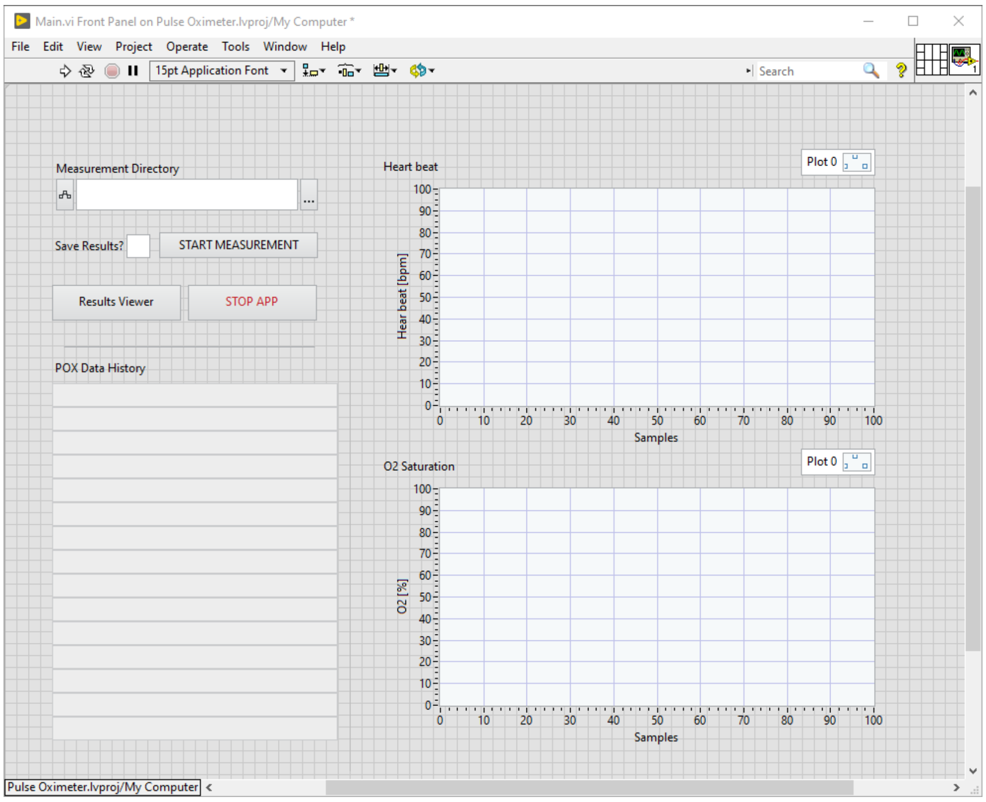
IMPLEMENTATION

1. Prepare a project tree.
   - Create a new project and save it. Create an empty top-level VI.
   - Prepare virtual folders for subVIs and typedefs. Create related folders on disk as well.

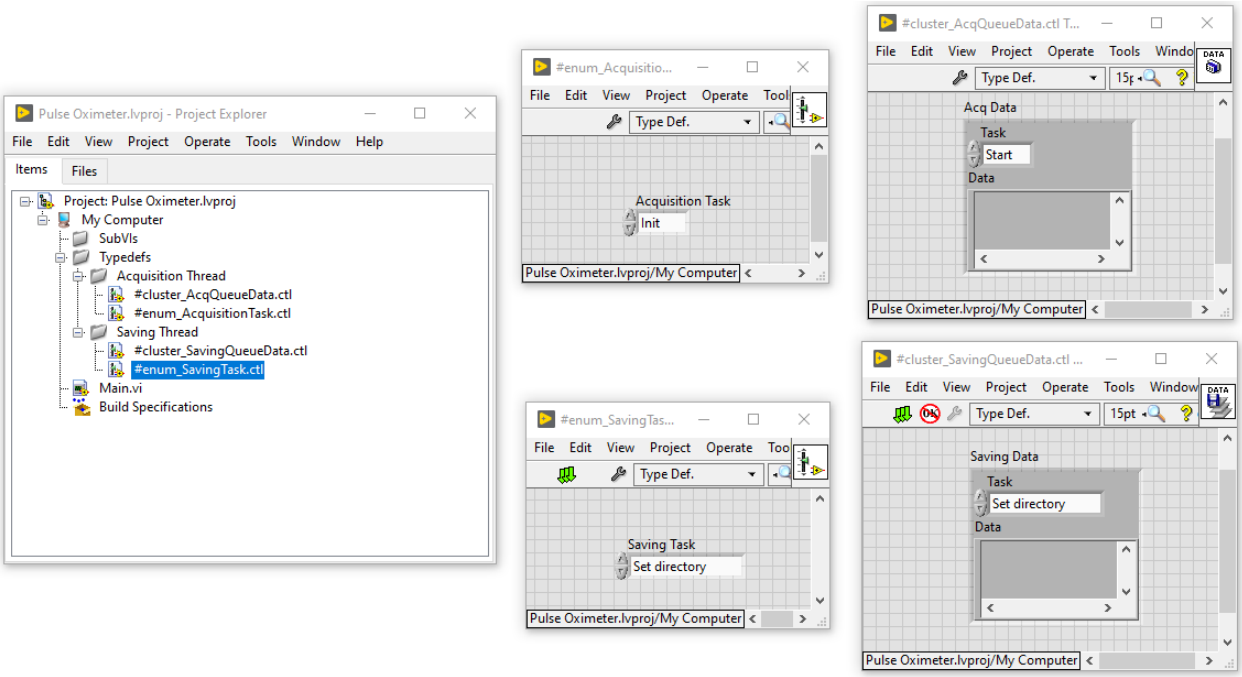

Pulse Oximeter.lvproj - ...
File Edit View Project Operate Tools Window
Items Files

```
Project: Pulse Oximeter.lvproj
    My Computer
        SubVIs
        Typedefs
        Main.vi
        Build Specifications
```

2. Prepare the GUI.
   - Based on the specification, create all necessary controls and indicators:
     i. Path control for *Measurement Directory*,
     ii. Buttons for *Start Measurement*, *Stop App* and *Result Viewer*,
     iii. Checkbox for *Save Results?*,
     iv. Waveform charts for *Heart beat* and *O2 Saturation*,
     v. String 1D-array for *POX Data History*.



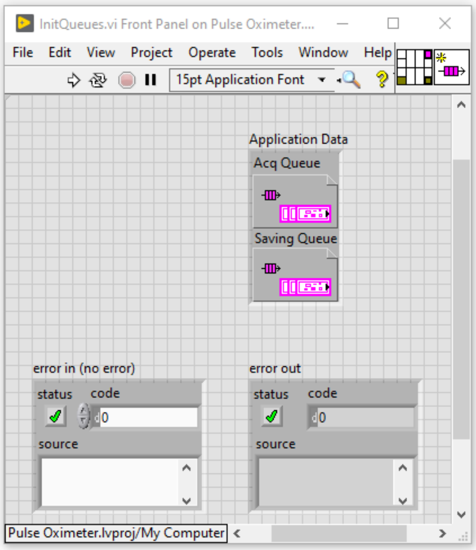
3. Prepare the threads.
   - The application should have three threads: Master Thread, Acquisition Thread, and Saving Thread. Moreover, it should be based on the Queued Message Handler (QMH) design pattern.
   - Two queues should be used for communication between threads: the first queue for the Acquisition Thread, and the second queue for the Saving Thread.
   - Prepare type definitions used for communication:
     i. Prepare an enum for Acquisition Thread tasks (items: *Init*, *Start*, *Finish*, *Close*),
     ii. Prepare an enum for Saving Thread tasks (items: *Set directory*, *Save*, *Stop*, *Close*),
     iii. Prepare two clusters to be used by the queues. Each cluster should contain the appropriate enum and a variant.

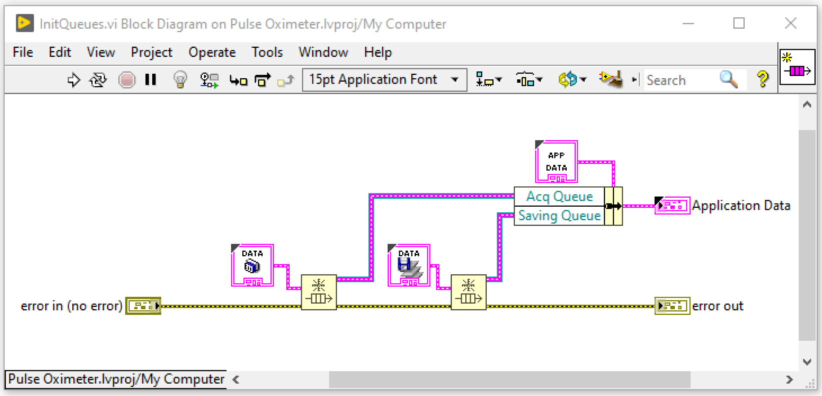


➢ Initialize the queues:

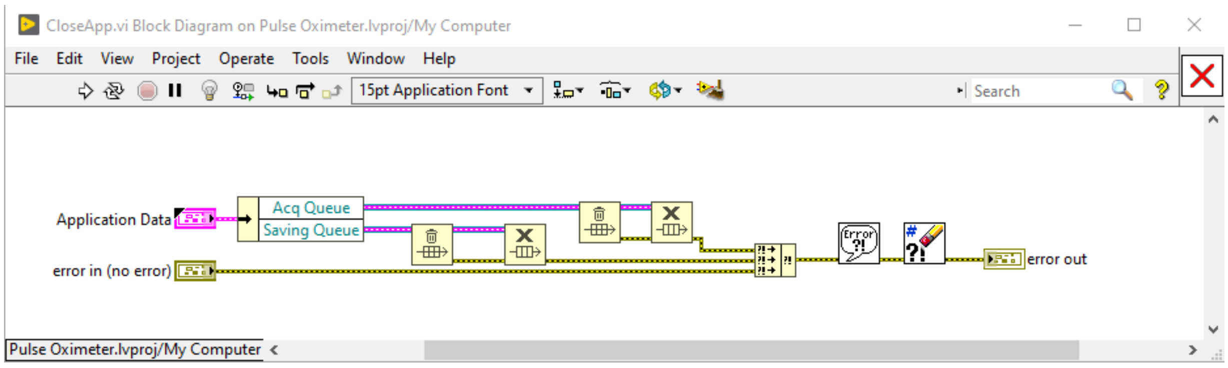
i. Prepare *InitializeQueues.vi* to obtain both queues using the proper typedefs,

ii. Prepare *Application Data* cluster that holds references for both queues and configure it in the *InitializeQueues.vi*.

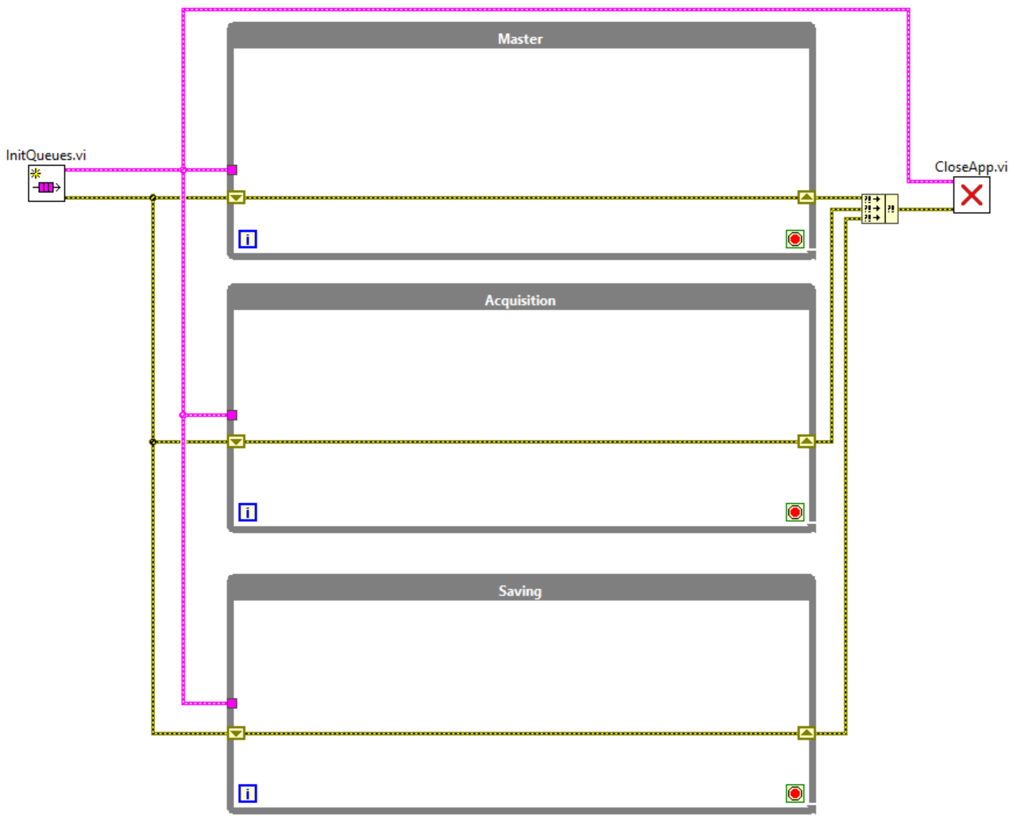



InitQueues.vi Block Diagram on Pulse Oximeter.lvproj/My Computer
APP DATA
DATA
Acq Queue
Saving Queue
Application Data
error in (no error)
error out
Pulse Oximeter.lvproj/My Computer

➢ Prepare *CloseApp.vi* that releases the queues.


CloseApp.vi Block Diagram on Pulse Oximeter.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Application Data
error in (no error)
Acq Queue
Saving Queue
error out
Pulse Oximeter.lvproj/My Computer

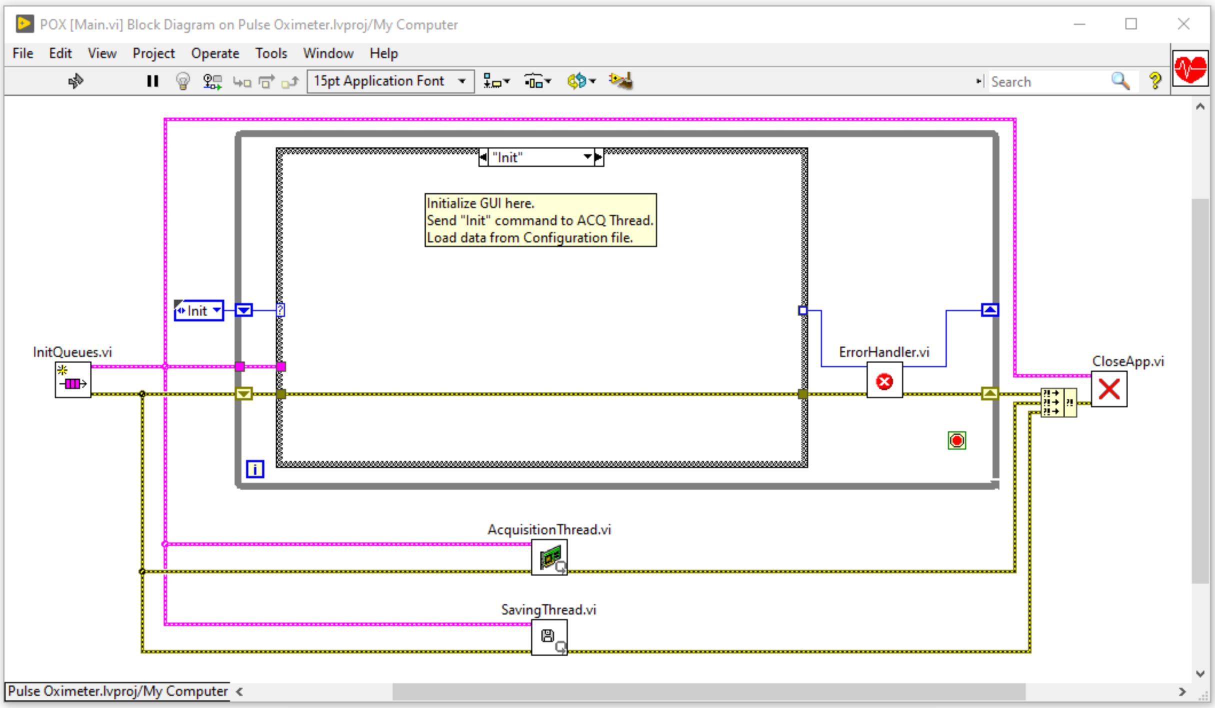
- Prepare three while loops in *Main.vi* and place the prepared VIs. Pass both application data and error wires to all threads.



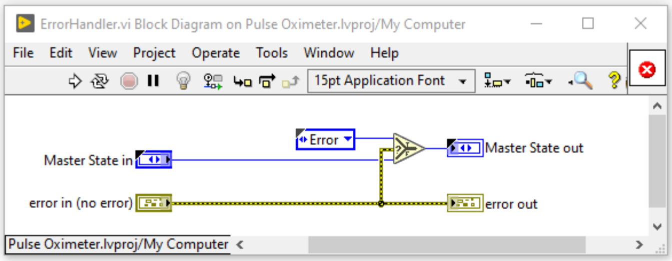
- Convert the Acquisition and Saving threads into subVIs: *Acquisition Thread.vi* and *Saving Thread.vi*.

4. Prepare the Master Thread.
   - Prepare an enum for Master Thread states (items: *Idle*, *Init*, *Close*, *Error*).

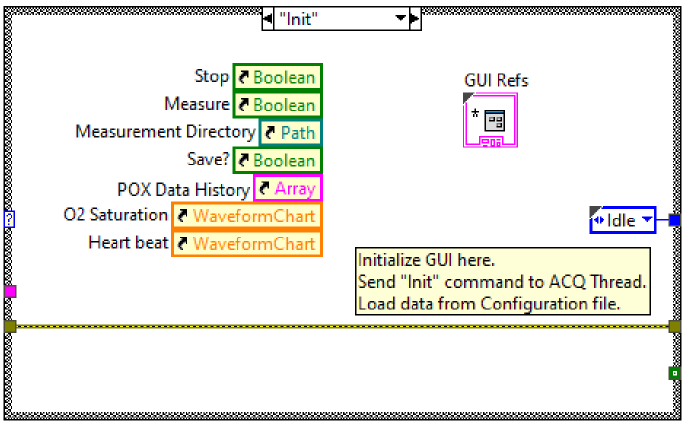
- Prepare a state machine with empty cases for all defined states inside the Master Thread and initialize it with the *Init* state. Inside each case, you can write a short comment describing what should be implemented there.
- Prepare *ErrorHandler.vi* that should be executed right after the CASE. Inside the *ErrorHandler.vi*, toggle the next requested state to *Error*, if an error has occurred.


POX [Main.vi] Block Diagram on Pulse Oximeter.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
"Init"
Init
InitQueues.vi
ErrorHandler.vi
CloseApp.vi
AcquisitionThread.vi
SavingThread.vi
Pulse Oximeter.lvproj/My Computer

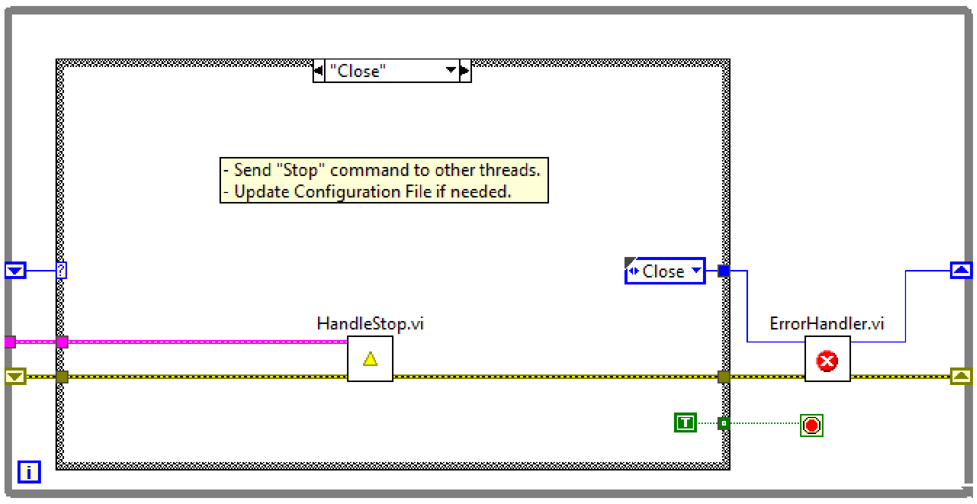
Initialize GUI here.
Send "Init" command to ACQ Thread.
Load data from Configuration file.


ErrorHandler.vi Block Diagram on Pulse Oximeter.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Error
Master State in
Master State out
error in (no error)
error out
Pulse Oximeter.lvproj/My Computer

- Go to the *Init* case. Place here references for all controls and indicators. Prepare a cluster (*GUI Refs*) that will contain all of these references. Go *Idle* as the next state.



- ➢ Prepare the application closing mechanism. In the *Idle* state, place an event structure and add an event for the Stop button. If this button is clicked, pass *Close* as the next state.
- ➢ Inside the *Close* state, stop the Master Thread. Additionally, send *Close* messages to the other two threads (this feature will be implemented later).


"Close"
- Send "Stop" command to other threads.
- Update Configuration File if needed.
Close
HandleStop.vi
ErrorHandler.vi

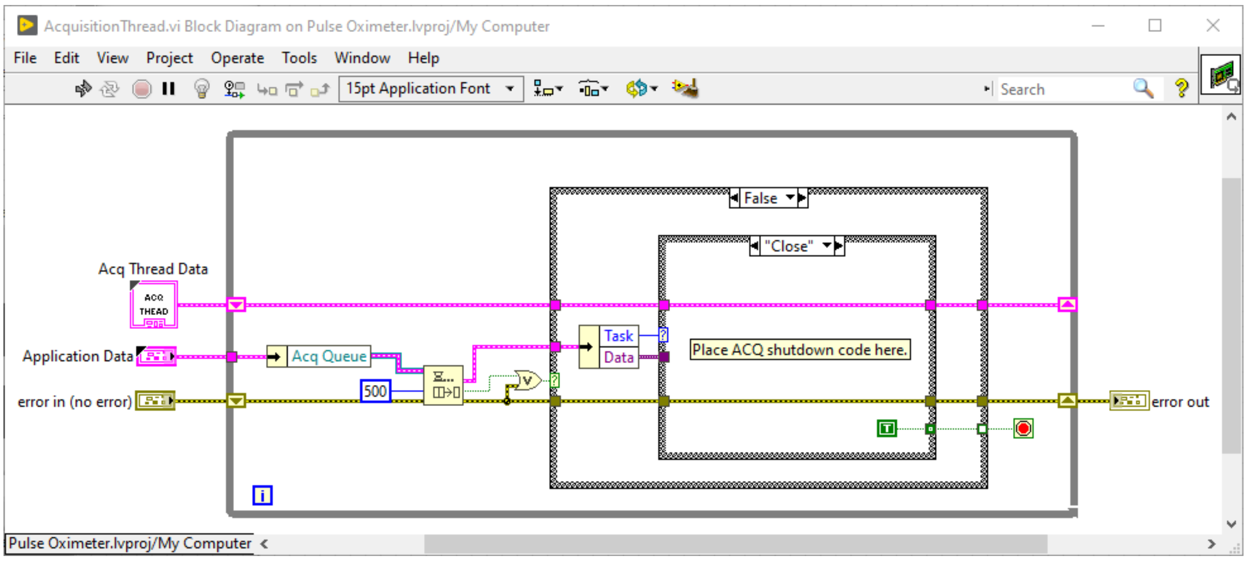
- ➢ Inside *Error* case, display the details of the error, and then clean it. Pass *Close* as the next state.

5. Prepare the Acquisition Thread.
   - ➢ Prepare an enum for this thread state (items: *Idle*, *Operate*, *Error*).
   - ➢ Prepare an *Acq Thread Data* cluster that will contain all data necessary for this thread operation:
     - i. Acquisition Thread state enum,
     - ii. Arduino and POX class instances,

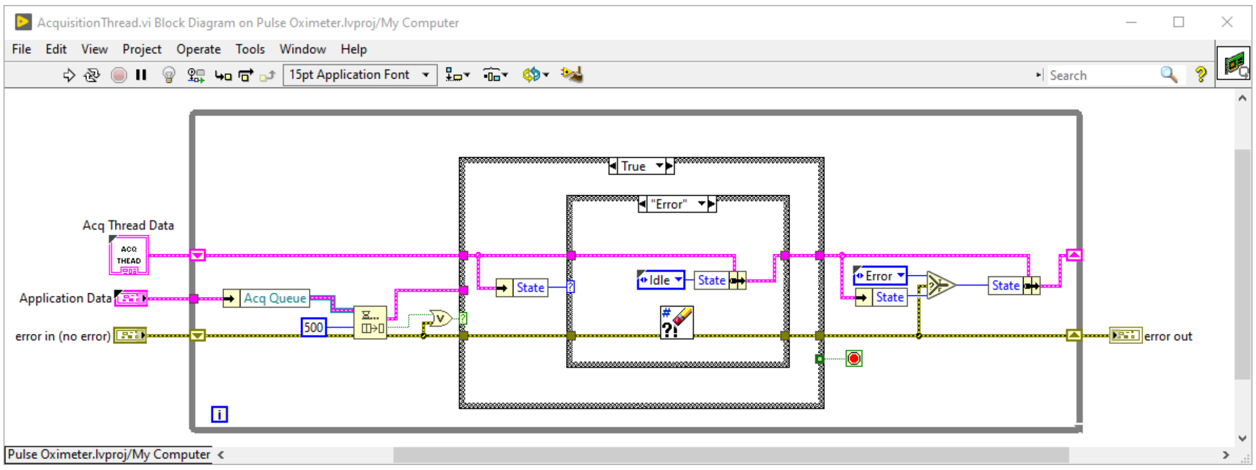
iii. Boolean for *Save?* flag,

iv. *GUI Refs* cluster (created during the *Init* state preparation of the Master Thread).

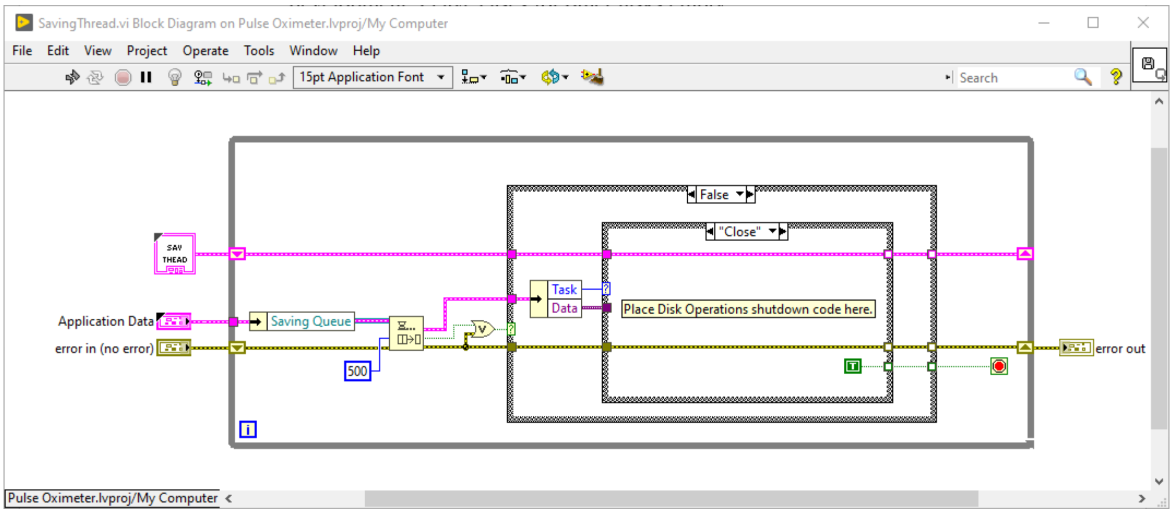
- Go to the Acquisition Thread loop and add a shift register for *Acq Thread Data* cluster.
- Inside the loop, read an element from appropriate queue – use *Dequeue Element* function. Add *Case Structure* controlled by a timeout or error outputted from this function.
- If no error or timeout occurred, add another *Case Structure* to handle the task received from the queue.
- If a *Close* request is received, close the thread. You can add a short comment for further development. Do not configure cases for other tasks.


AcquisitionThread.vi Block Diagram on Pulse Oximeter.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Acq Thread Data
ACQ THEAD
Application Data
error in (no error)
Acq Queue
500
False
"Close"
Task
Data
Place ACQ shutdown code here.
error out
Pulse Oximeter.lvproj/My Computer

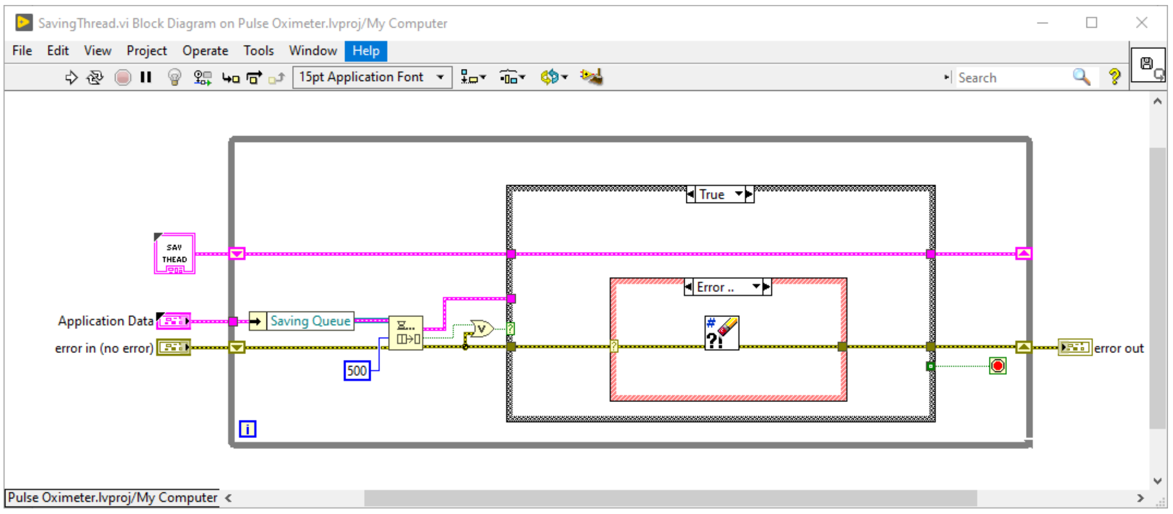
- If an error or timeout occurred, check the current status of thread acquisition (read this information from the *Acq Thread Data* cluster). Add another *Case Structure* here and configure it with state enum.
- Add a case for the *Error* state. Leave other states not implemented for now. Inside the *Error* state, clear the error and pass *Idle* as the next state.
- Outside of both CASEs, implement simple error handling mechanism. If an error has occurred, go to the *Error* state. Otherwise, do not change the requested state.


AcquisitionThread.vi Block Diagram on Pulse Oximeter.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Acq Thread Data
Application Data
error in (no error)
Acq Queue
500
True
"Error"
State
Idle
State
Error
State
State
error out
Pulse Oximeter.lvproj/My Computer

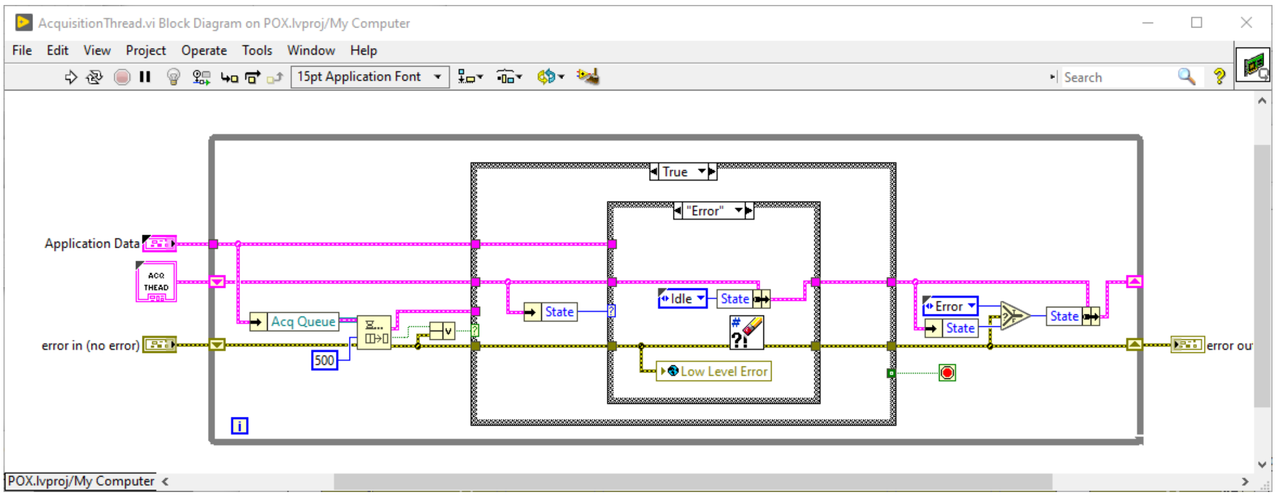
6. Prepare the Saving Thread.
   - ➢ Prepare a *Saving Thread Data* cluster that will contain all data necessary for this thread operation:
     - i. Path control for *Measurement Directory*,
     - ii. Boolean for *File Opened?* flag,
     - iii. Reference for TDMS file.
   - ➢ Go to the Saving Thread loop and add a shift register for *Saving Thread Data* cluster.
   - ➢ Inside the loop, read an element from appropriate queue – use *Dequeue Element* function. Add *Case Structure* controlled by a timeout or error outputted from this function.
   - ➢ If no error or timeout occurred, add another *Case Structure* to handle the task received from the queue.
   - ➢ If a *Close* request is received, close the thread. You can add a short comment for further development. Do not configure cases for other tasks.


SavingThread.vi Block Diagram on Pulse Oximeter.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
False
"Close"
Task
Data
Place Disk Operations shutdown code here.
Application Data
error in (no error)
Saving Queue
500
error out
Pulse Oximeter.lvproj/My Computer

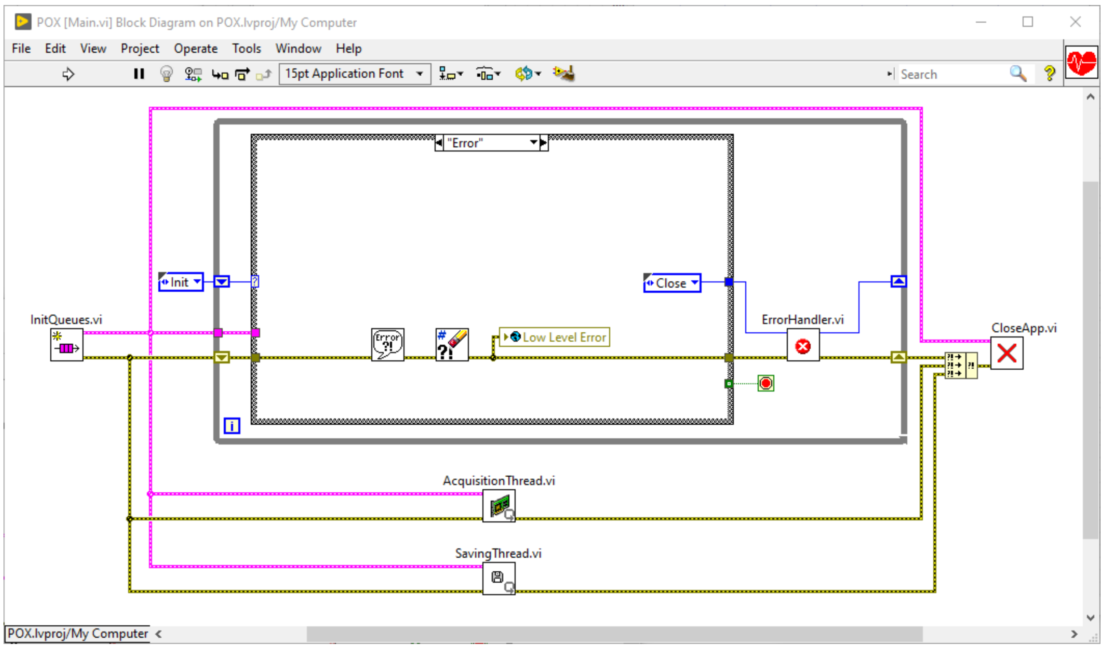
- ➢ If error or timeout is returned by the *Dequeue Element* function, check whether error occurred. Add another *Case Structure* and configure it with the error wire. Now go to the *Error* case and clear the error.


SavingThread.vi Block Diagram on Pulse Oximeter.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
True
Error ..
Application Data
error in (no error)
Saving Queue
500
error out
Pulse Oximeter.lvproj/My Computer

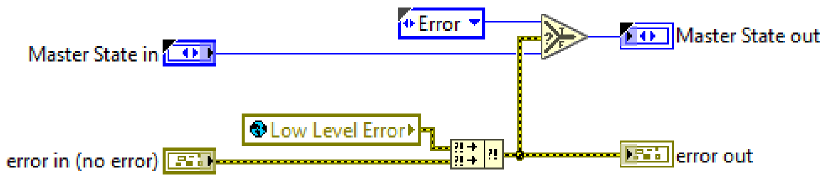
7. Implement global error handling mechanism using a global variable. Each thread should write to this variable if an error occurs. The Master Thread should be responsible for processing this error.
   - ➢ Add a new global variable holding an error cluster.
   - ➢ Inside the *Error* case in the Acquisition Thread, write an error to the global error variable.



- Similarly, update the global error variable in the Saving Thread inside the *Error* case.
- Reset the global error variable by writing *No error* value to it inside the *Error* state in the Master Thread.



- Modify *ErrorHandler.vi*, so that the *Error* state is passed when an error occurs in the Master Thread or in the global error variable (use *Merge Errors* function).



8. Clean-up the project and test the application.
   - ➢ Clean-up the code (you can use subVIs if needed).
   - ➢ Verify that the top-level VI is runnable and troubleshoot any issues. If there are empty cases with unconfigured (hollow) tunnels, provide the necessary values.
   - ➢ Launch the application. At this point, the application should do nothing special, but it should close properly when the *Stop* button is clicked.

END OF EXERCISE

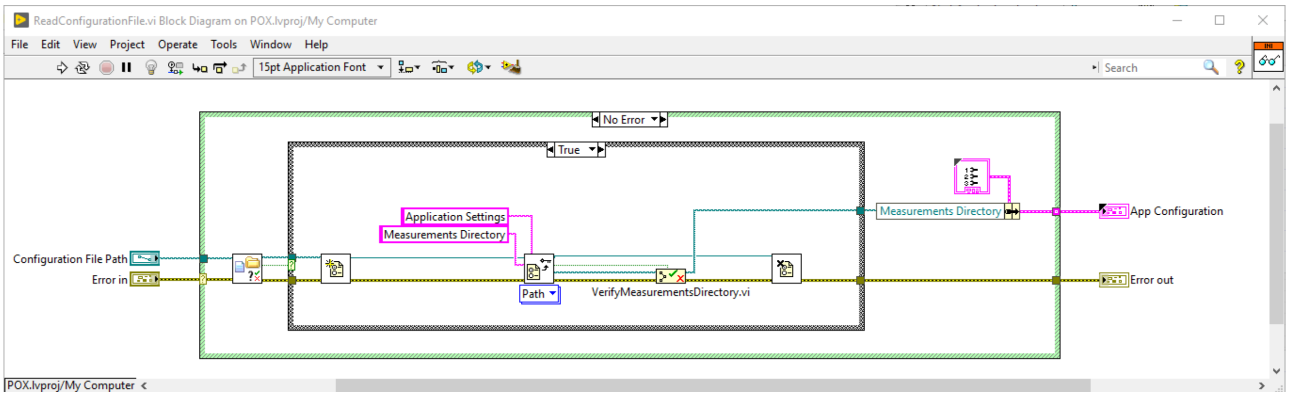
PULSE OXIMETER 3 Implementation, Evaluation, Distribution

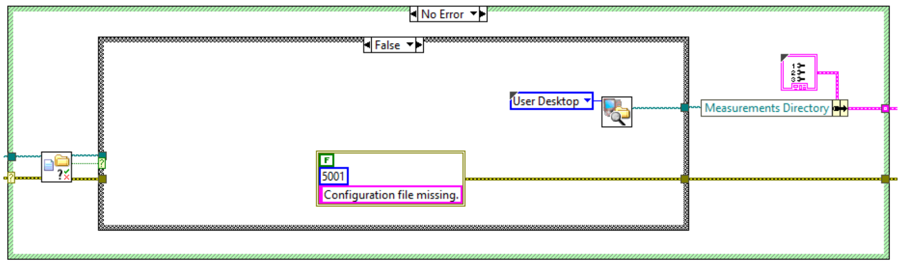
GOAL

Implement all functionalities required by the project specification. Fill in the gaps prepared in the previous step while working on the application architecture.

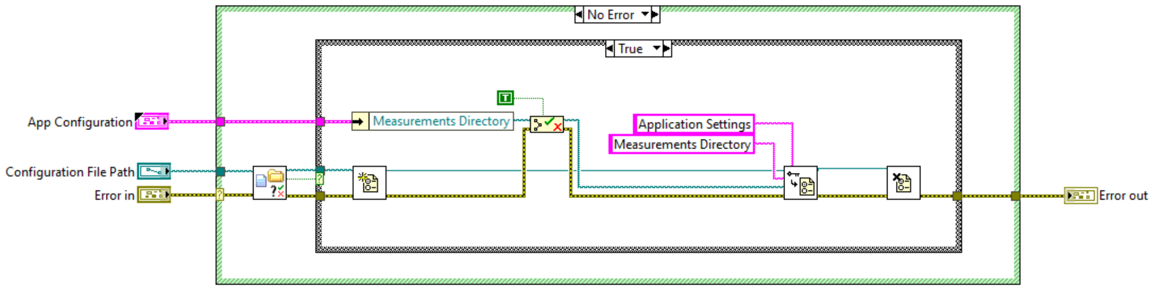
IMPLEMENTATION

1. Implement the configuration file support (use *File I/O → Configuration File VIs* palette).
   - Create a *Configuration.ini* file in the …\data subfolder of the project location. Fill it with the appropriate values.
   - Create a cluster that stores all data read from the configuration file.
   - Create a VI for reading the configuration file. This VI should check if the file exists and all values defined in that file are correct. In case of any detected misbehaviors, proper error handling should be implemented. For example, if the defined measurement path does not exists on disk, the default measurement directory should be set. Another example: an error should be generated if the configuration file is not found.



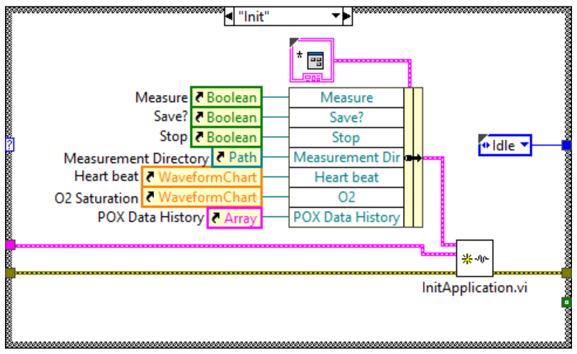


➢ Create a VI for updating the configuration file.



2. Implement the initialization and shutdown of the application.

   ➢ Inside the *Init* state of the Master Thread, build a *GUI Refs* cluster using the prepared references. Create and place *InitApplication.vi* in this case.

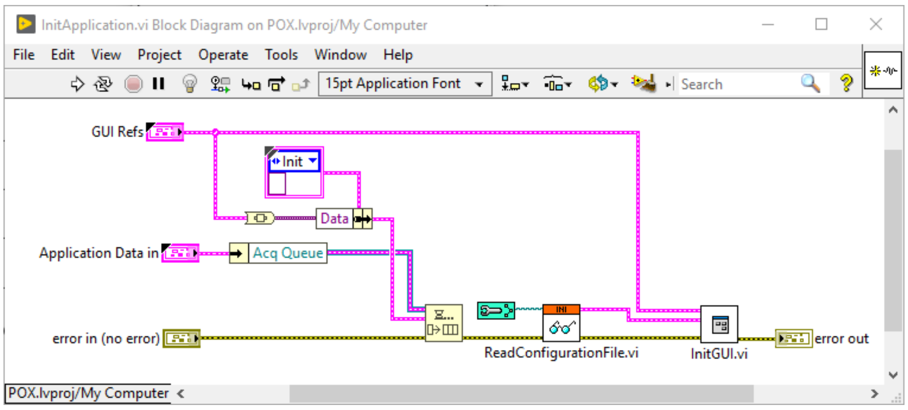


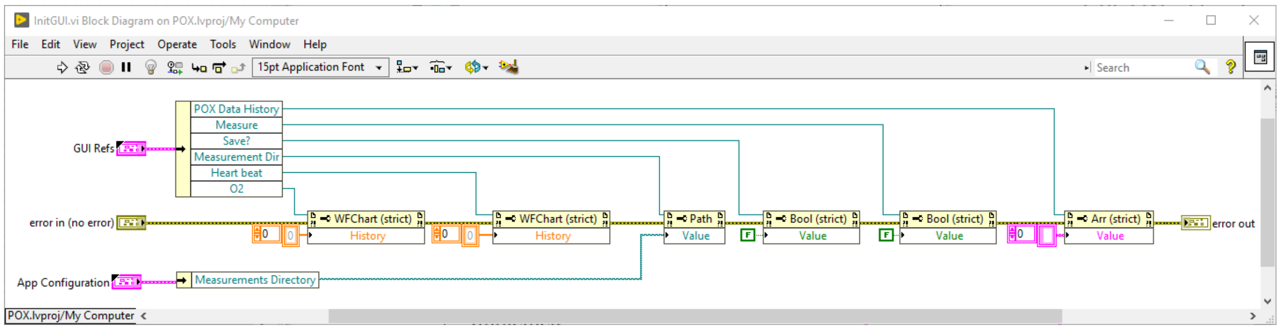
   ➢ Implement *InitiApplication.vi* so it covers the following functionalities:

      i. Reads the configuration file,

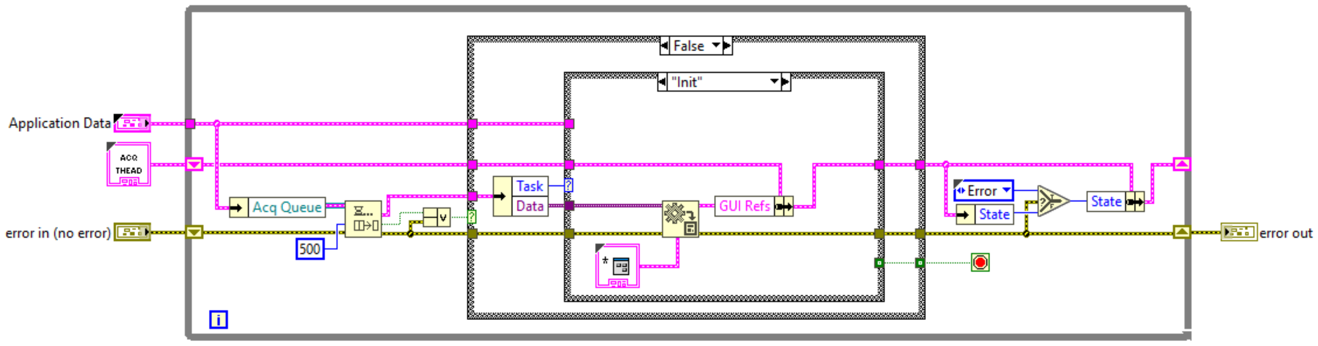
      ii. Initializes all controls and indicators using *GUI Refs* cluster and *Property Nodes* (the *Measurement Directory* control should be initialized with the value read from the *Confiugration.ini* file).

      iii. Sends the *Init* task to the Acquisition Thread (the *GUI Refs* cluster should be sent within the message – it will come in handy later).

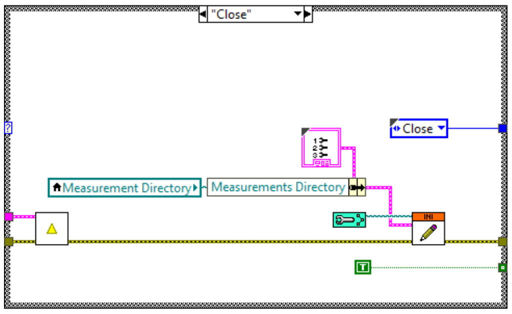




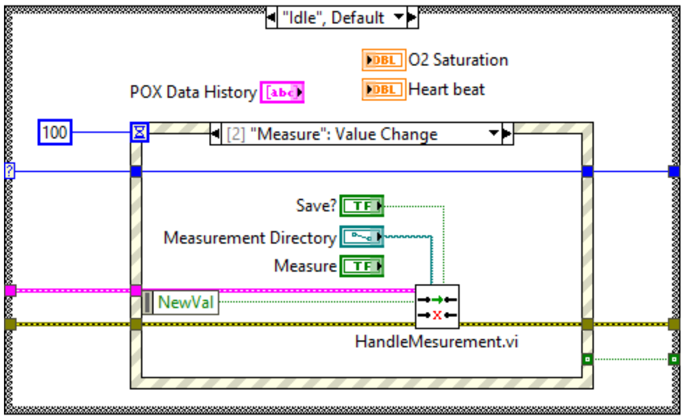
- ➢ Inside the Acquisition Thread, add a case for the *Init* task received from the queue. Update the *GUI Refs* cluster inside the Acquisition Thread data cluster (use *Variant to Data* function). Thanks to this operation, access to interface objects located in another thread was obtained inside the acquisition loop.



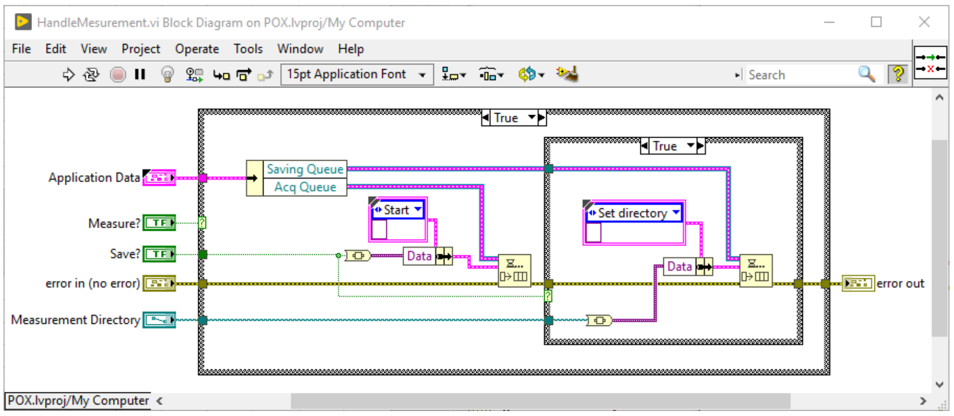
- ➢ Inside the *Close* case of the Master Thread, read currently set Measurement Directory value and update the *Configuration.ini* file with it.

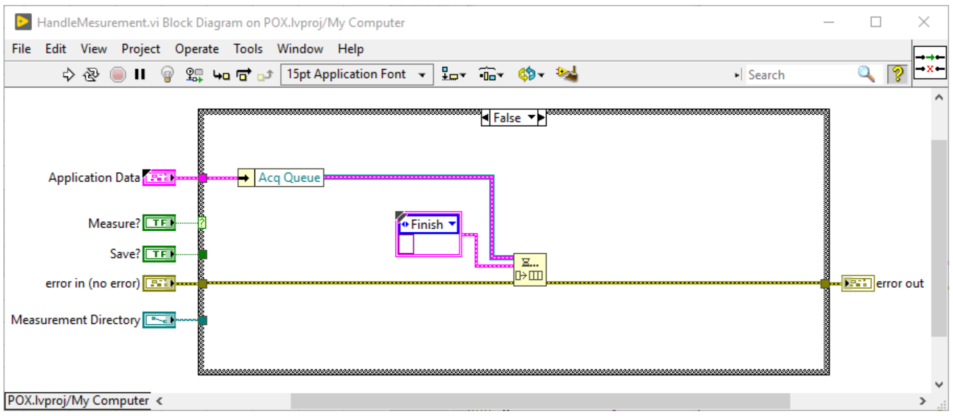


3. Implement data acquisition.
   - Inside the *Idle* state of the Master Thread, add a new event for the button responsible for starting the measurement.
   - Prepare a subVI (*HandleMeasurement.vi*) that handles all the actions performed on this button click. You will need to check whether the button value is *True* or *False*, and pass the measurement configuration.



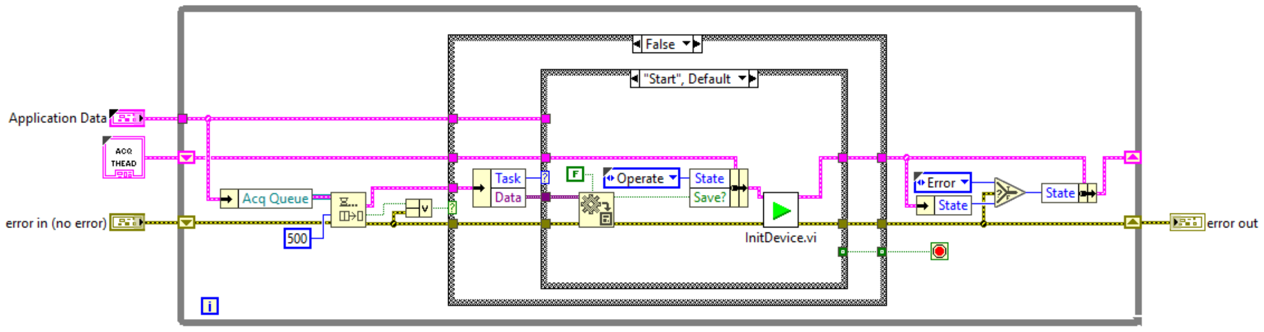
   - Inside the *HandleMeasurement.vi*, send relevant tasks to the Acquisition and Saving threads. If the saving is enabled and the measurement starts, *Set Directory* task should be sent.

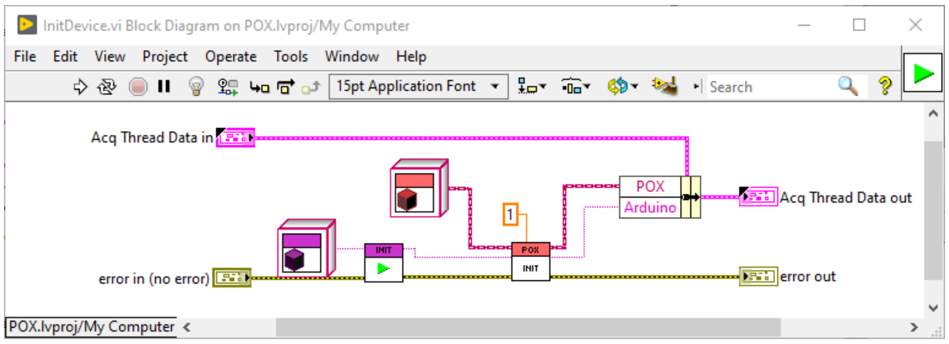




➢ Inside the Acquisition Thread, add a case for handling the *Start* and *Finish* actions.

➢ If the *Start* task is requested, the thread state should be set to *Operate*. Both Arduino and POX devices should be initialized (use the functions from the *Magic Box* palette).



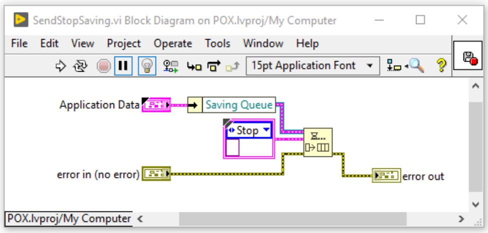


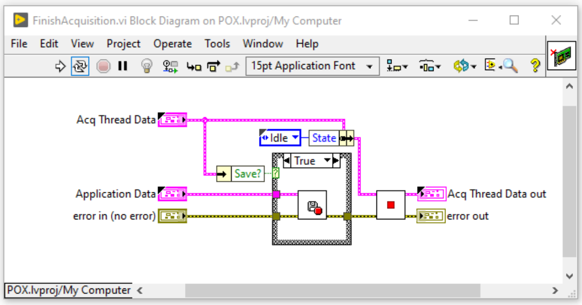
➢ If the *Finish* task is received, the following operations should be performed:

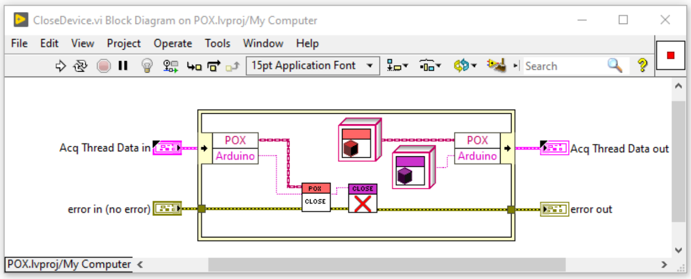
i. Set the thread state to *Idle*,

ii. Close both Arduino and POX devices,

iii. Send the *Stop* request to the Saving Thread if the saving was enabled.

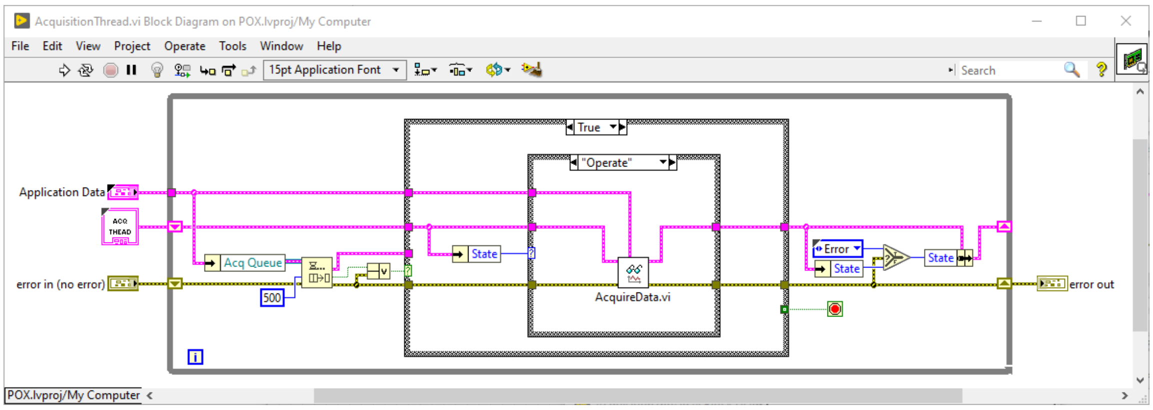

SendStopSaving.vi Block Diagram on POX.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Application Data
Saving Queue
Stop
error in (no error)
error out
POX.lvproj/My Computer





➢ Inside the *Close* task, close all devices using the same function as for the *Finish* task.

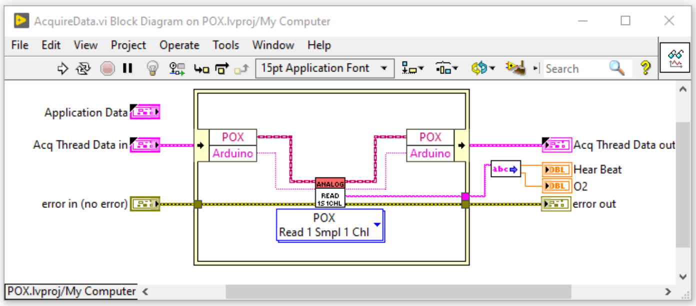
➢ Implement continuous acquisition. Inside the Acquisition Thread, the measurement should be done if there is no other task requested and the thread state is equal to *Operate*.

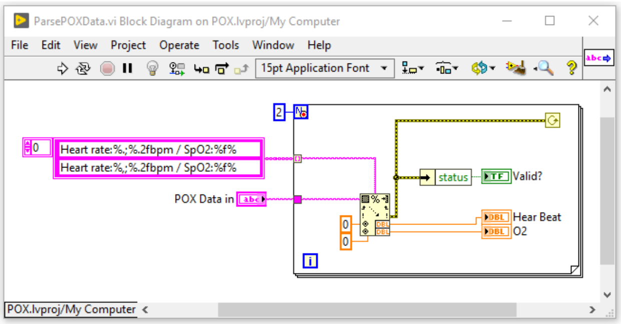


➢ In order to acquire the data, use the function from the *Magic Box* palette. Parse the Heartbeat and O2 Saturation values to numeric. Note that there are two possible string formats to support depending on the development environment:

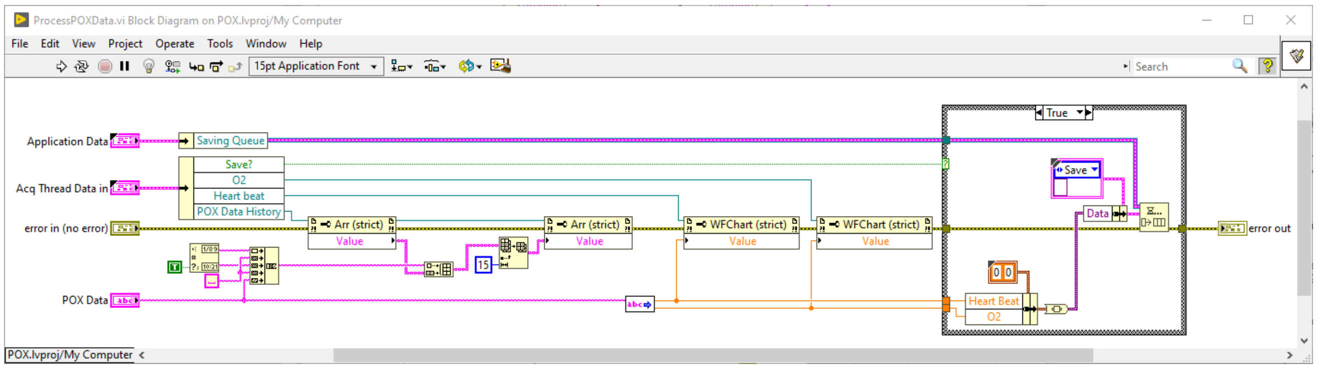
i. Heart rate:%.;%.2fbpm / SpO2:%f%

ii. Heart rate:%,;%.2fbpm / SpO2:%f%




ParsePOXData.vi Block Diagram on POX.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Heart rate:%;%.2fbpm / SpO2:%f%
Heart rate:%;%.2fbpm / SpO2:%f%
POX Data in
status
Valid?
Hear Beat
O2
POX.lvproj/My Computer

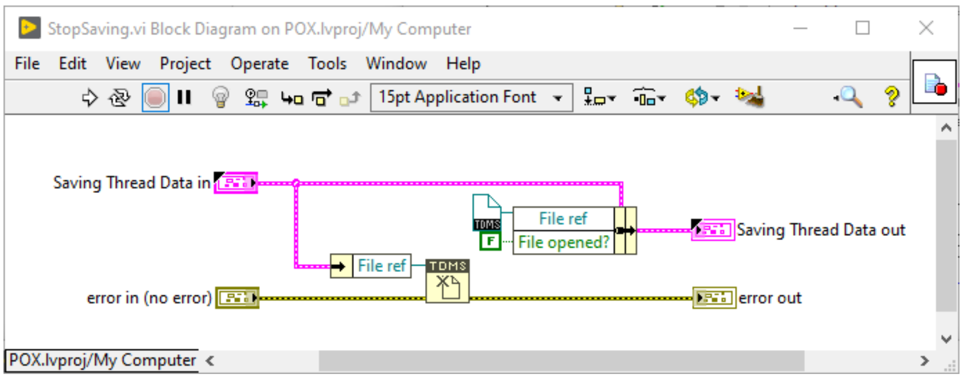
- ➢ Modify the VI for acquiring data so it displays measurement on the GUI using the *GUI Refs* cluster and *Property Nodes*:
  - i. Every data read from the POX should be displayed in the *POX Data History* array with the timestamp of the measurement (use *Concatenate Strings* function),
  - ii. If data is valid, numeric values of Heartbeat and O2 Saturation should be displayed on the graphs.
- ➢ Send the *Save* task to the Saving Thread if the saving is enabled. Within this task, numeric values read from the POX should be provided (you may want to create an appropriate typedef).



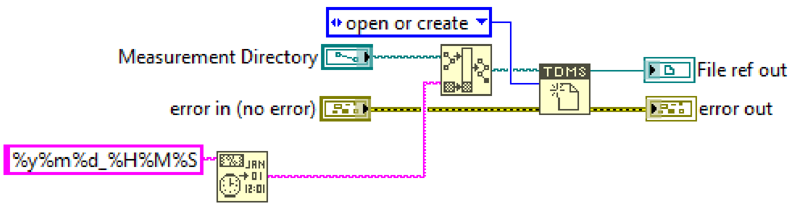
4. Test the current status of the application. Verify the following functionalities:
   - ➢ No measurement is taken if the *Measure* button is not clicked,
   - ➢ The measurement directory control is filled with the correct path (read from the configuration file or default),
   - ➢ After starting the measurement, the *POX Data History* indicator updates itself and displays the latest data on the top of the list,
   - ➢ Correct values are displayed on the graph – if a sample is invalid, zero values are added to the graphs,

- The measurement can be stopped and started once again without closing the application.

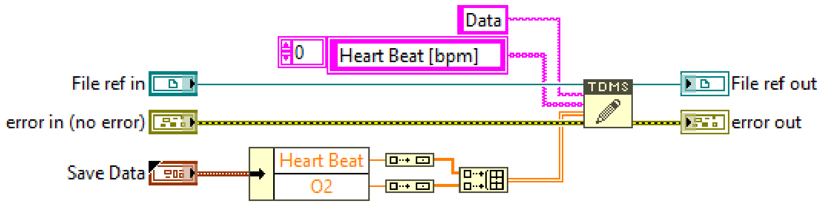
5. Implement data saving.
   - Inside the Saving Thread, add a case for *Set directory* task and update the measurement path in the thread data cluster with the received value.
   - Prepare a VI (*StopSaving.vi*) for closing the TDMS file and reset the values stored in the thread cluster data.

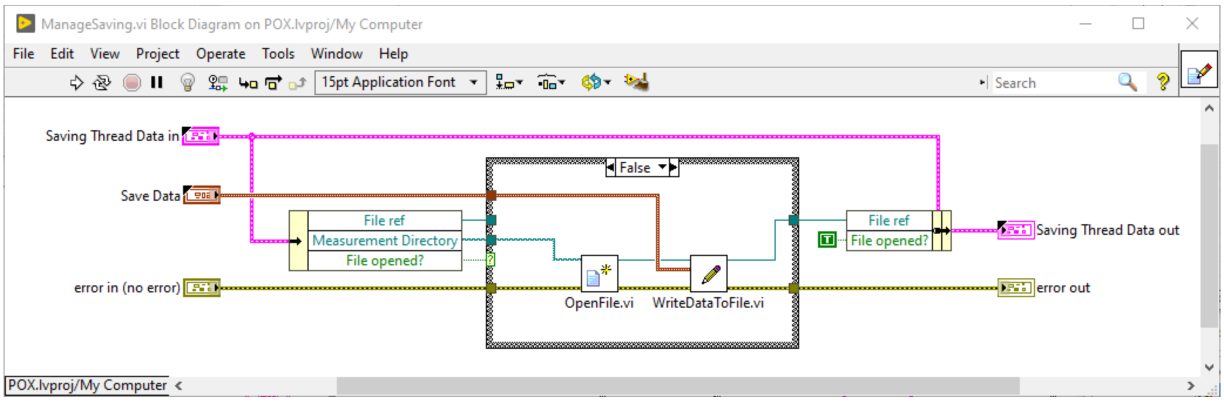


   - Place the *StopSaving.vi* function inside the *Stop* case. Additionally, place this VI inside the *Close* case and clear possible errors right after. There is no need to handle any errors here.
   - Prepare a VI for *Save* task that writes the data to a TDMS file.
      i. If any file is not yet opened, create and open a new file. Name the file basing on the current timestamp.



      ii. Set the *File opened?* Flag to *True*.
      iii. Write the data received from the queue to the TDMS file.

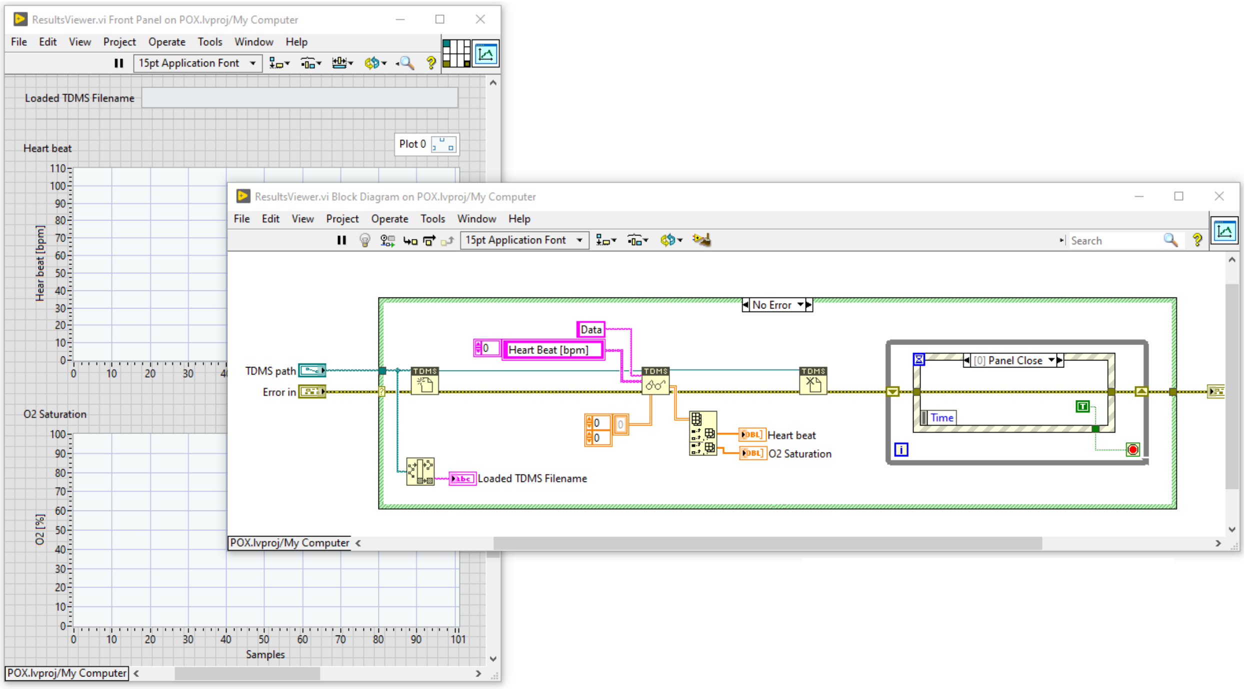




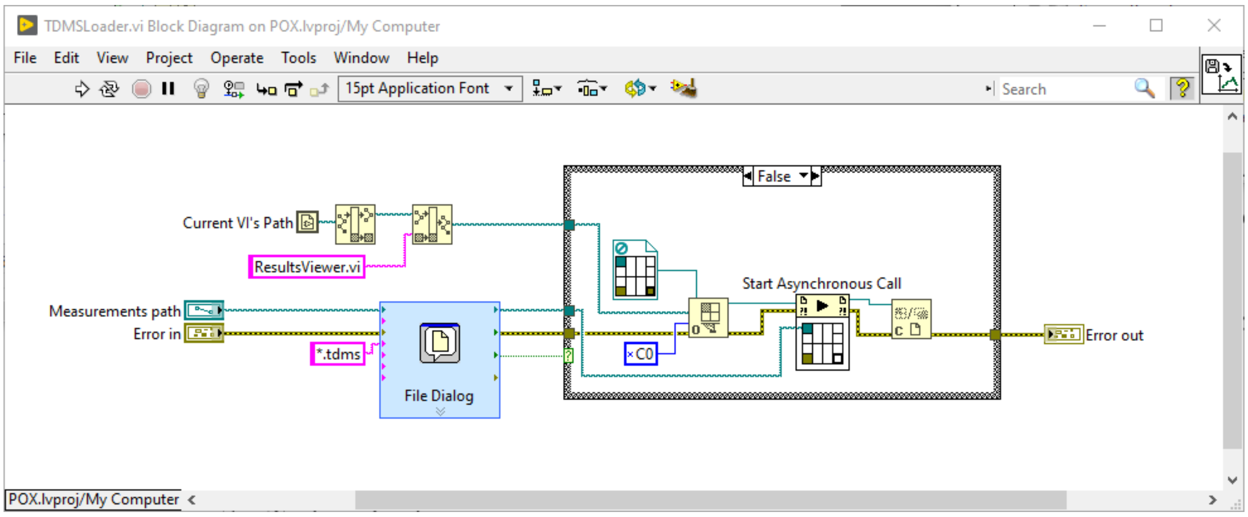
➢ Test the application and check whether the data is correctly streamed to the TDMS file.

6. Implement a result viewer window (*ResultViewer.vi*).

➢ Create a result viewer window that reads the selected file and displays values on the graphs. The window should close when the native X-button is clicked.

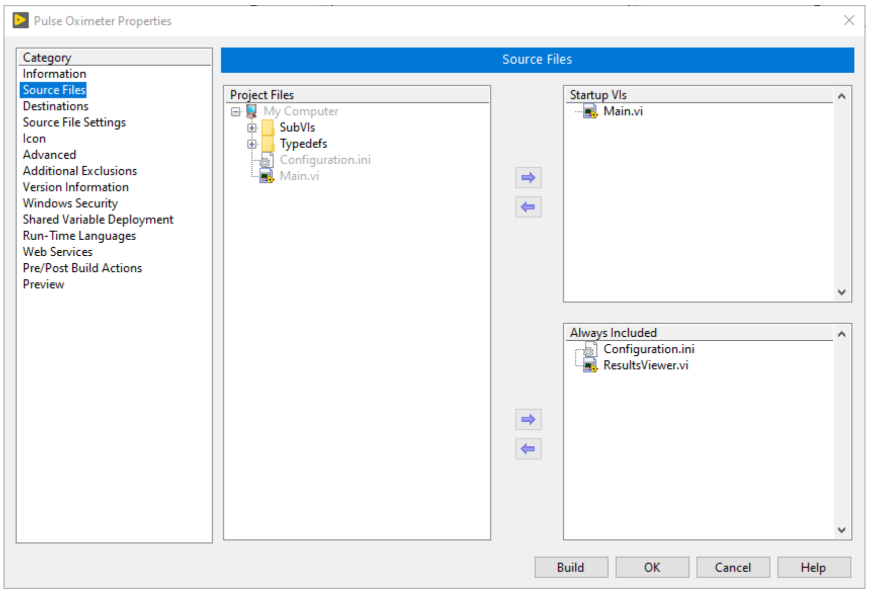


- Prepare a VI that launches a result viewer window.
    i. Use *File Dialog* function to ask the user for a TDMS file to read.
    ii. Use *Open VI Reference* function in order to open a reference to the *ResultViewer.vi* window. Configure it by defining a relative path to the VI.
    iii. Set the *type specifier VI Refnum* input of this function (create a constant and select *Select VI Server Class → Strictly Typed VIs → Interface 0* option from its the context menu). Create a constant for the *options* input and set its value to C0 (refer to LabVIEW Help for more info).
    iv. Use *Start Asynchronous Call* function to call the VI and then close the reference.



- Inside the Master Thread and the *Idle* state, add an event case for the *Result Viewer* button. Use the launcher VI prepared in the previous step to launch the viewer.

7. Test all functionalities requested by the project specification. Check the application behavior if the hardware is not connected to the PC. Fix all detected issues.
8. Create an executable.
    - Create a new build definition for the application. Set correct build names and paths.
    - Configure the *Source Files* section so that *Main.vi* is selected as the startup VI. Note that the *ResultViewer.vi* is launched as asynchronous VI, so it should be always included in the build.



- ➢ Configure other sections of the build (if needed) and create an executable.
- ➢ Test the application launched from created *.exe file.

END OF EXECRISE