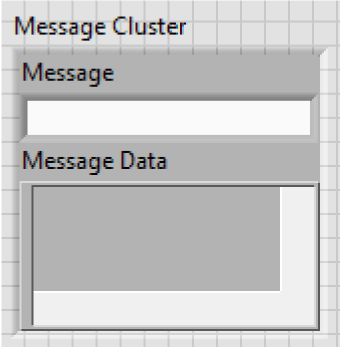
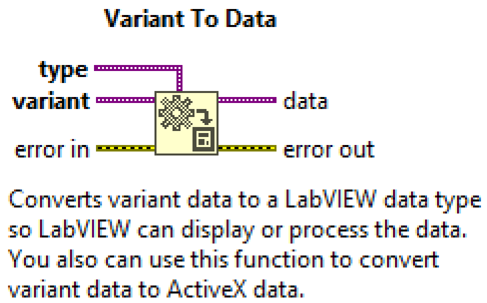
Queued Message Handler

Queued Message Handler

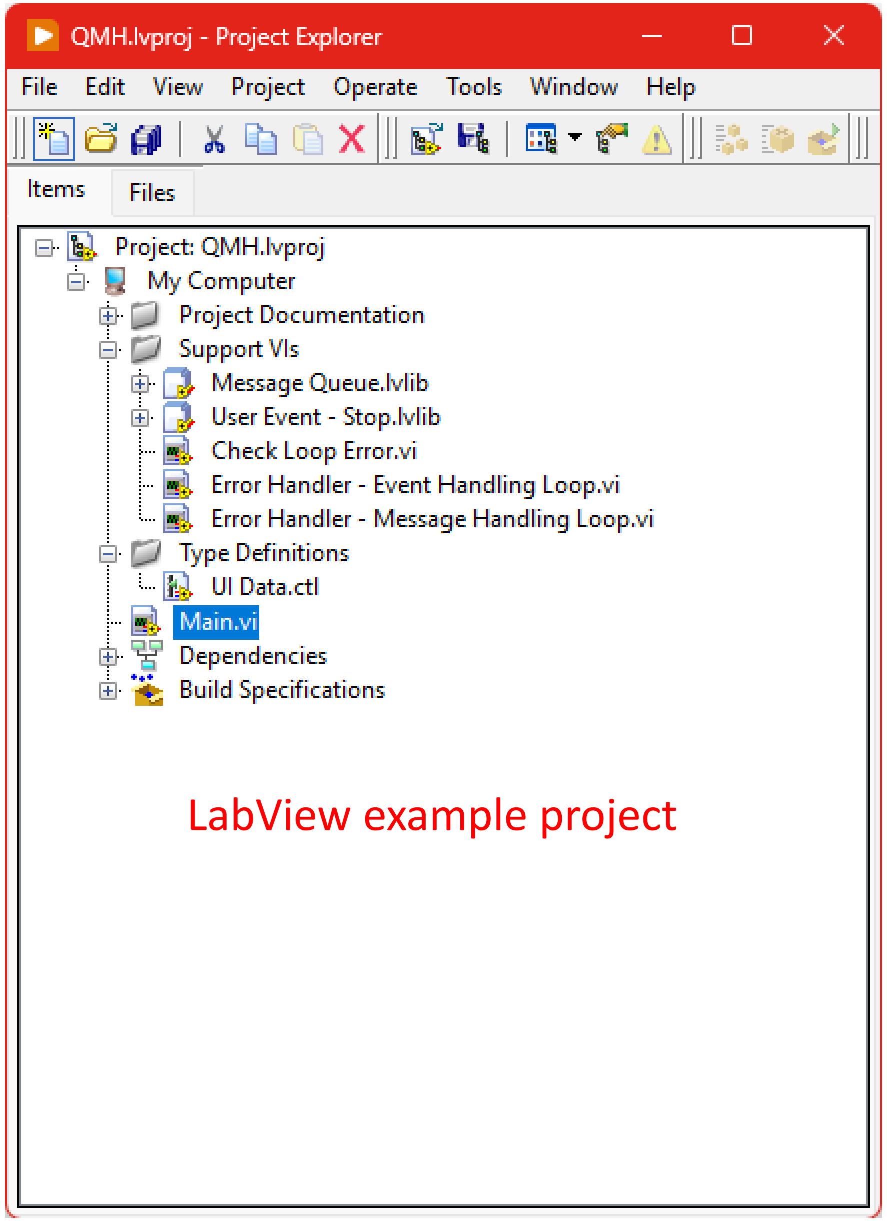
Messages are a cluster of a *string* (‘Message’) and a *variant* (‘Message Data’). *Variant* accepts any type which can be later cast into a proper type using *Variant To Data*.




Variant To Data

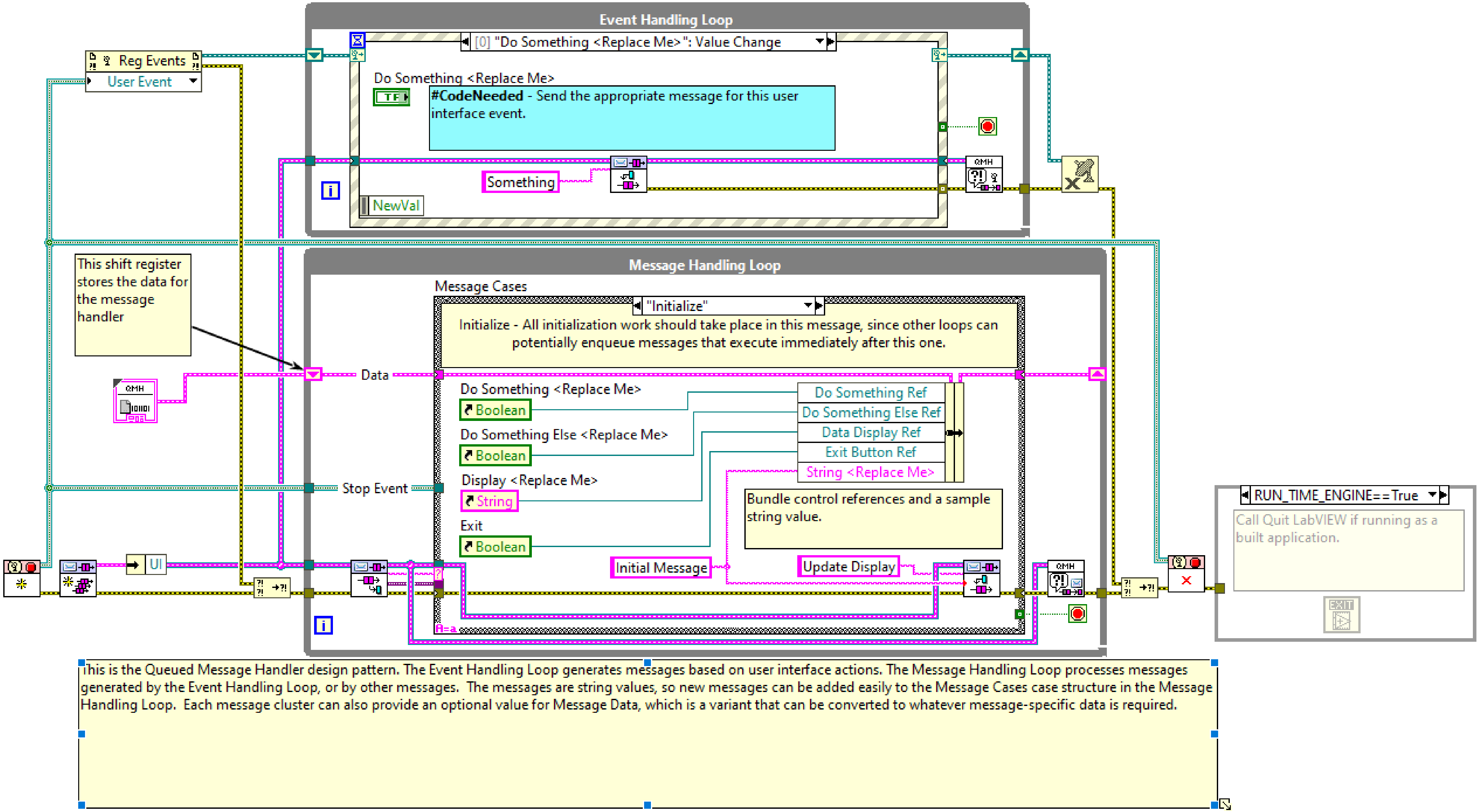
Converts variant data to a LabVIEW data type so LabVIEW can display or process the data. You also can use this function to convert variant data to ActiveX data.

The program works through two loops in the *Producer/Consumer* scheme. The *Event Handling Loop* queues actions (*Messages*), which are executed by the *Message Handling Loop.*



- Project: QMH.lvproj
  - My Computer
    - Project Documentation
    - Support VIs
      - Message Queue.lvlib
      - User Event - Stop.lvlib
      - Check Loop Error.vi
      - Error Handler - Event Handling Loop.vi
      - Error Handler - Message Handling Loop.vi
    - Type Definitions
      - UI Data.ctl
    - Main.vi
    - Dependencies
    - Build Specifications

LabView example project



#CodeNeeded - Send the appropriate message for this user interface event.

This shift register stores the data for the message handler

Initialize - All initialization work should take place in this message, since other loops can potentially enqueue messages that execute immediately after this one.

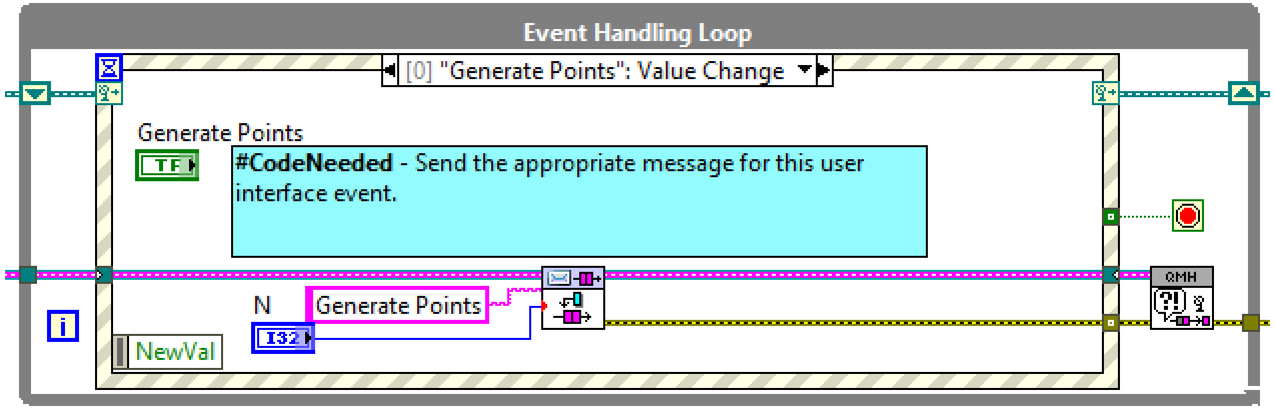
Bundle control references and a sample string value.

Call Quit LabVIEW if running as a built application.

This is the Queued Message Handler design pattern. The Event Handling Loop generates messages based on user interface actions. The Message Handling Loop processes messages generated by the Event Handling Loop, or by other messages. The messages are string values, so new messages can be added easily to the Message Cases case structure in the Message Handling Loop. Each message cluster can also provide an optional value for Message Data, which is a variant that can be converted to whatever message-specific data is required.

Adding new behaviour

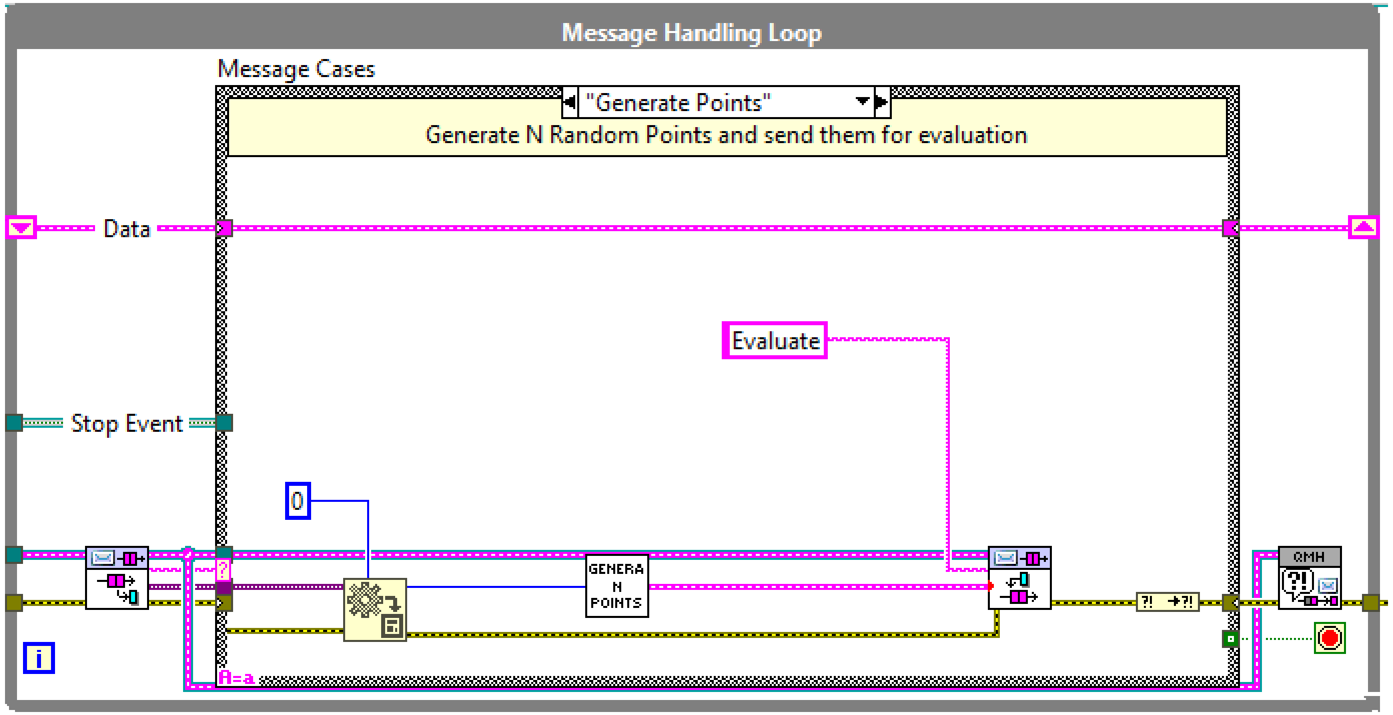
In the *Event Handling Loop a*dd a new *Event* in the
Event Structure (for example a button pressed)
Enqueue a message with needed data



#CodeNeeded - Send the appropriate message for this user interface event.

Adding new behaviour

In the *Message Handling Loop* add a new case to the *Case Structure* (the case needs to have the same name as the message provided before)
Inside add code that should be executed for the given case.
You can enqueue next message to make a sort of *State Machine*

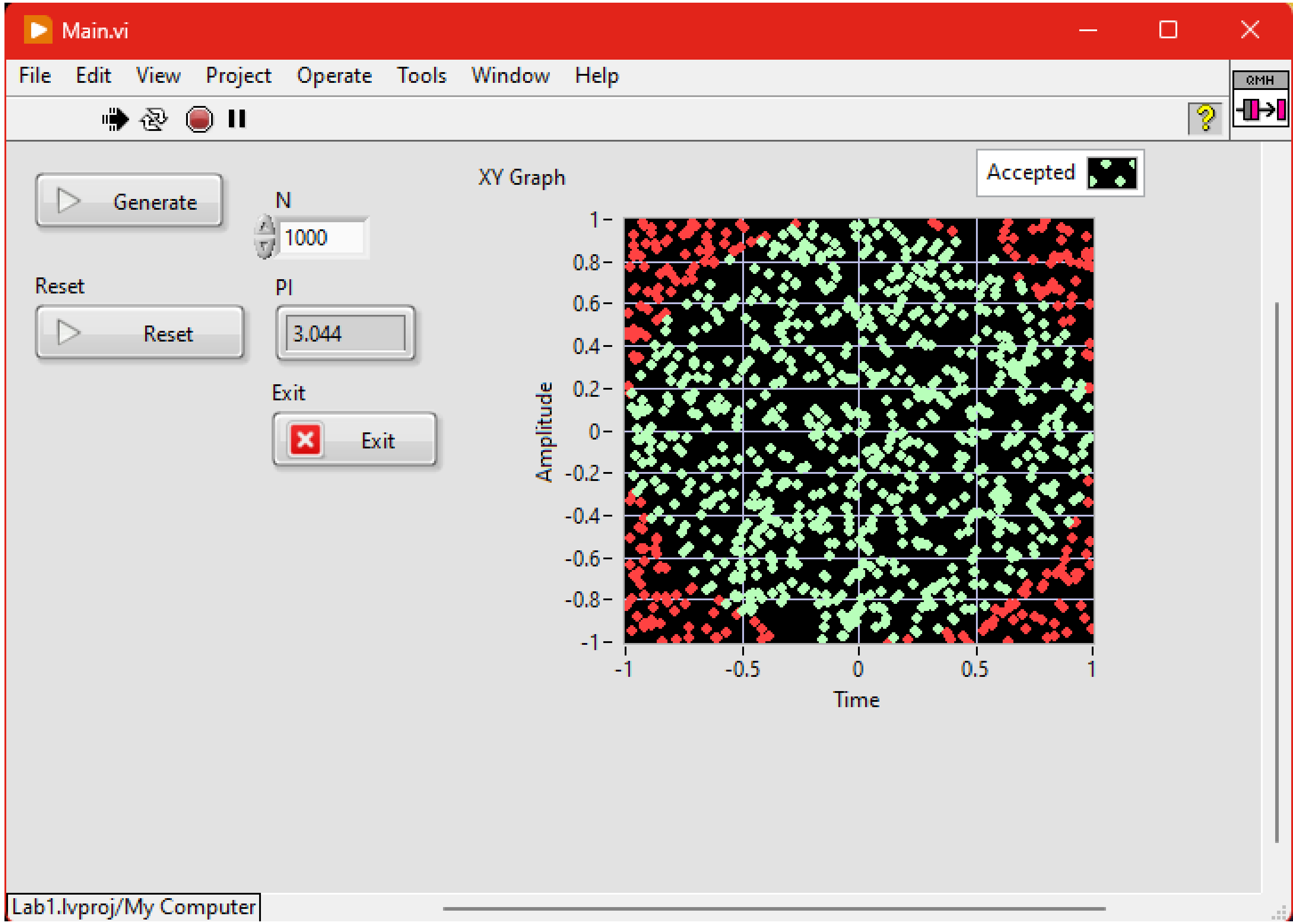


The task

Edit the example project so that you can generate N new points and then calculate the PI using the points.

Program should:

- Have a button that generates N pairs *(x,y)*. N is provided by the user (control)
- Generated points should be divided into two groups: accepted if $x^2 + y^2 < 1$ or rejected otherwise. Only separated points should be saved in the memory
- To calculate PI we use formula: $$\pi = 4 * \frac{N_{accepted}}{N}$$
- Generation, separation of points, calculation of PI should be three different messages
- You shouldn't change the structure of the program's logic



QMH in classes

General class

Introduction

LabView class is a structure that consists of a cluster of private data and VIs that are the methods of the class.

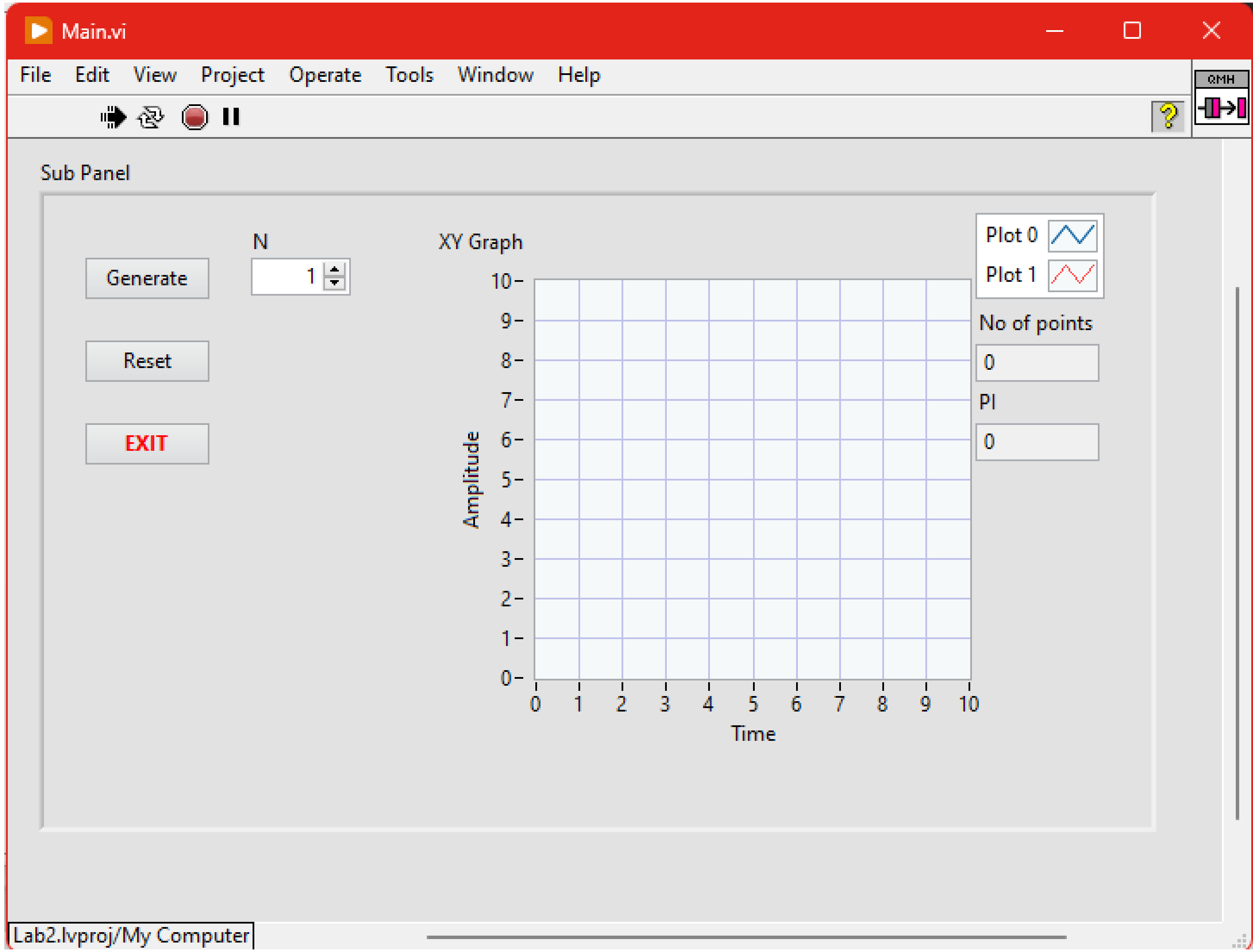
The class wire inside the class methods behaves as a cluster, so all data can be accessed or modified using unbundle or bundle. Specific VIs need to be made to give access to private data outside the class.

LabView class doesn’t have a constructor. The default values can be directly defined in the private data cluster (put the default value in the control, right-click, *Data Operation > Make Current Value Default*)

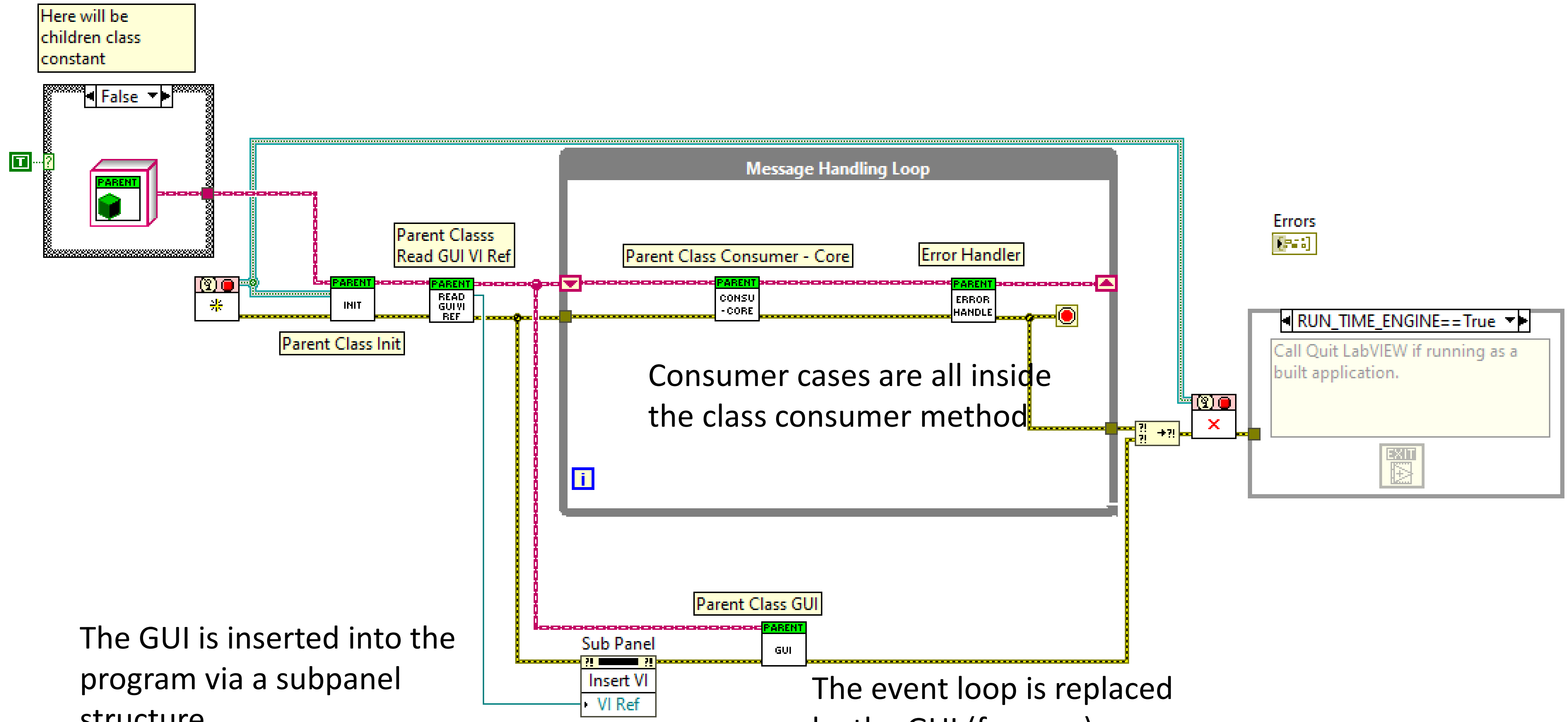
The goal

We want to understand the behaviour of a uService and how the messages work on a simpler example

Modify the *Queued Message Handler* such that the queue and GUI are specified inside a class.



An example of the main program



Here will be children class constant

Call Quit LabVIEW if running as a built application.

Consumer cases are all inside the class consumer method

The GUI is inserted into the program via a subpanel structure

The event loop is replaced by the GUI (for now)

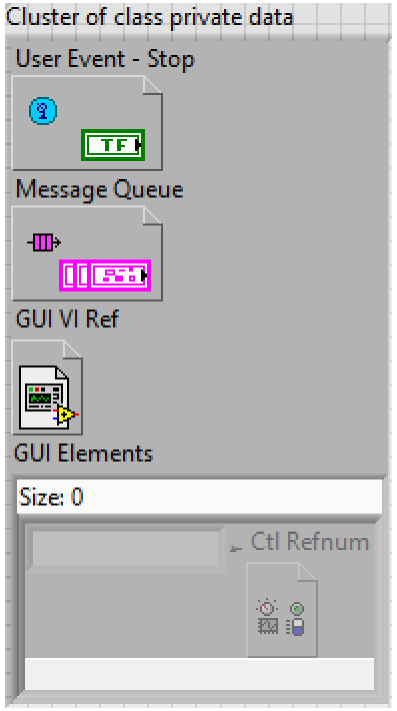
The SubProgram Parent Class

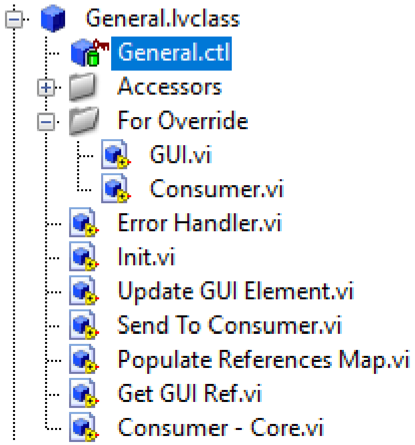
In the private data it should have:

- Reference to the Exit User Event
- Queue for messages
- GUI VI Reference
- References to all elements on the GUI (easiest as a map)

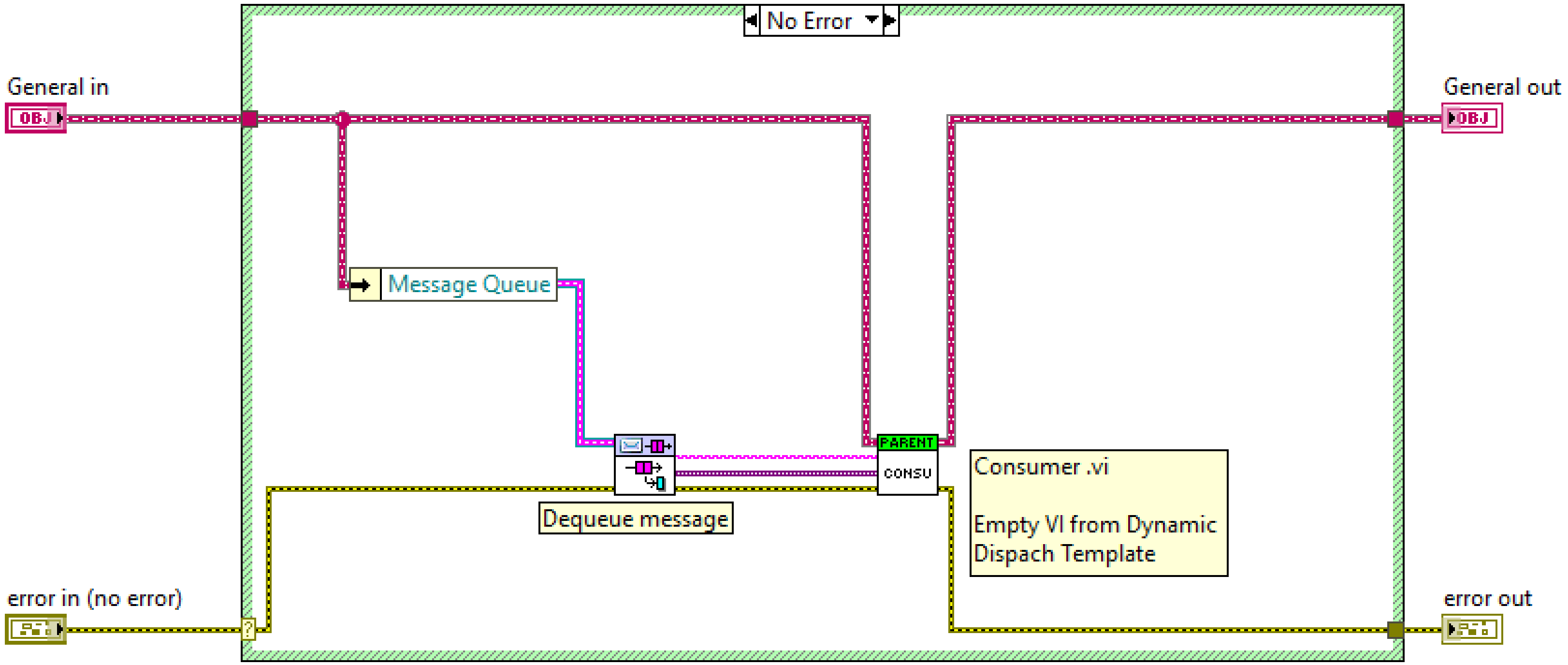
Required methods:

- Consumer – Core
- Populate reference map
- Get GUI Reference
- Update GUI value
- Send to Consumer (enqueue message)
- Init
- Error Handler
- Accessors (if needed)
  - Read n Register Stop Event
- For Override (Empty VI):
  - Consumer
  - GUI





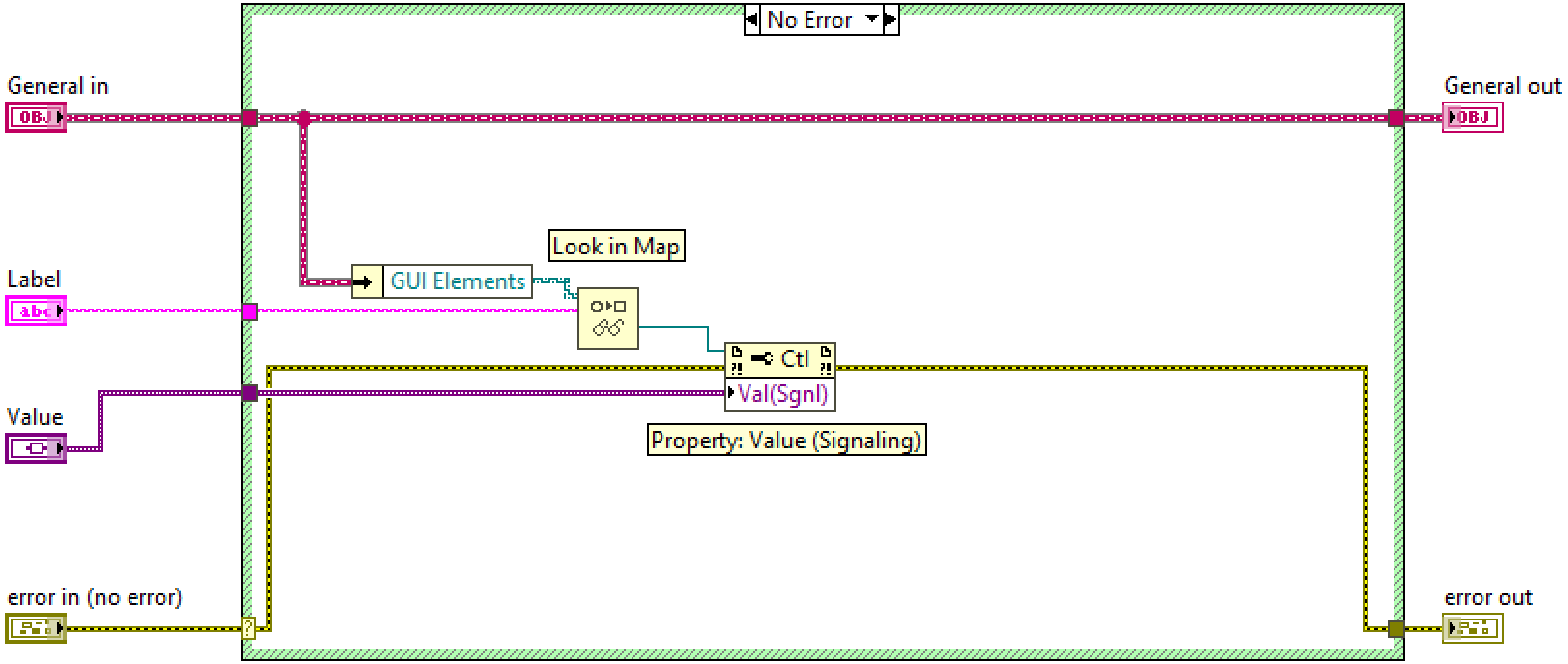
Consumer – core



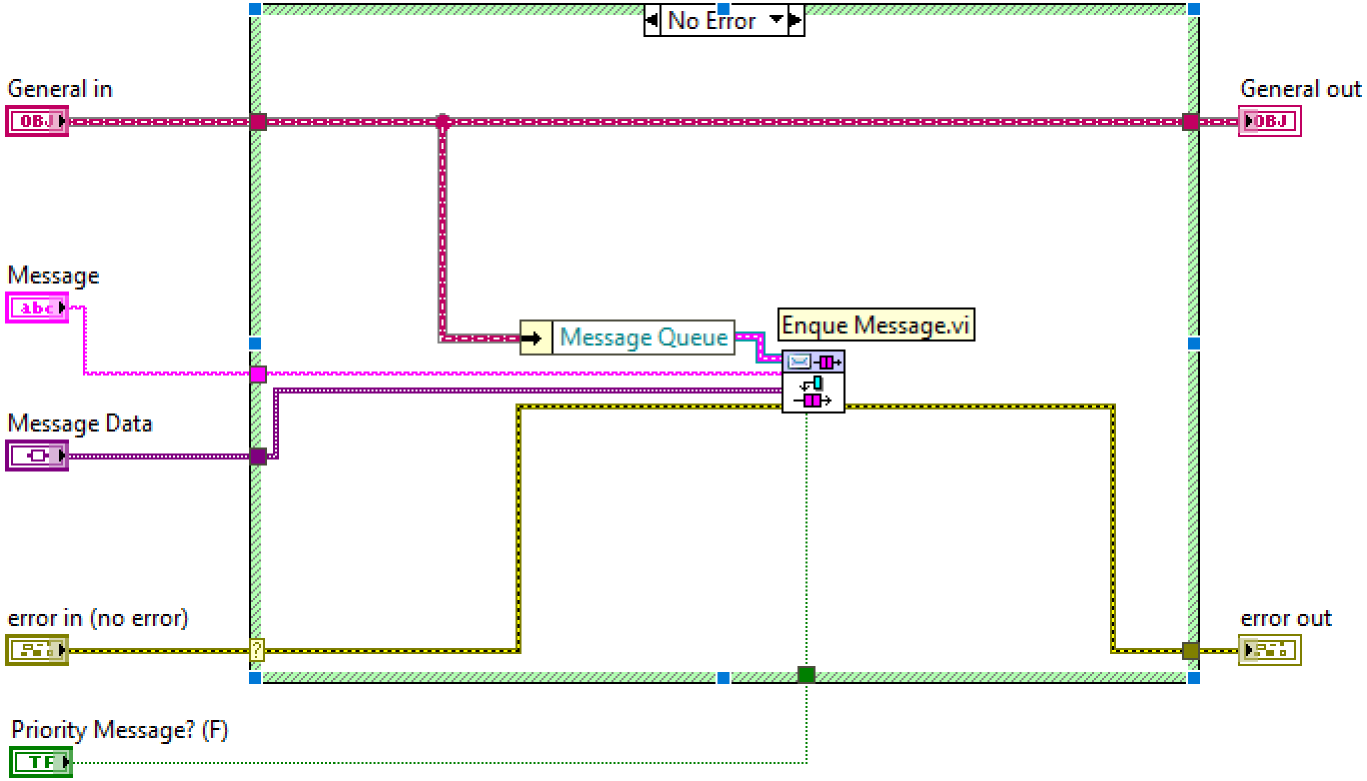
Consumer .vi

Empty VI from Dynamic Dispach Template

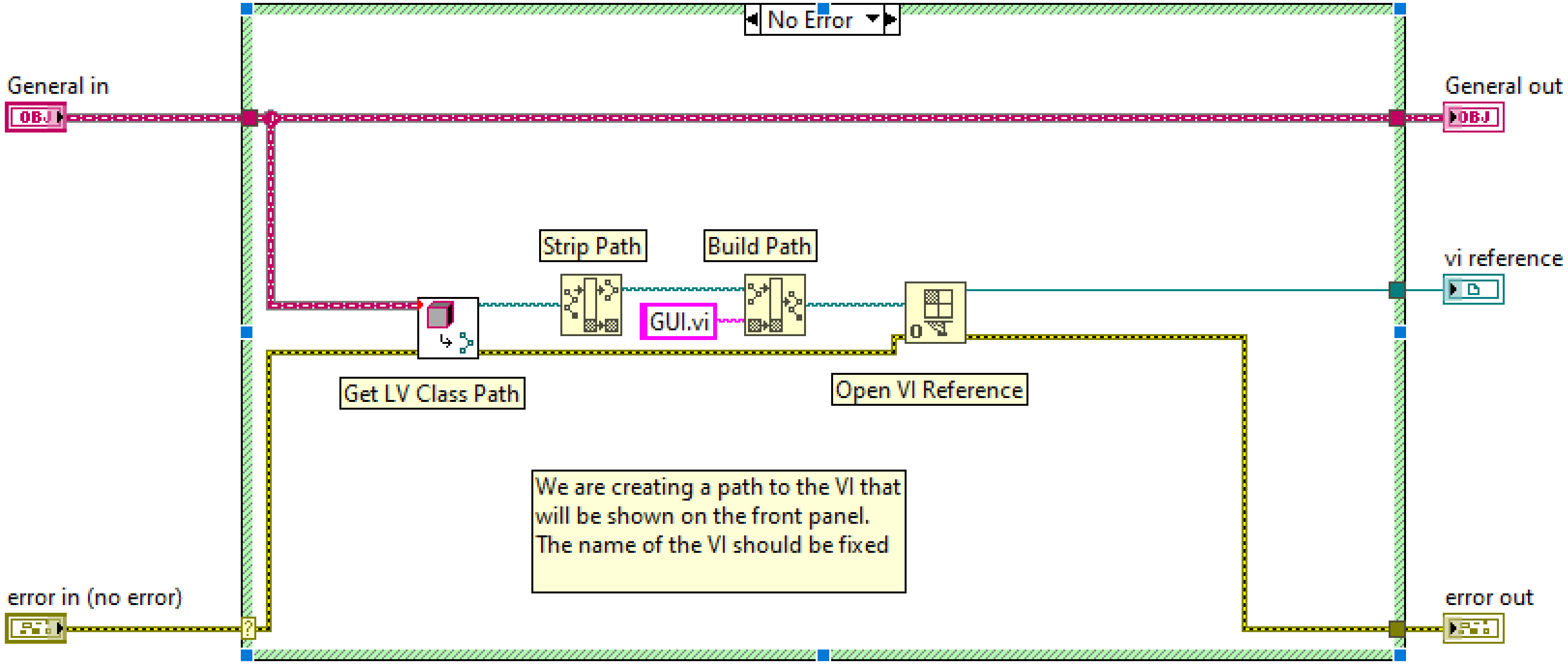
Update GUI Element



Send to Consumer

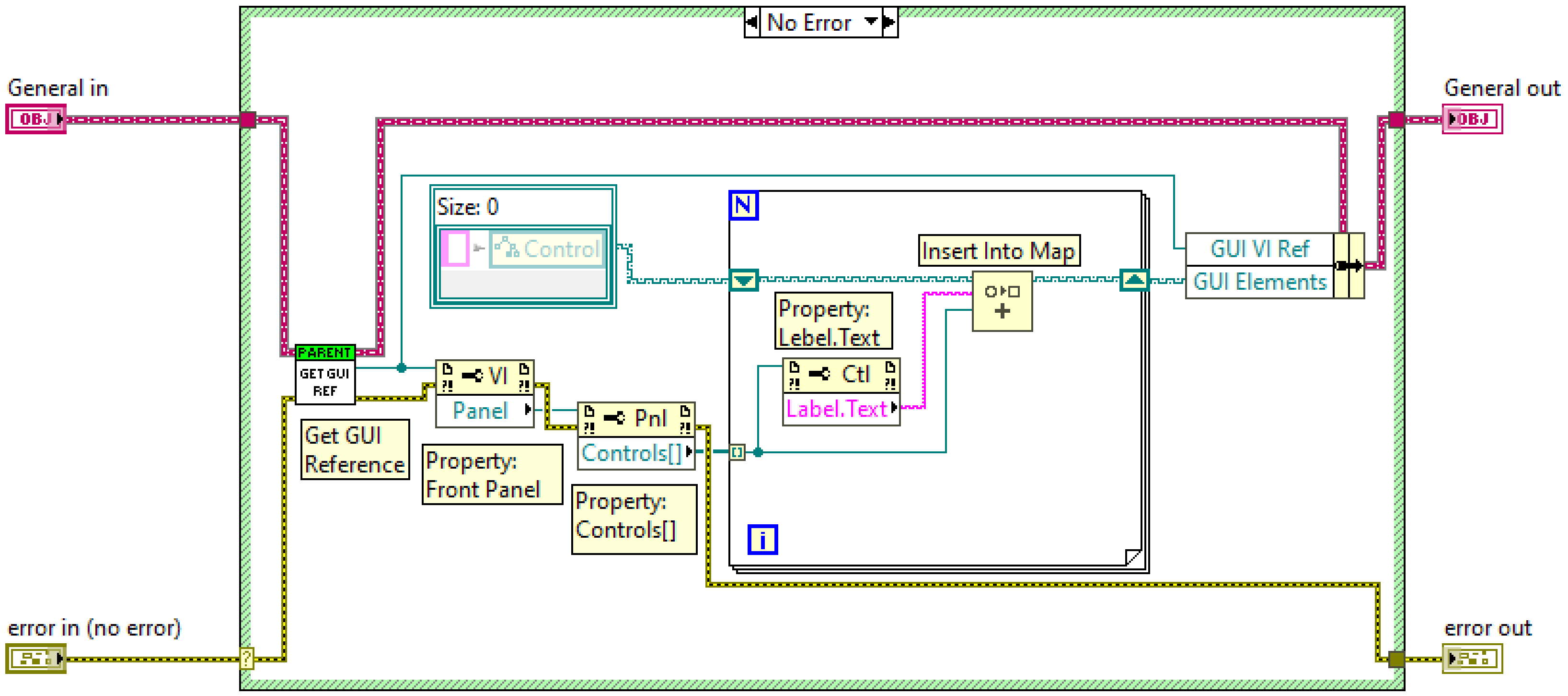


Get GUI reference

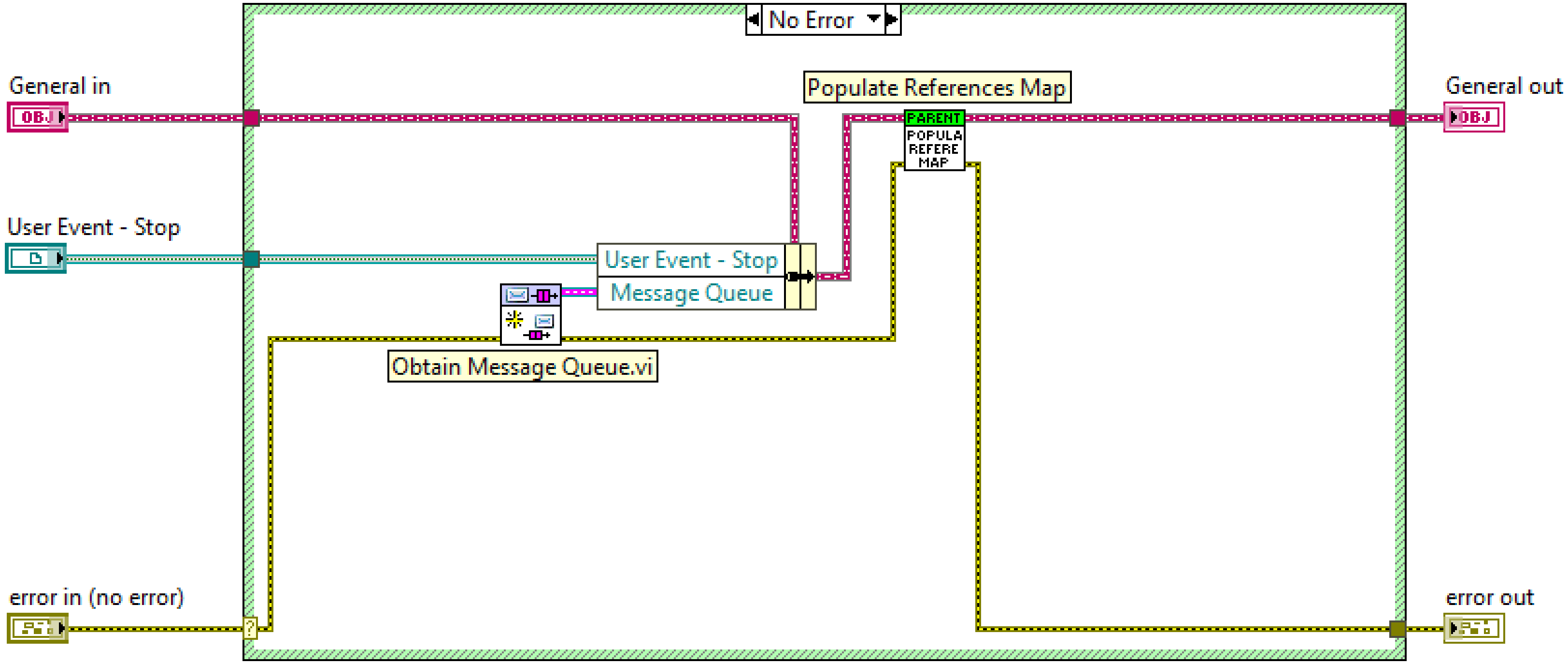


We are creating a path to the VI that will be shown on the front panel. The name of the VI should be fixed

Populate References Map

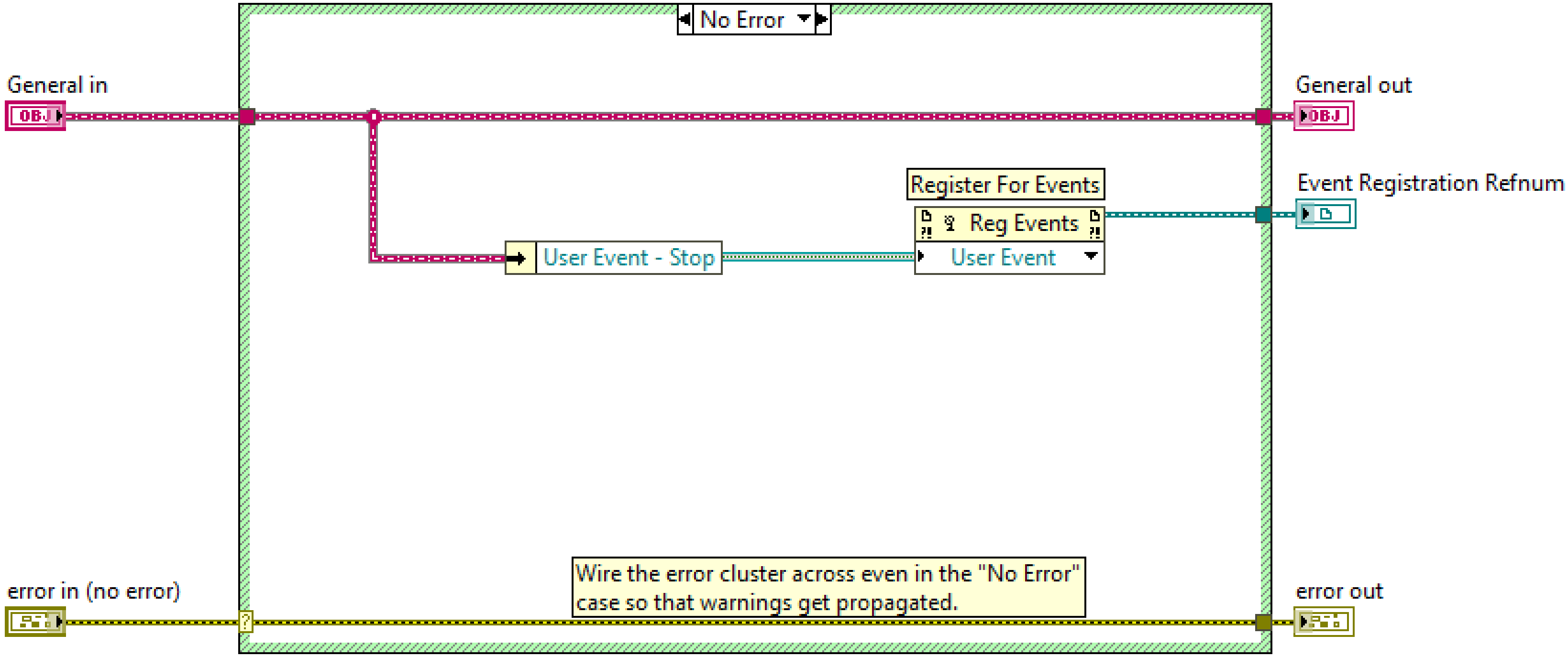


Init



No Error
General in
OBJ
User Event - Stop
error in (no error)
Populate References Map
PARENT
POPULA REFERE MAP
User Event - Stop
Message Queue
Obtain Message Queue.vi
General out
OBJ
error out

Read n Register Stop User Event

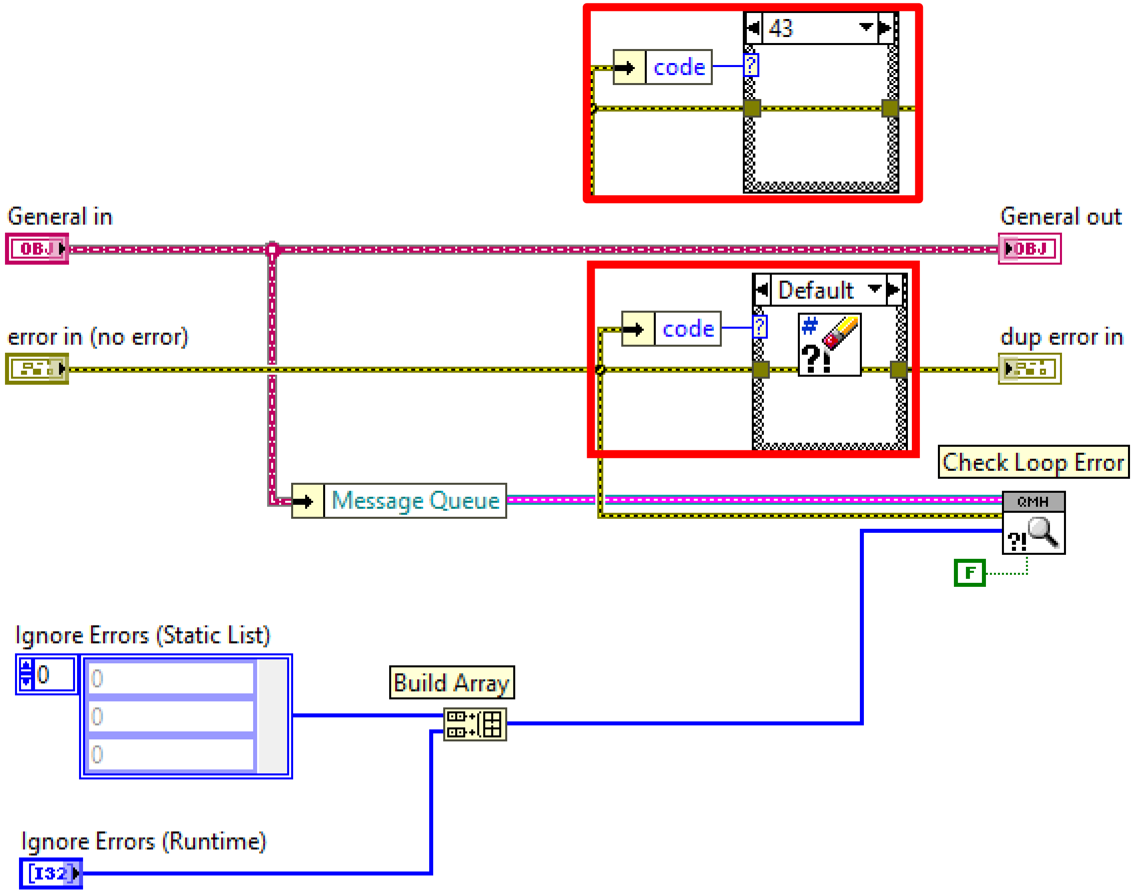


Wire the error cluster across even in the "No Error" case so that warnings get propagated.

Error Handler

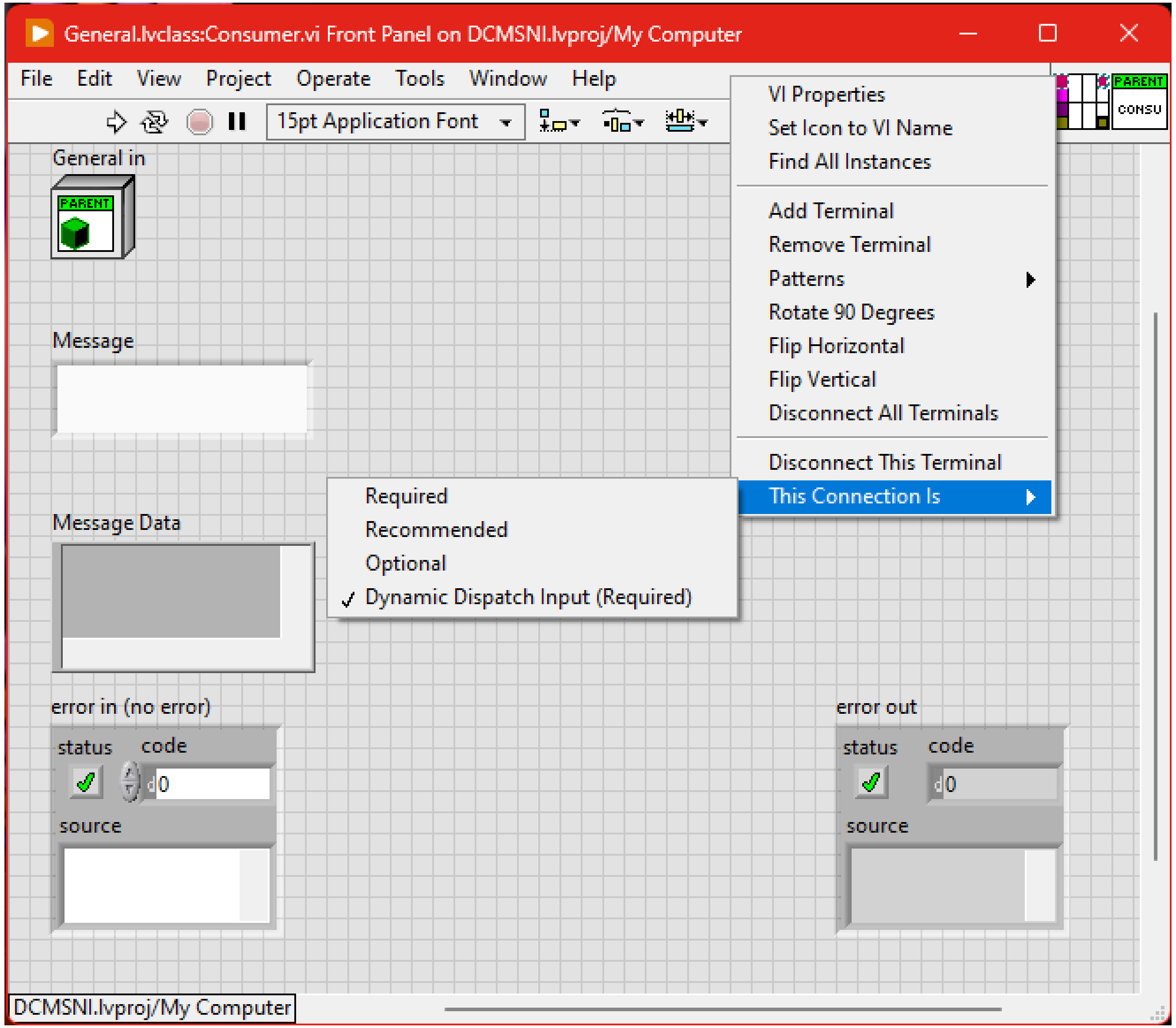
For the error handler, we will just use the handler from the example but modify it to use our class instead. This isn't an actual error handler; it's more of an error notifier

We are going to use error 43 (generated in the exit case to stop the loop



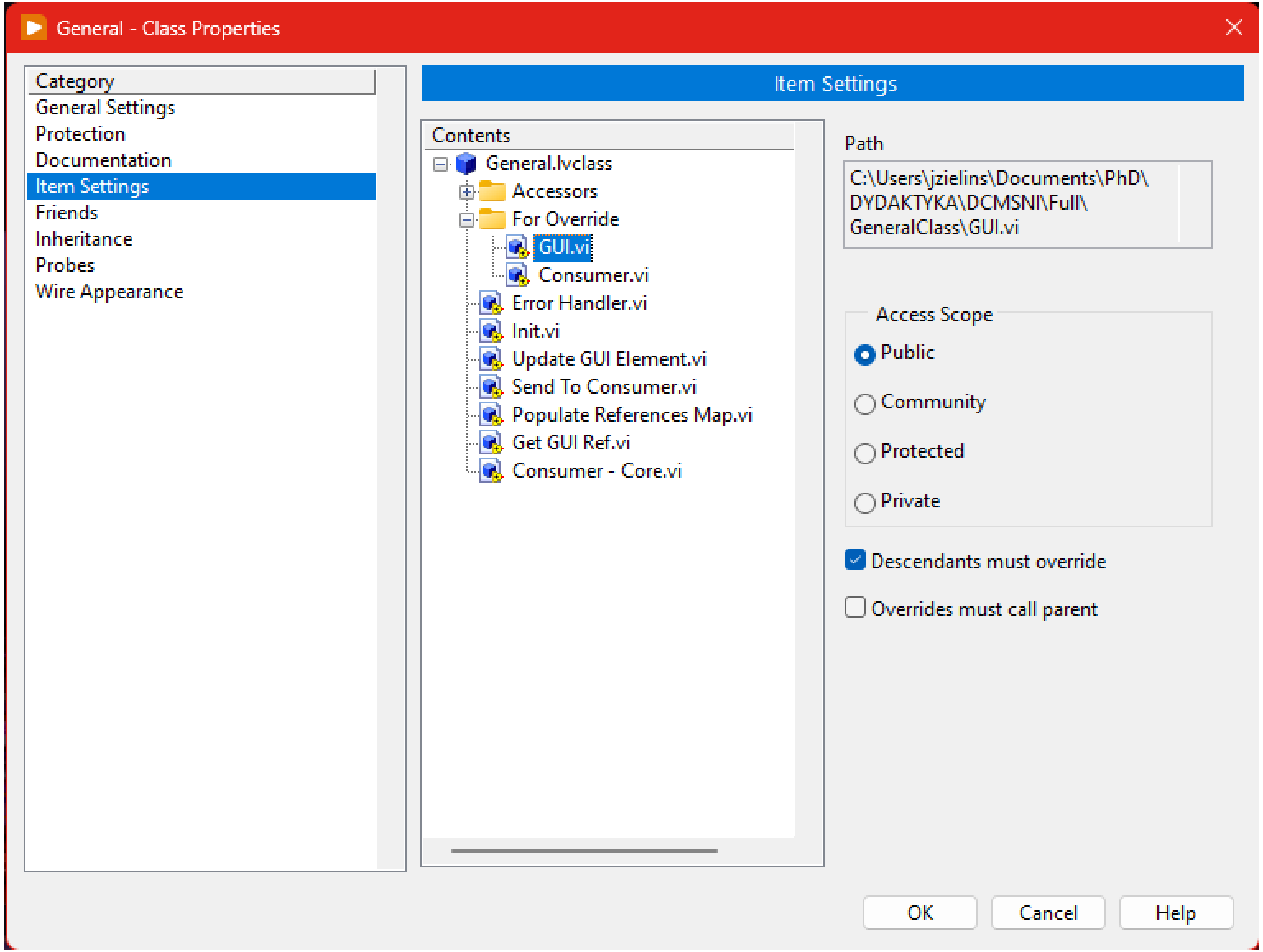
Methods for override

Any method for override needs to be made according to the Dynamic Dispatch Template.
It means that connections for the class input and output need to be set as *Dynamic Dispatch Input/Output (Required)*



Methods for override

Then, by going into the class properties, we can select the given method and make it necessary to override for all descendants.
Additionally we can specify if the parent method needs to be executed by descendants. In our case, the second option is not required.



Path
C:\Users\jzielins\Documents\PhD\DYDAKTYKA\DCMSNI\Full\GeneralClass\GUI.vi

Access Scope

● Public

○ Community

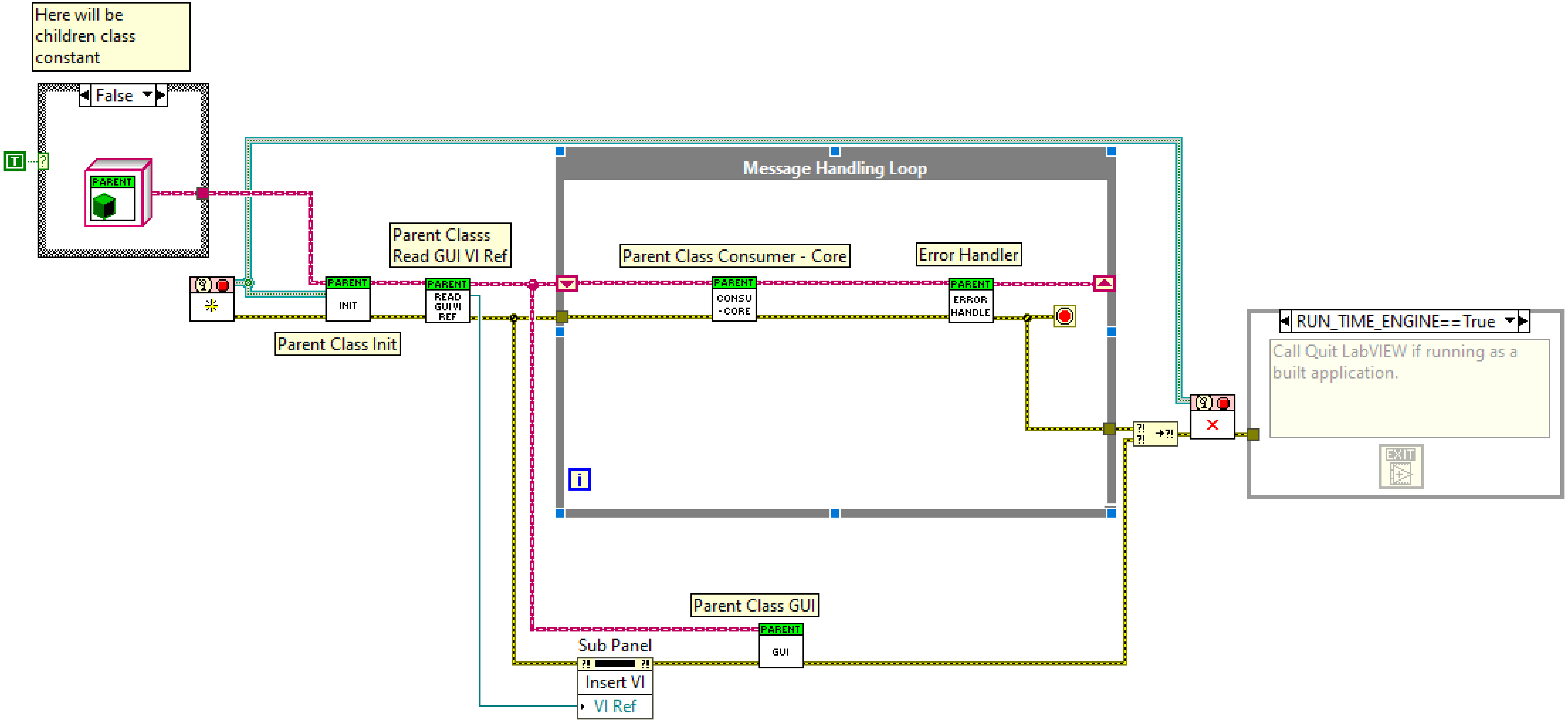
○ Protected

○ Private

☑ Descendants must override

☐ Overrides must call parent

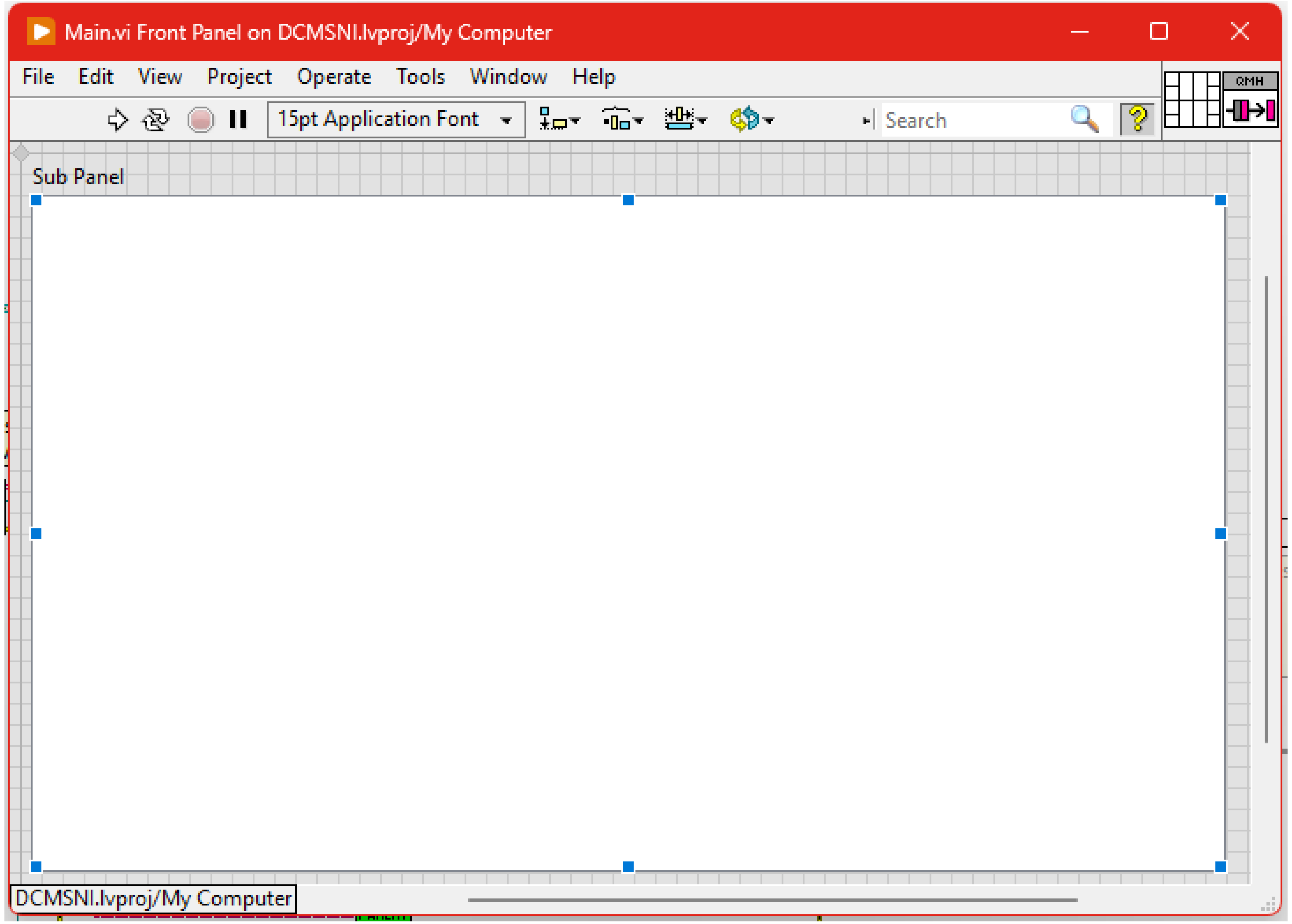
Simplifying Main VI



Here will be children class constant

Call Quit LabVIEW if running as a built application.

Simplifying Main VI (Front Panel)



I am using Sub Panel from System section (this is just visual difference). For most front panel elements on the GUI I am using System or NXG style

QMH in classes

Descendant class

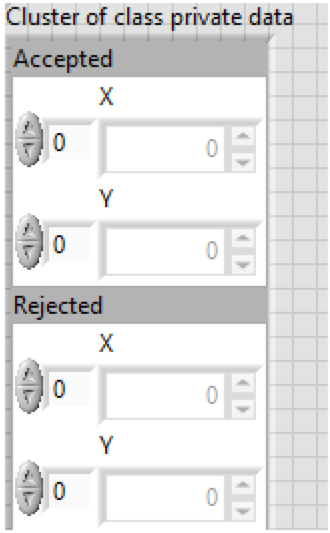
The SubProgram Child Class

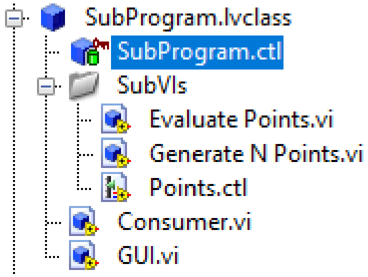
In the private data it should have:

- A cluster of accepted points
- A cluster of rejected points

Required methods:

- Consumer (override)
- GUI (override)
- Optional:
  - Generate Points
  - Evaluate PI



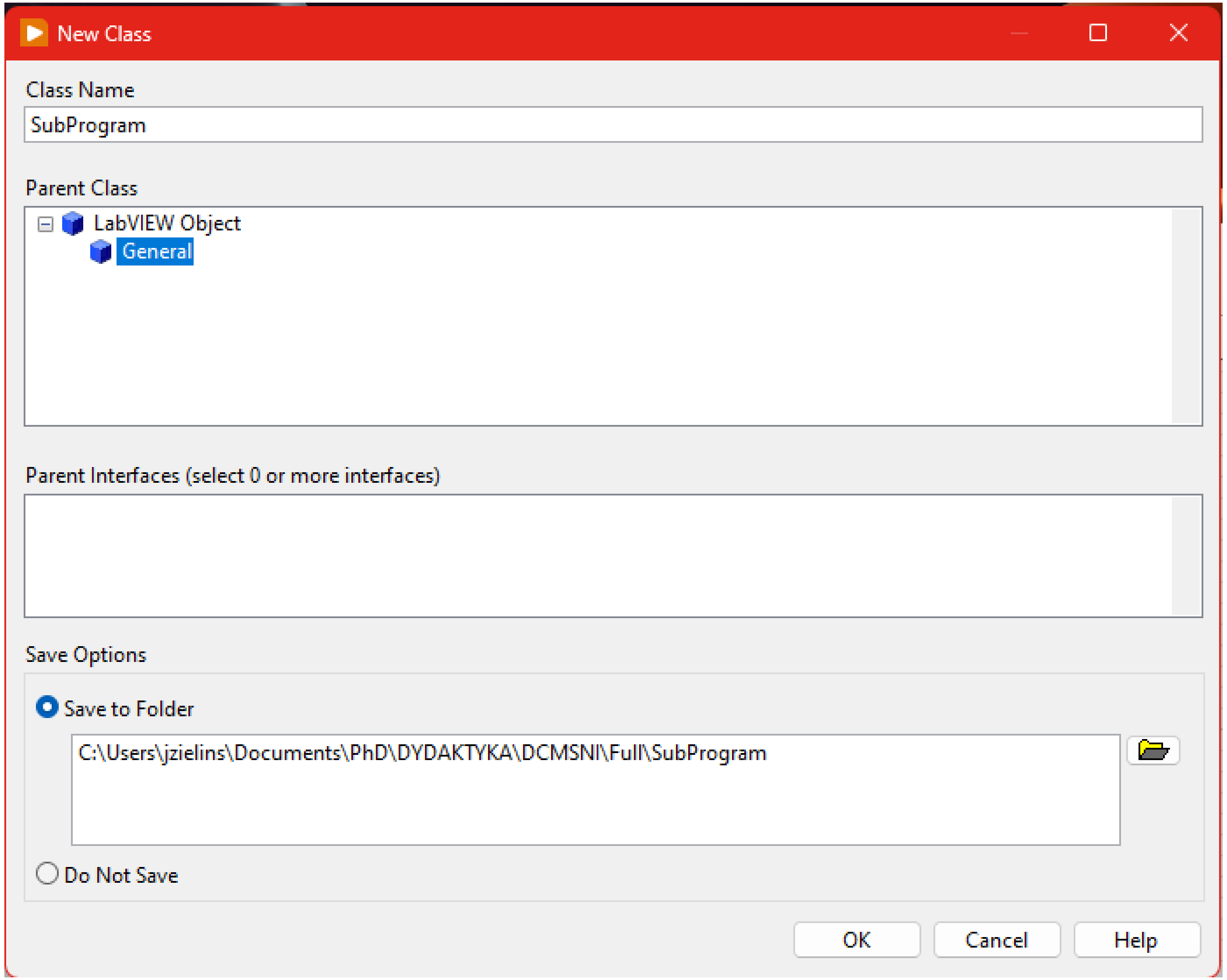


```
SubProgram.lvclass
    SubProgram.ctl
    SubVIs
        Evaluate Points.vi
        Generate N Points.vi
        Points.ctl
    Consumer.vi
    GUI.vi
```

As an example, this class will do the job of calculating PI using the Monte-Carlo method.
But, this class can be done in any way you wish, doing any specific action using the producer/consumer scheme. Feel free to experiment

Creating the class

We want to make the new class inherit from the General class



Class Name

SubProgram

Parent Class

- LabVIEW Object
  - General

Parent Interfaces (select 0 or more interfaces)

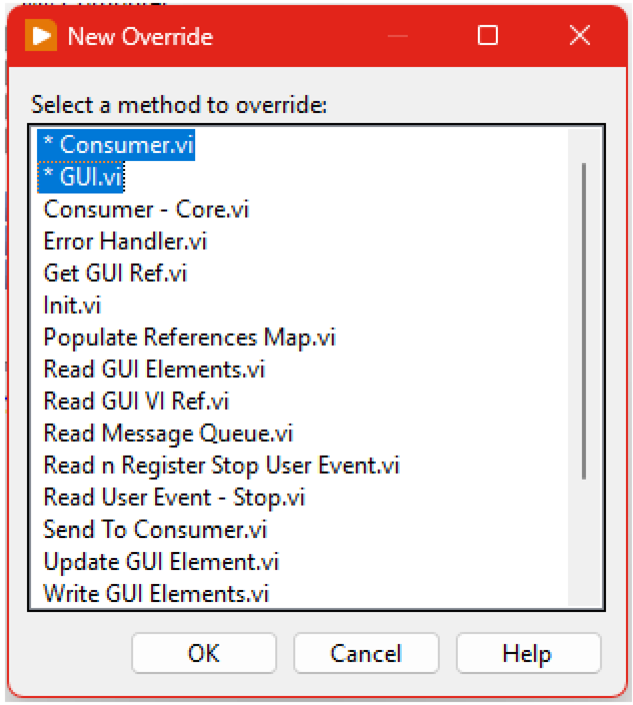
Save Options

● Save to Folder

C:\Users\jzielins\Documents\PhD\DYDAKTYKA\DCMSNI\Full\SubProgram

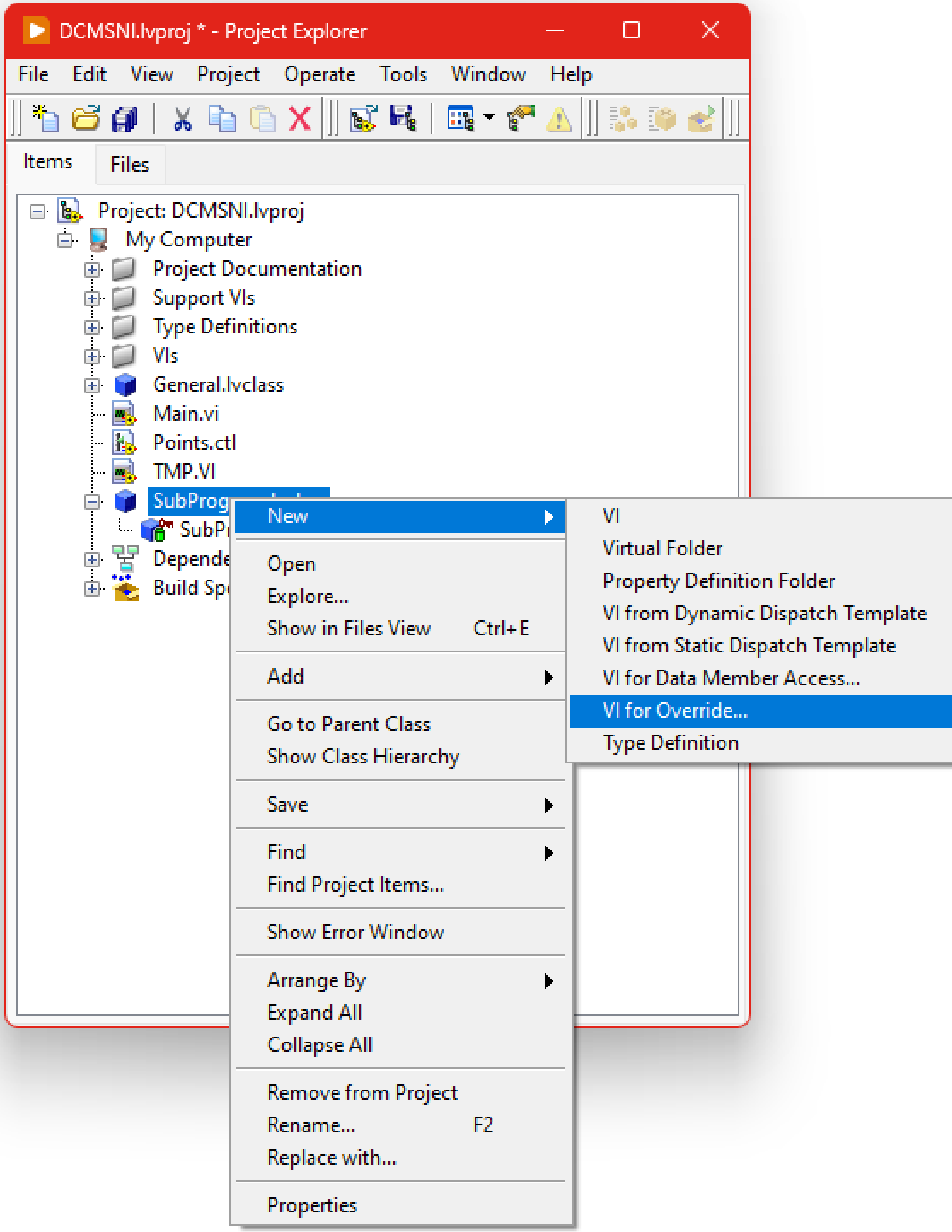
○ Do Not Save

Adding needed methods



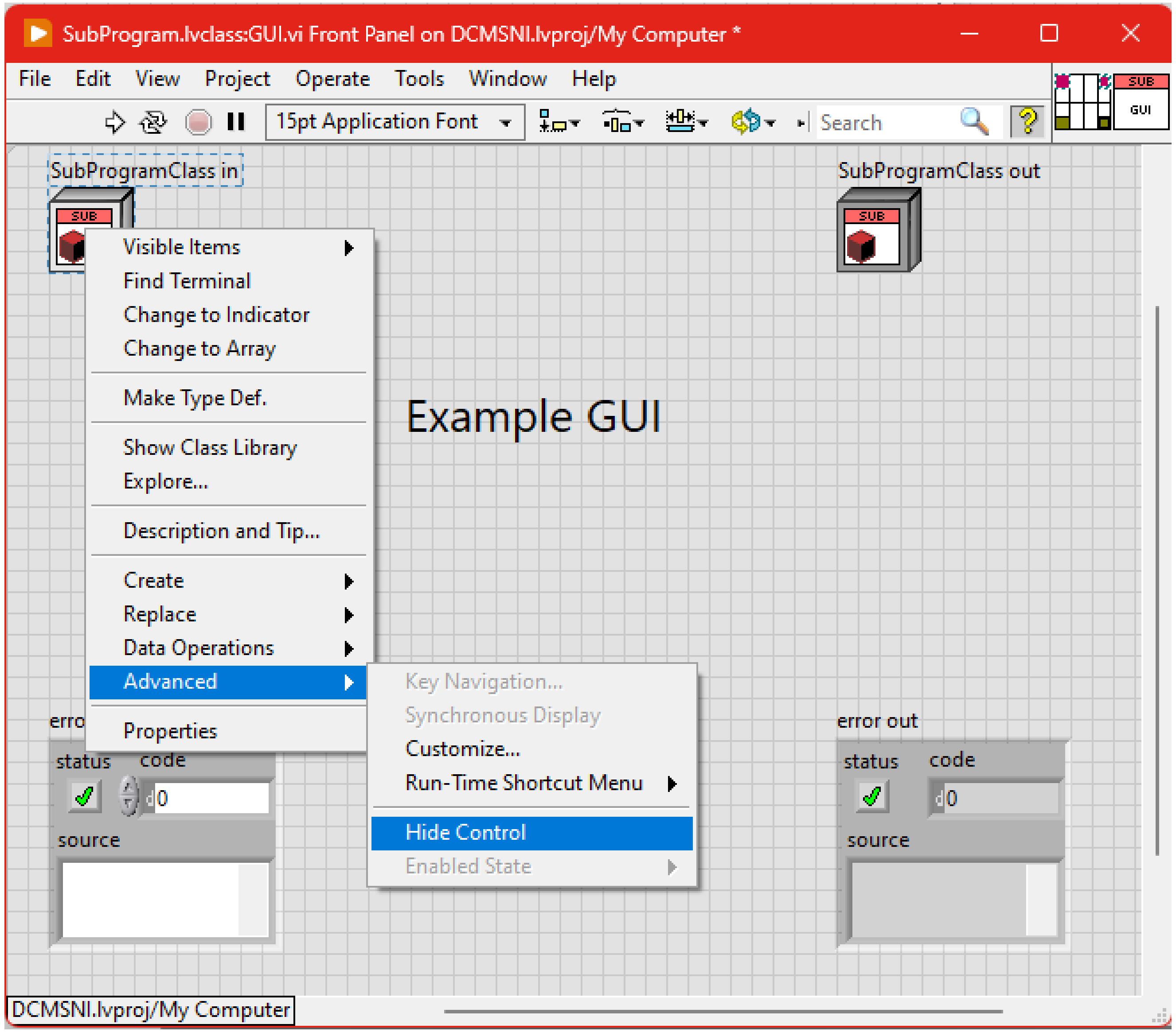
Select a method to override:

* Consumer.vi
* GUI.vi
Consumer - Core.vi
Error Handler.vi
Get GUI Ref.vi
Init.vi
Populate References Map.vi
Read GUI Elements.vi
Read GUI VI Ref.vi
Read Message Queue.vi
Read n Register Stop User Event.vi
Read User Event - Stop.vi
Send To Consumer.vi
Update GUI Element.vi
Write GUI Elements.vi

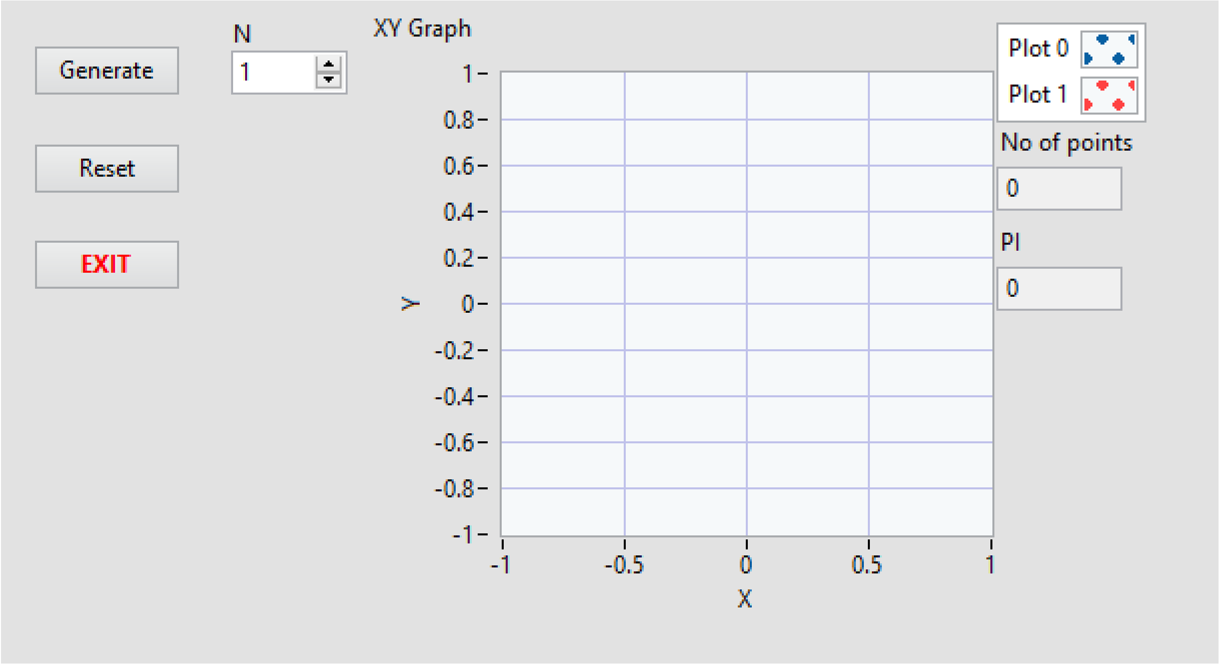


GUI

We don't want to see the Class controls and errors on the GUI, so we hide the controls



GUI



N

1

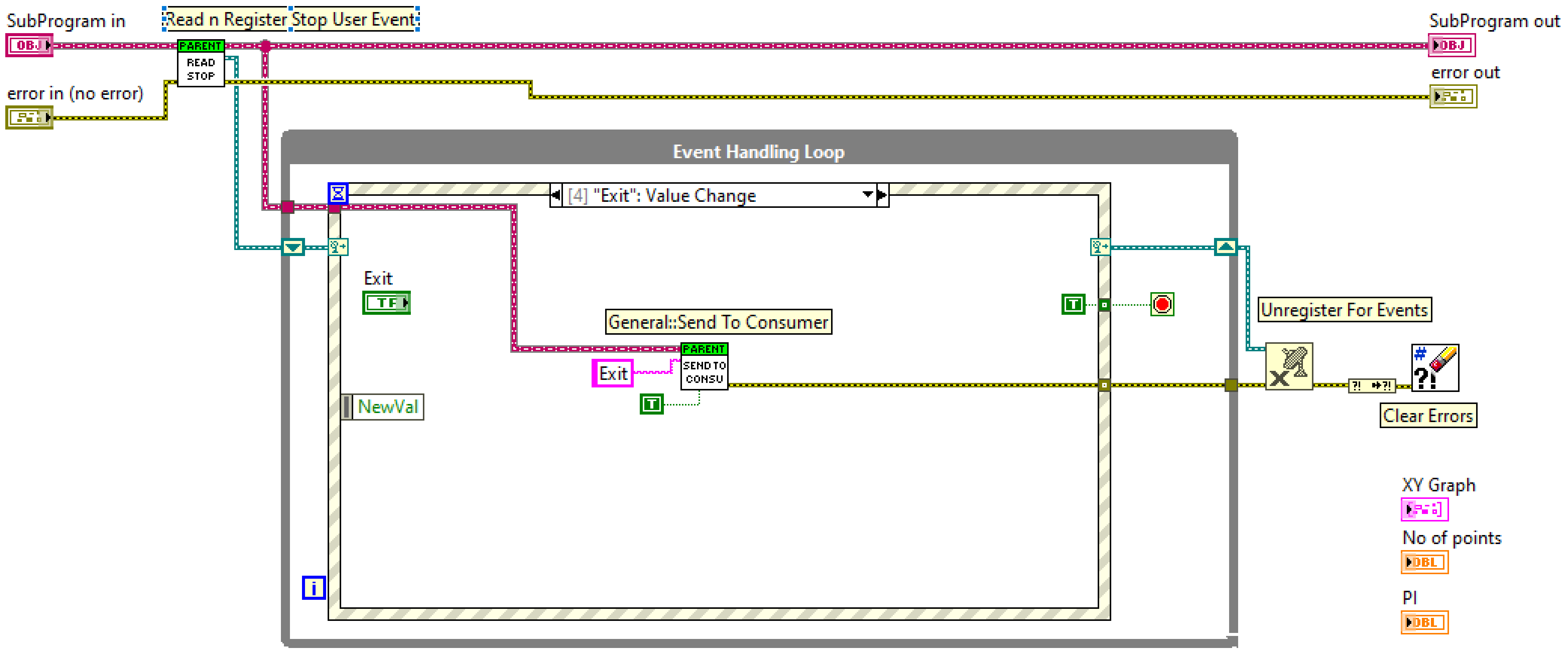
No of points

0

PI

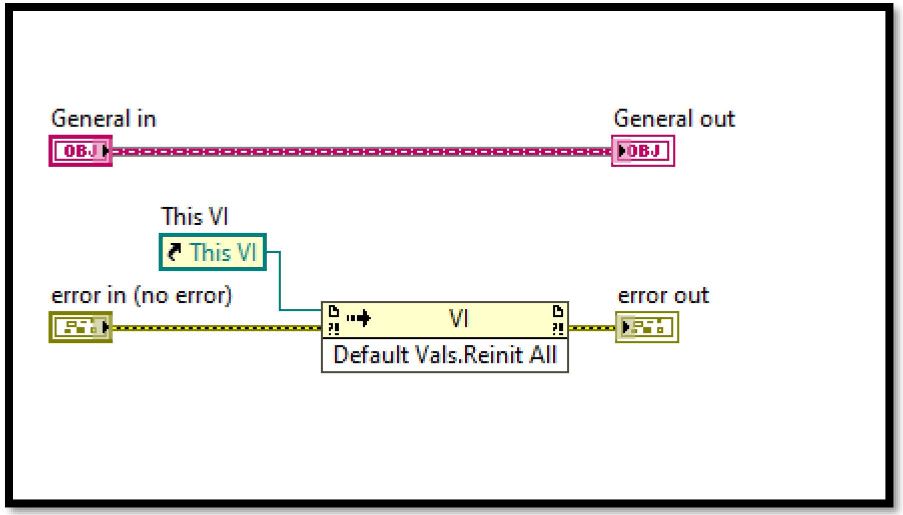
0

GUI

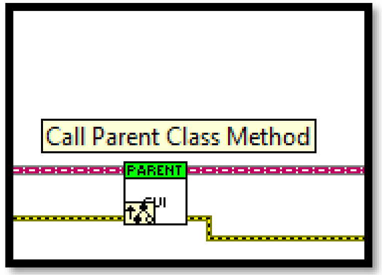

SubProgram in
OBJ
Read n Register Stop User Event
PARENT
READ STOP
error in (no error)
SubProgram out
OBJ
error out
Event Handling Loop
[4] "Exit": Value Change
Exit
TF
General::Send To Consumer
PARENT
SEND TO CONSU
Exit
T
NewVal
T
Unregister For Events
Clear Errors
XY Graph
No of points
DBL
PI
DBL
i

Optional: reinitialize to default using parent class

Parent class



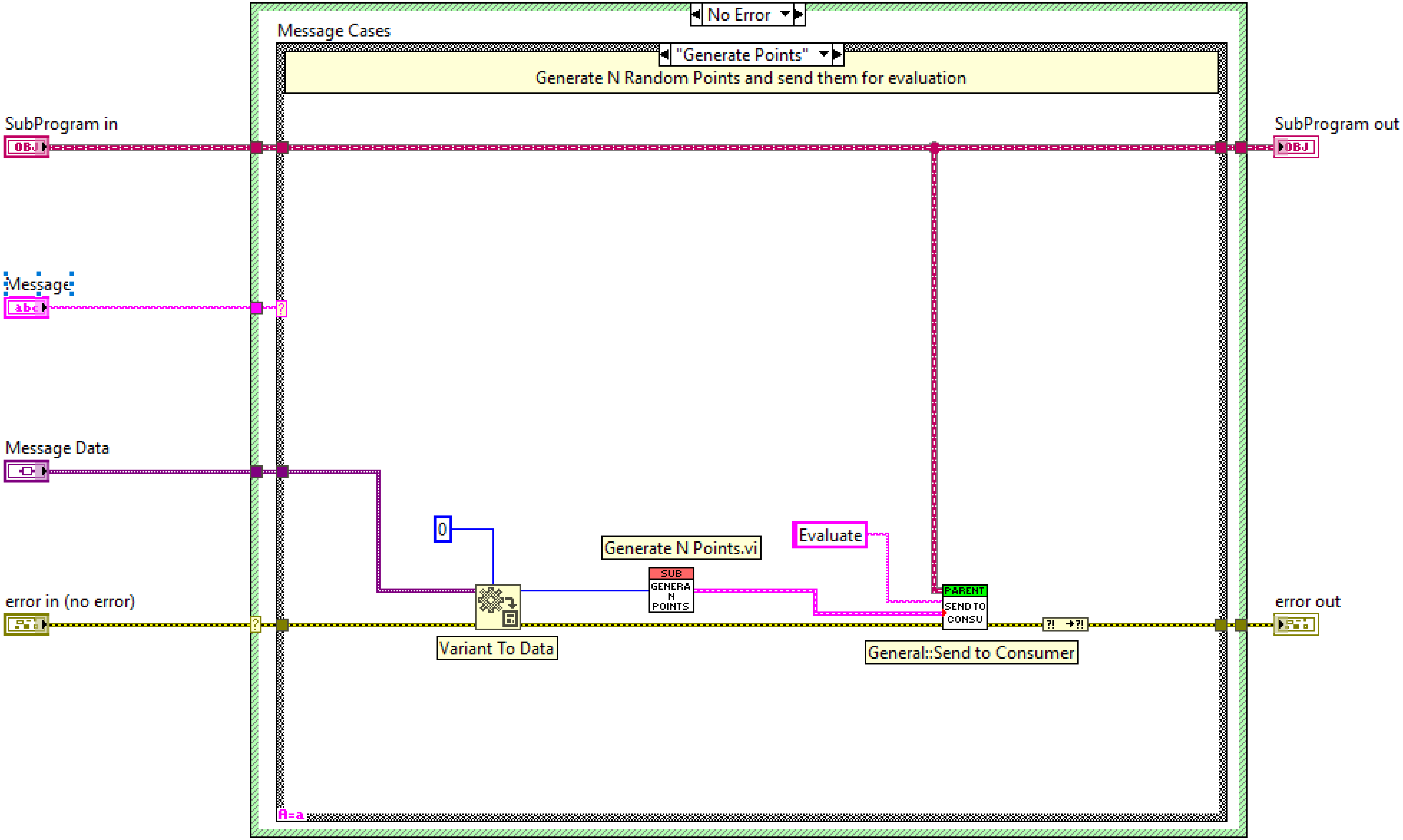
Child class



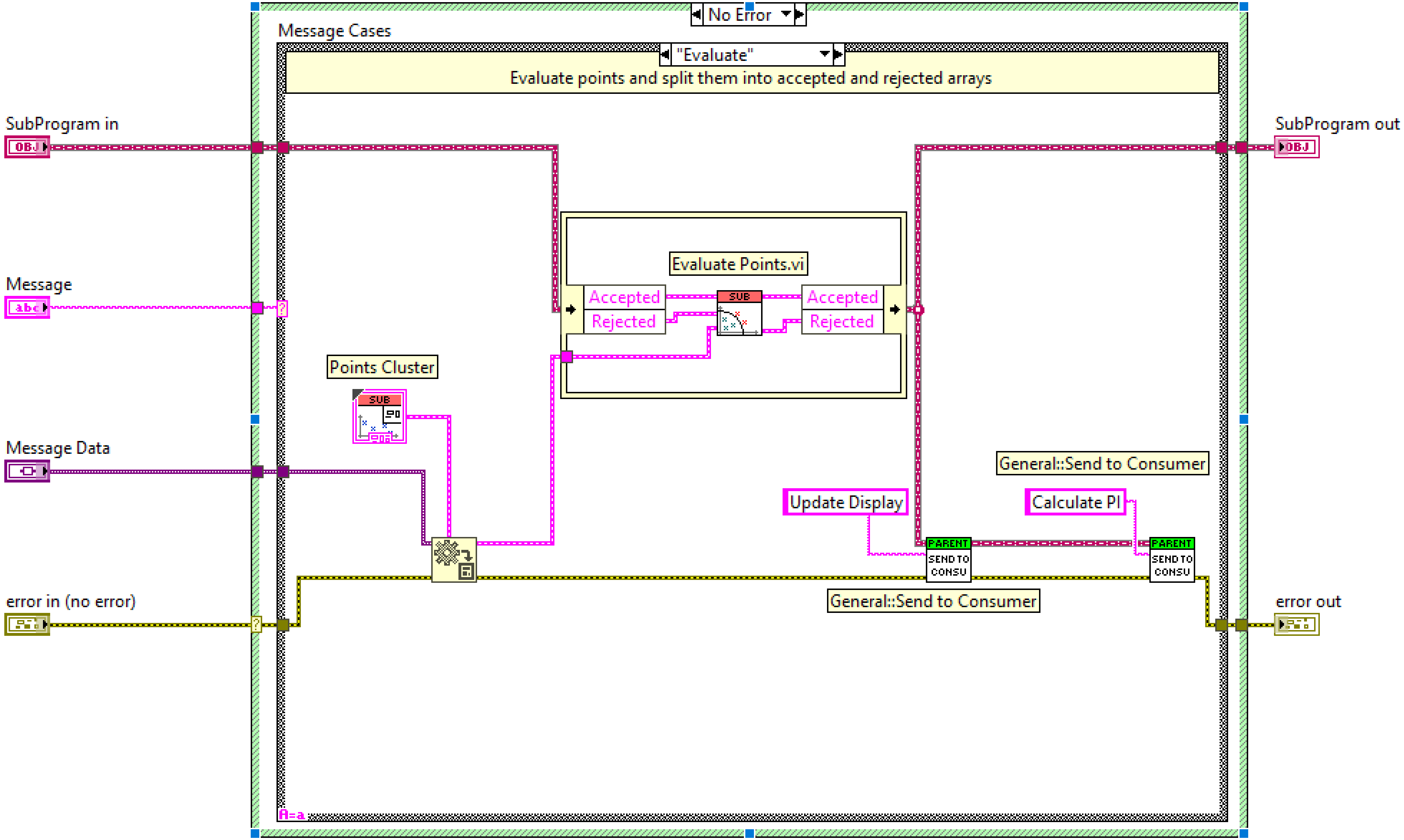
Consumer

Consumer will get the full consumer case. The cases must be modified so that the class is fully used. Instead of using the Enqueue message, we will replace them with Send to Consumer.vi. Any place for updating the value on the GUI should be replaced with the Update GUI Element.vi

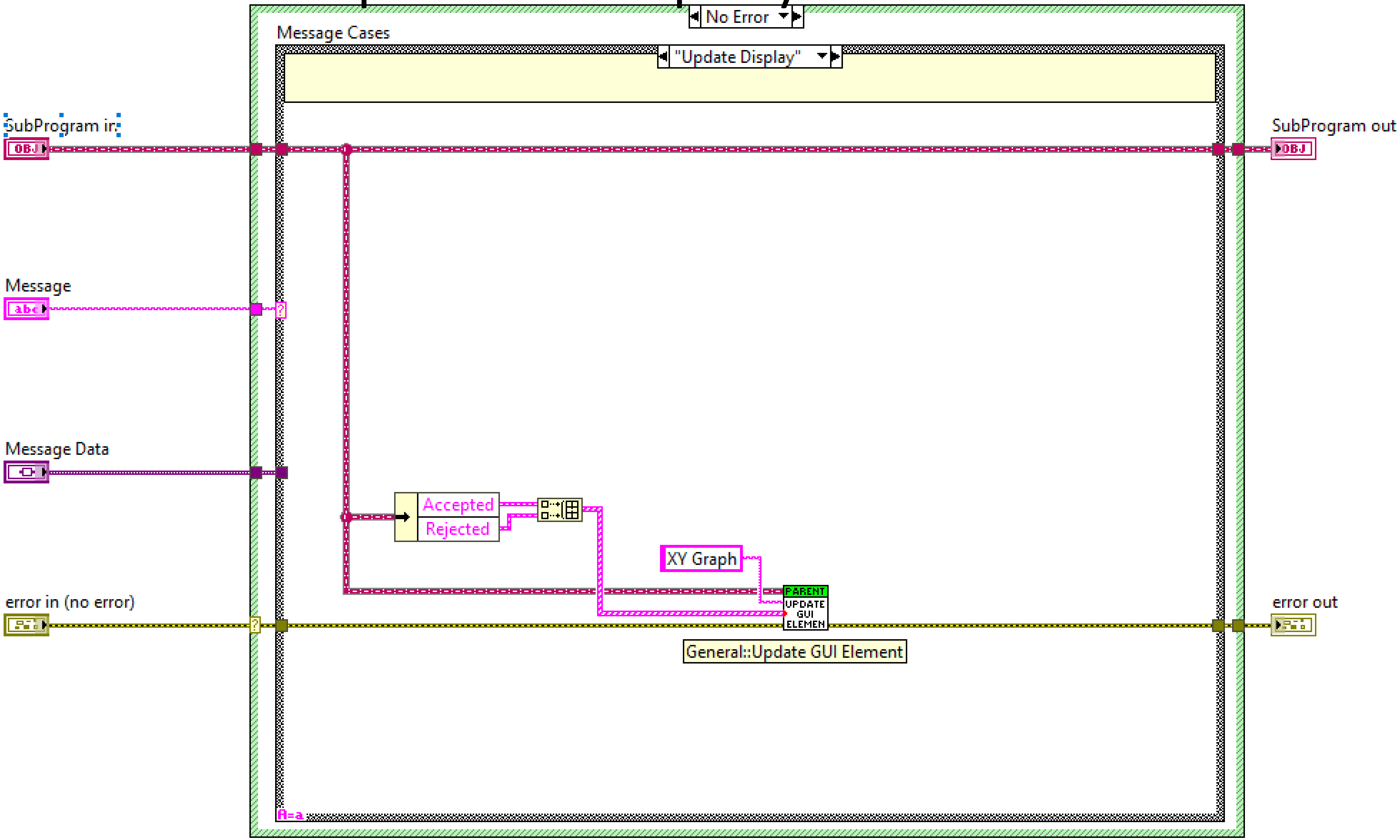
Consumer: Generate Points



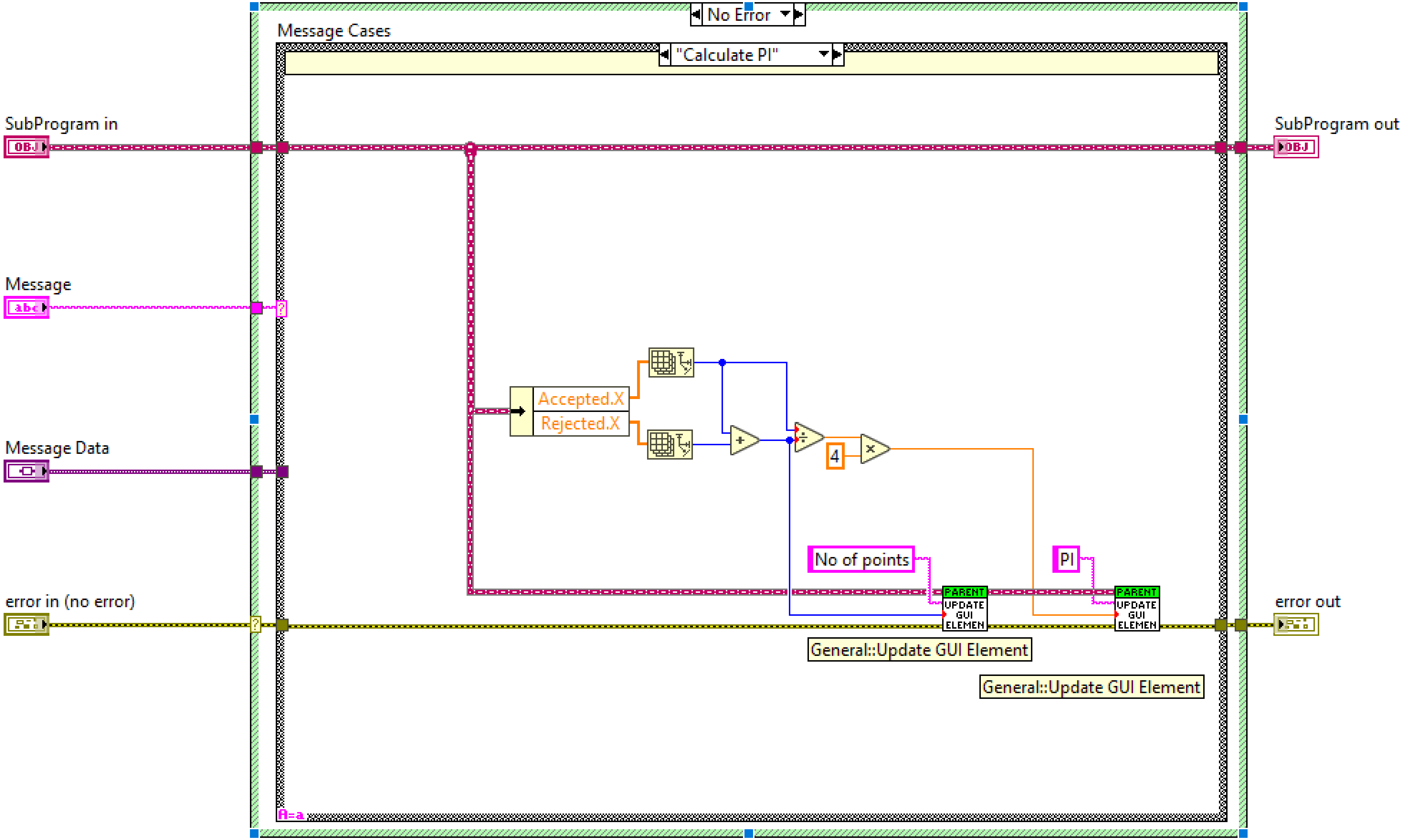
Consumer: Evaluate



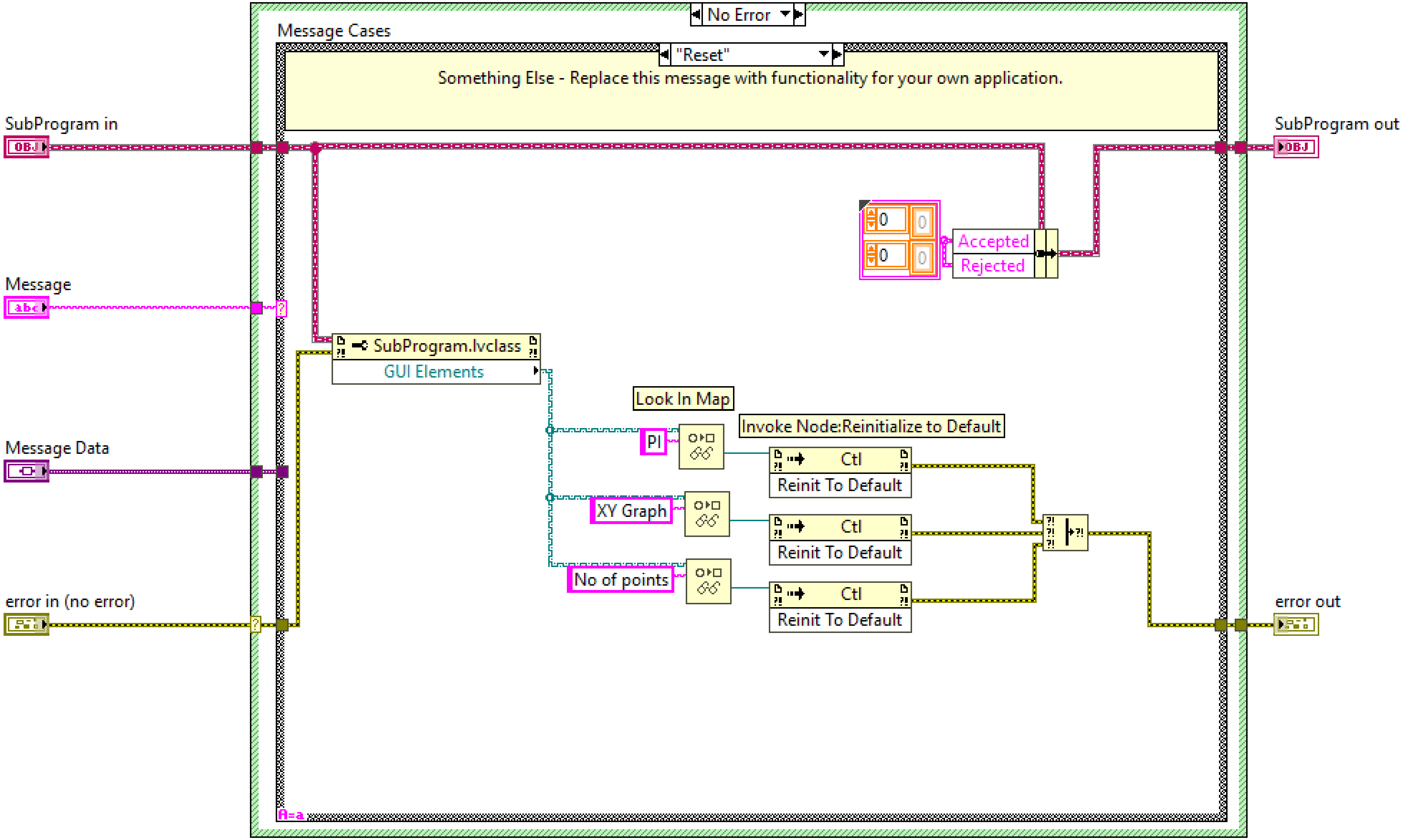
Consumer: Update Display



Consumer: Calculate PI

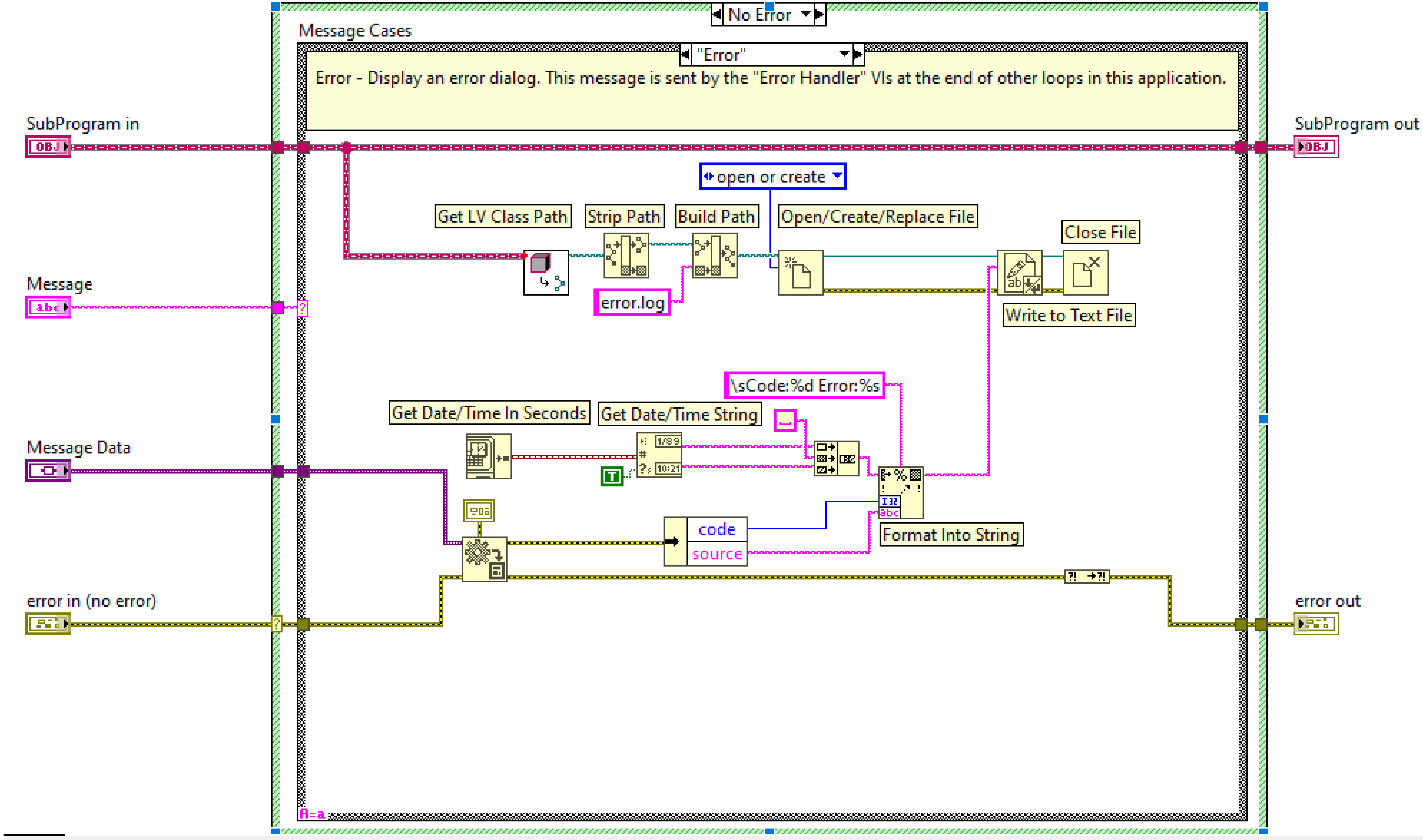


Consumer: Reset



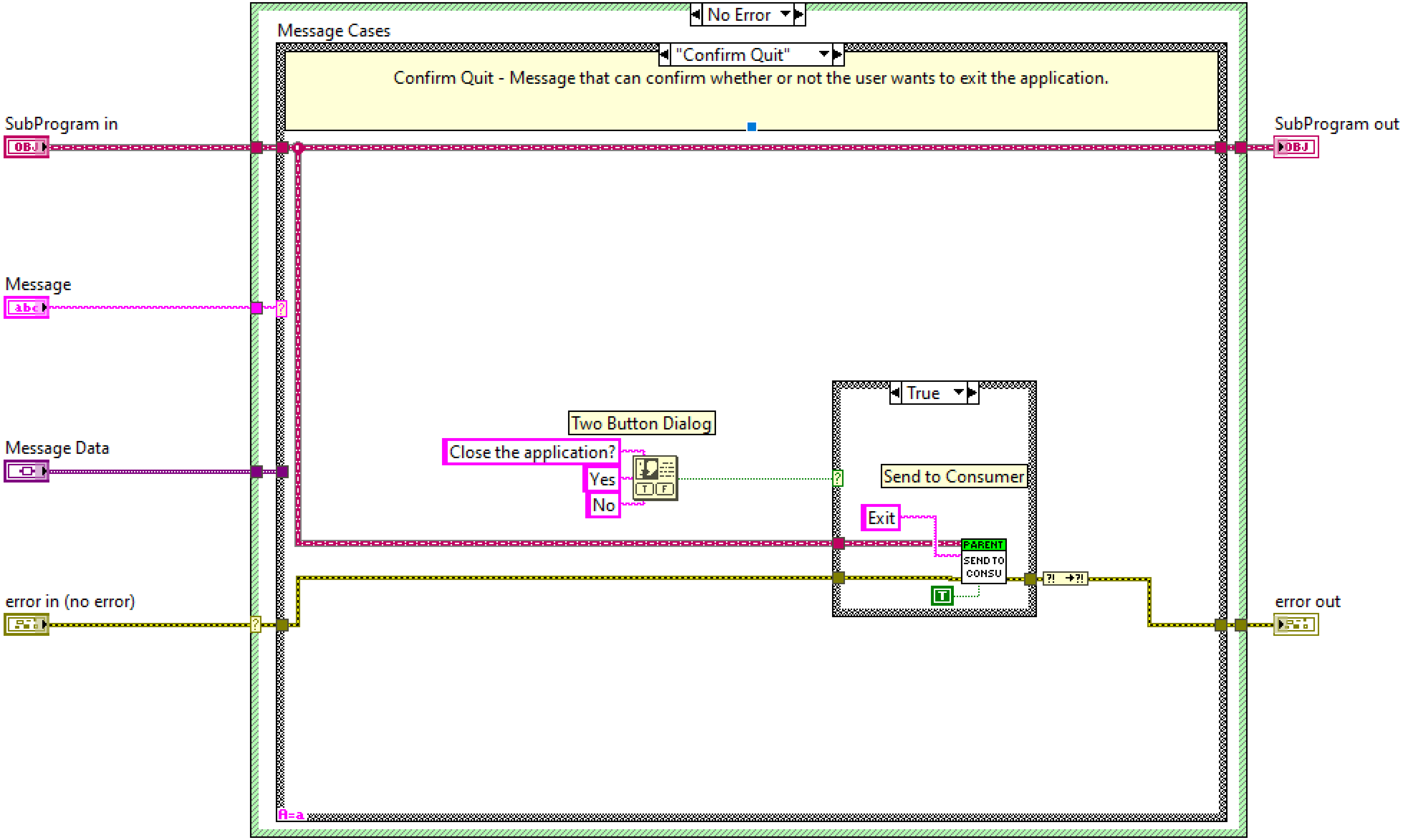
Consumer: Error

This case is used got logging errors in a file.

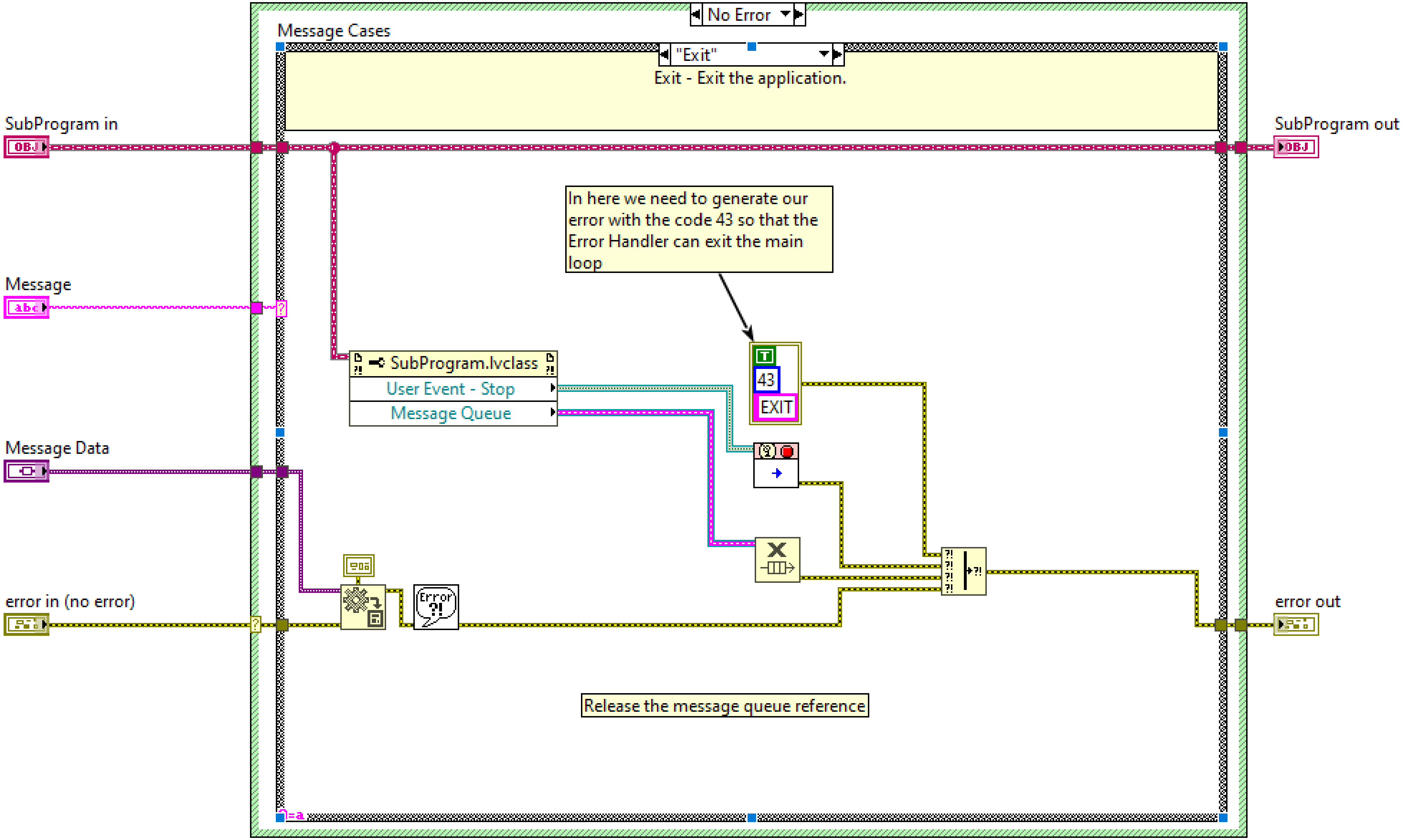


Error - Display an error dialog. This message is sent by the "Error Handler" VIs at the end of other loops in this application.

Consumer: Confirm Quit



Consumer: Exit

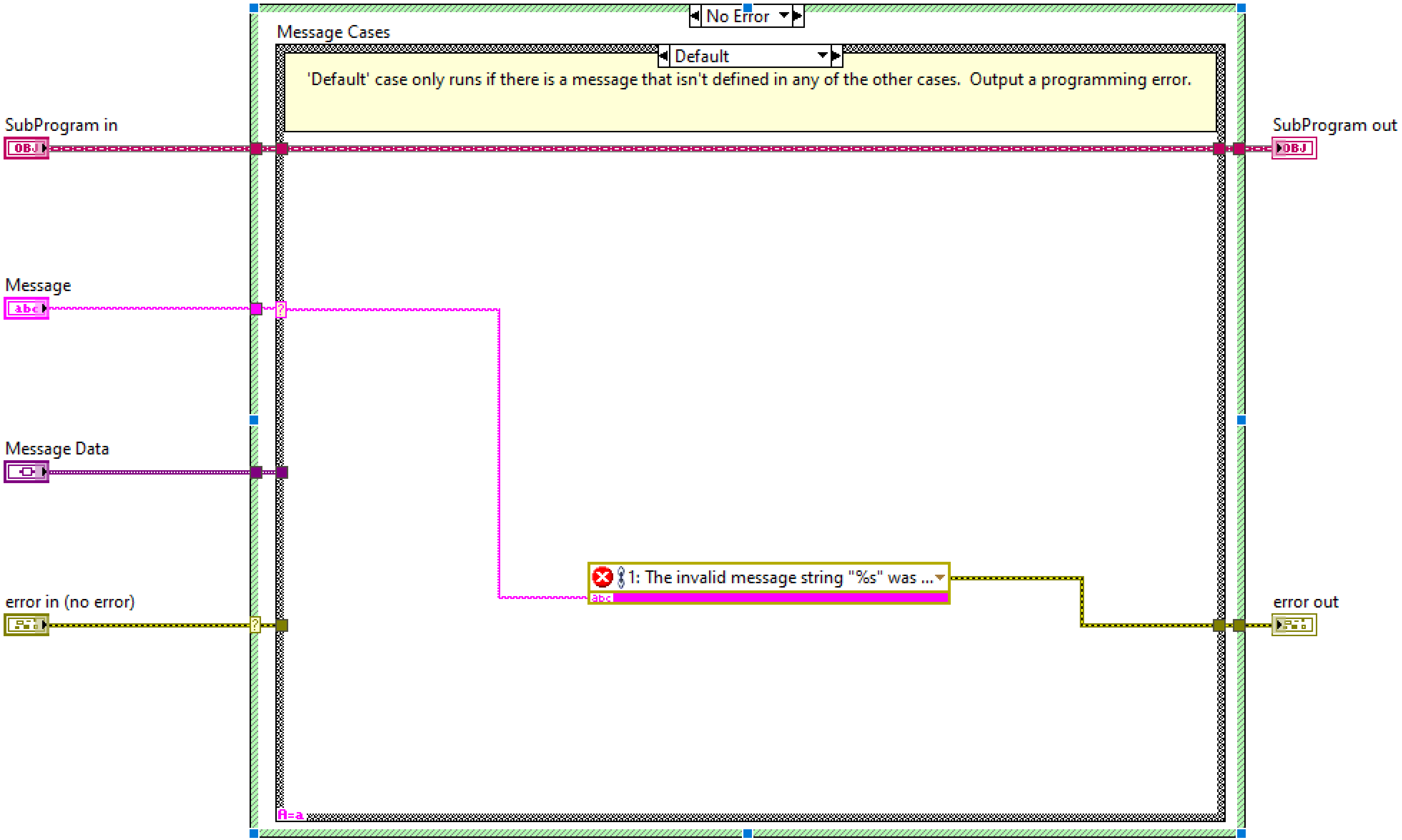


Exit - Exit the application.

In here we need to generate our error with the code 43 so that the Error Handler can exit the main loop

Release the message queue reference

Consumer: Default



'Default' case only runs if there is a message that isn't defined in any of the other cases. Output a programming error.

1: The invalid message string "%s" was ...