wörkers™

FUNDAMENTALS

Support guide to the Workers™ Fundamentals sample project

October 2019

Contents

Getting Started with Workers™

This sample project is the recommended starting place for all those who are new to Workers™. Workers™ employs the same design structure as the LabVIEW Queued Message Handler, while being modular and highly scalable at the same time. It is assumed that you are already familiar with the LabVIEW Queued Message Handler before proceeding further with this document.

For those who are new to Workers™, this guide will introduce to you the main features of the API and will also introduce you to the *Workers™ tools*, to help you efficiently build-up and debug your Workers™ applications.

In addition to the Fundamentals sample project, we also have a Workers™ version of the well-known 'Continuous Measurement and Logging' sample project that ships with LabVIEW. We created this so that you can compare side-by-side the classic LabVIEW version of this project with the Workers™ version of this project, to make obvious the similarities and differences between the two methods.

For further information, the Workers™ User Guide.pdf, which is a reference manual for the framework and tools, can be consulted.

In summary, here is a list of all of the available help material created for the framework.

- Fundamentals.pdf (this document here)
- Workers™ User Guide.pdf (found under LabVIEW menu: Help >> Workers™)
- Context and detailed help for all framework VIs in the Workers™ palette

Here is a list of all the sample projects that are provided:

- Workers™ Fundamentals sample project
- Workers™ Continuous Measurement and Logging sample project

Happy programming! ☺

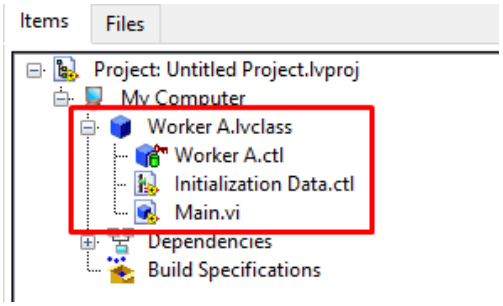
Structure

Worker Contents

A Worker is defined as:

***Simple definition**: A modular Queued Message Handler*

Every Worker can be thought of as a library, having the ".lvclass" extension, and ***must*** contain the following items:



- *(Worker name) . lvclass*

The class that contains the VIs and controls of the Worker. Think of this class simply as a library that has its own type definition. For the majority of developers, this is all you need to know about the use of LabVIEW classes in the framework.

- *(Worker name) . ctl*

The Worker's private data cluster, accessible only from VIs within your Worker. This data cluster can be used to store any data types that you require to be read and written to by the Worker's QMH.

- *Initialization Data . ctl*

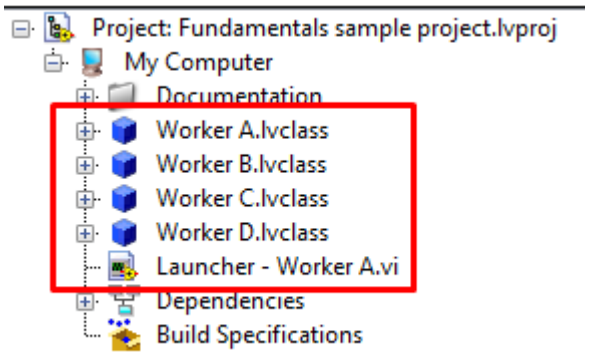
This is a cluster that you define. Any data that you want the Worker to receive when it is initialized can be added to this cluster.

- *Main . vi*

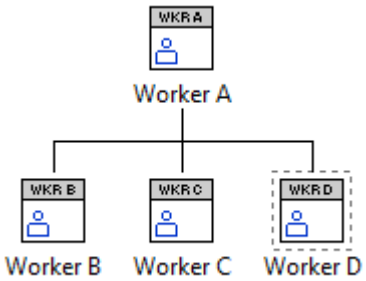
This VI contains the Queued Message Handler of your Worker.

Project Explorer Contents

The Fundaments sample project contains four Workers, and one Launcher VI, as shown below.

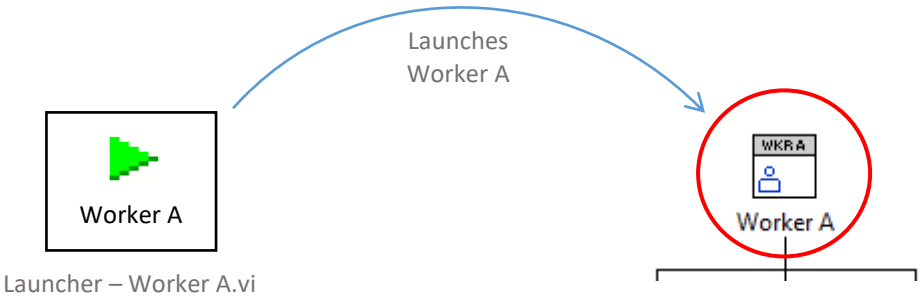


These Workers are linked together to form an application. **Worker A** is launched by the Launcher VI (*Launcher – Worker A.vi*). **Worker A** is statically-linked to **Worker B** and **Worker C** and these all load together when the application is launched. **Worker A** loads **Worker D** dynamically, on demand. The Worker call-chain hierarchy of this application can be illustrated by the diagram below.



Note: the dashed line around **Worker D** indicates that this Worker is loaded dynamically, and is not statically-linked to the Worker that loads it (**Worker A**).

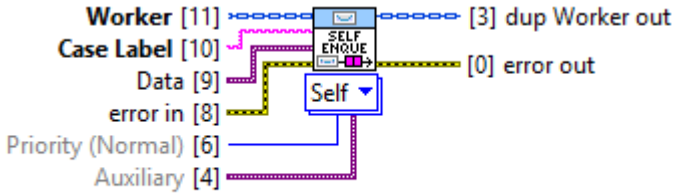
A Worker cannot run itself directly, and must be executed with a Launcher VI. **Worker A** sits at the top of the Worker call-chain hierarchy, and so we call it the 'Top-Most' Worker of the application. We can run **Worker A** by use of a Launcher VI (*Launcher – Worker A.vi*), shown below.



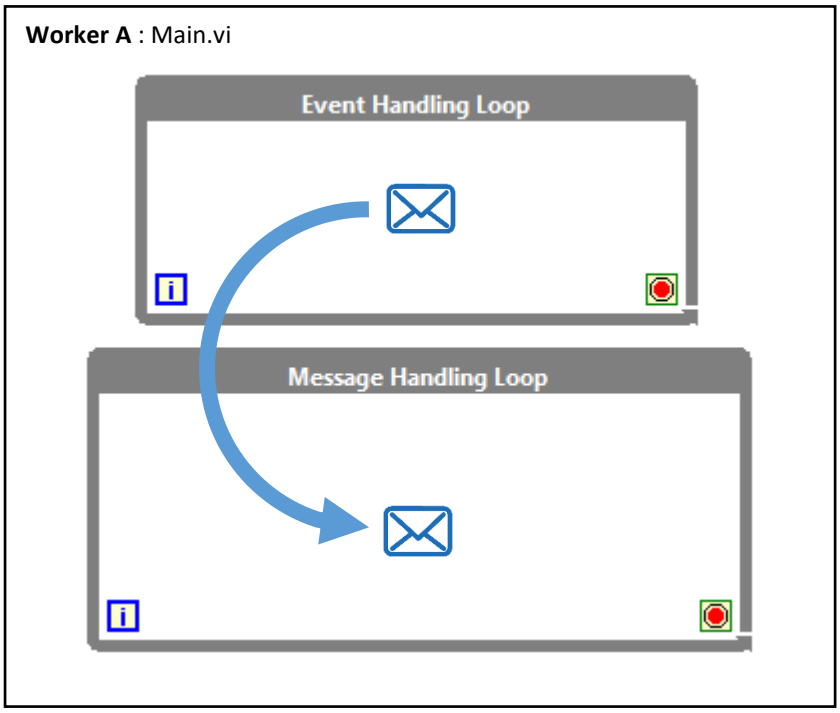
API

Step 1. Send Standard Message – Same Worker

This step demonstrates how to send a message from **Worker A** to itself, using the *Enqueue Message to Self.vi* (see context help for detailed information).



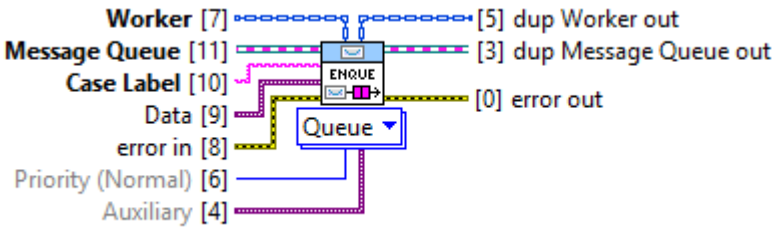
To demonstrate the use of this VI, we are sending a message from the EHL of **Worker A** to the MHL of **Worker A**, as shown below.



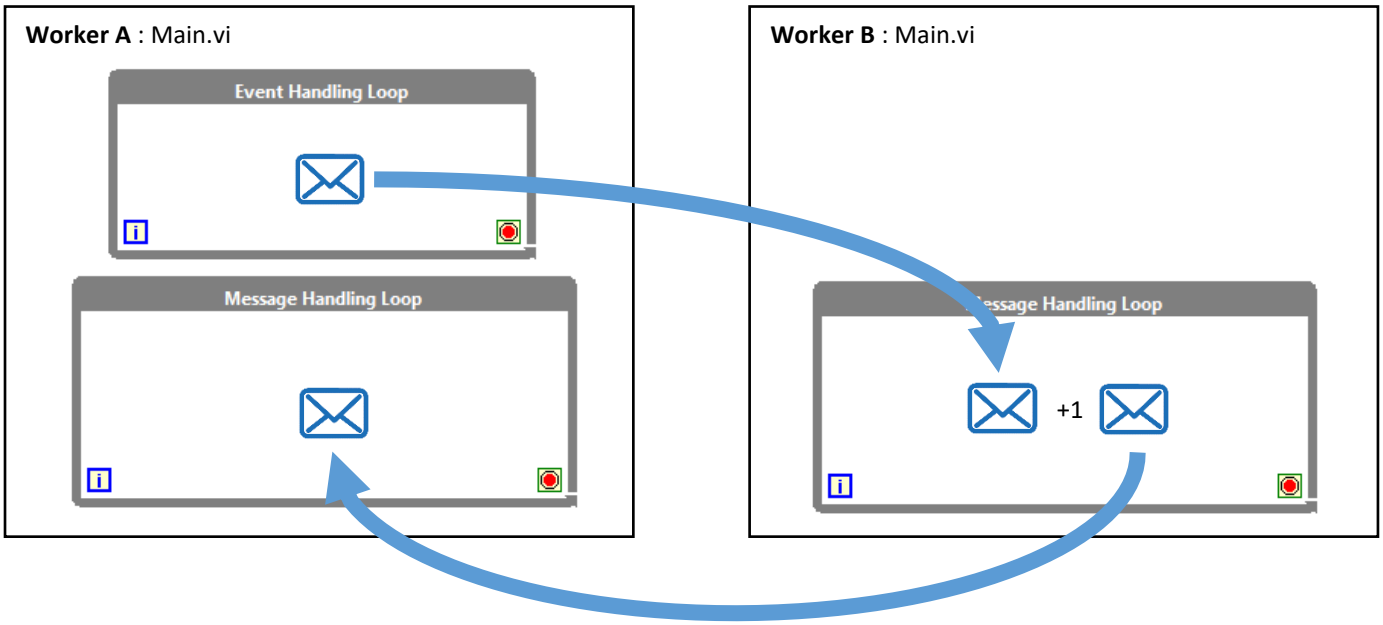
This VI can also be used to send messages from one case of the MHL to another case of the MHL in the same Worker.

Step 2. Send Standard Message – Different Workers

This step demonstrates how to send a message from **Worker A** to another Worker, using the *Enqueue Message to Queue.vi* (see context help for detailed information).



To demonstrate the use of this VI, we are sending a message from the EHL of **Worker A** to the MHL of **Worker B.**
Worker B increments the value in the message and then then sends the new value back to the MHL of **Worker A**, as shown below.

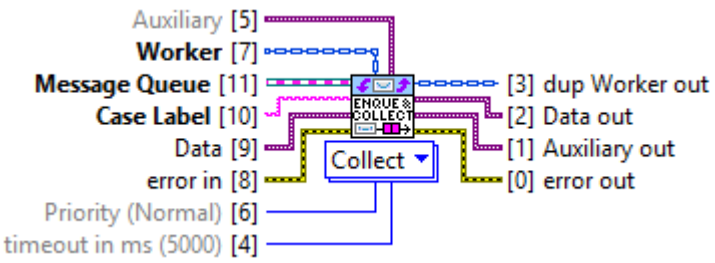


For **Worker B** to send the message back to **Worker A**, it must receive **Worker A**'s Queue as part of its Initialization Data. Investigate the block diagrams to see how this is achieved.

Step 3. Send Callback Message

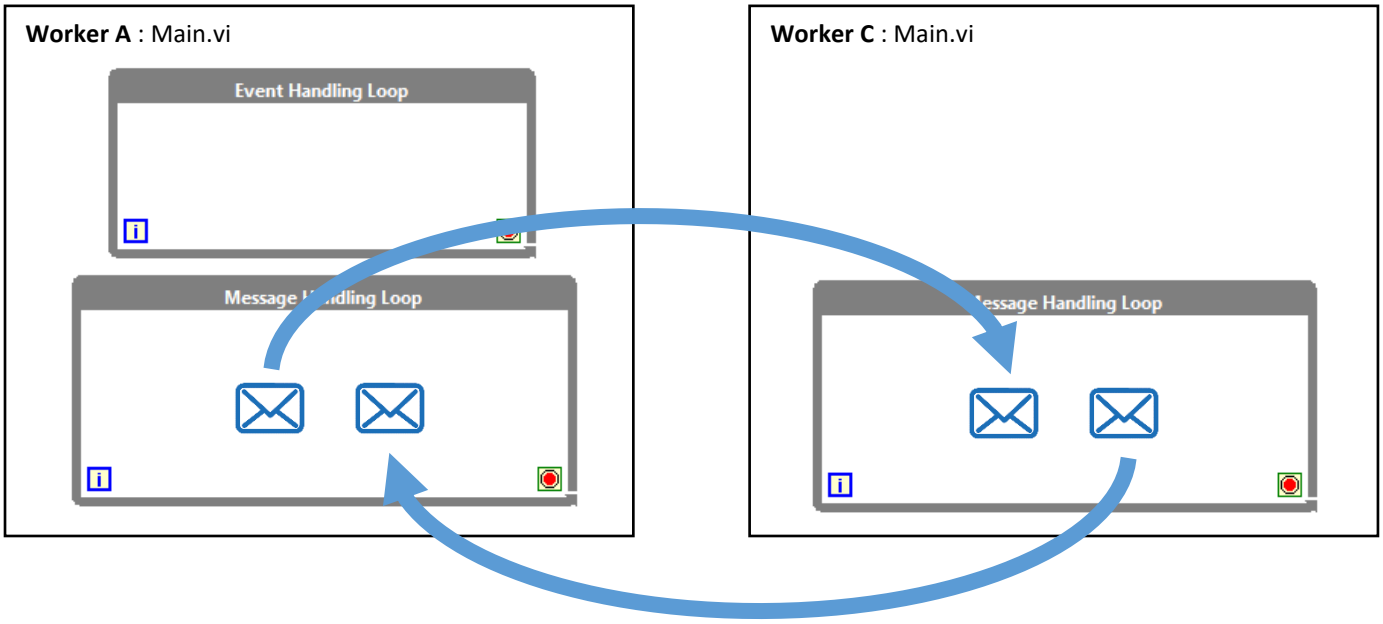
The difference between a Callback Message and a standard Message, is that a Callback Message contains the queue reference of the Worker that sends the message, so that the recipient Worker can send a reply message **back** to the Worker that sends the Callback Message, without explicitly needing access to the Worker's queue.

This step demonstrates how to send a message from **Worker A** to another Worker, using the *Enqueue and Collect.vi* (see context help for detailed information).



To demonstrate the use of this VI, we are sending a message from the MHL of **Worker A** to the MHL of **Worker C. Worker C** will then send a reply message back to **Worker A.**

However unlike in Step 2, we are not sending the reply message back to a different case of **Worker A** *and* **Worker C** doesn't need explicit access to the queue of **Worker A**. Instead the *Enqueue and Collect.vi* will wait until **Worker C** sends a 'Callback Reply' message. The process described above is shown below.

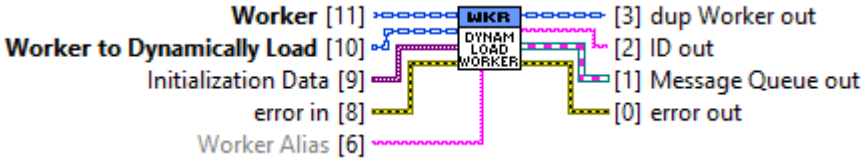


Unlike in Step 2, **Worker C** doesn't receive **Worker A**'s Queue as part of its Initialization Data. Investigate the block diagrams to see the differences.

Note: caution is advised when using this VI, since it blocks execution until it receives either a Callback Reply message or the VI times out. There is also a non-blocking version of this VI called *Enqueue and Forget.vi* which can be used in its place. See the context help for more information.

Step 4. Load Worker Dynamically

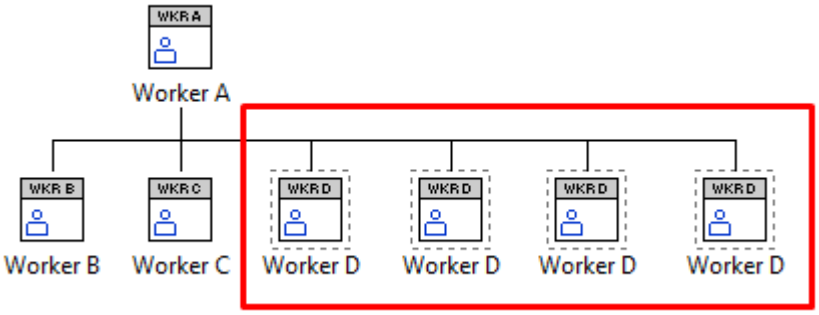
This step demonstrates how to load a Worker dynamically, on demand, using the *Dynamically Load Worker.vi* (see context help for detailed information).



To demonstrate the use of this VI, we will load **Worker D** from the MHL of **Worker A**, and initialize **Worker D** with a color and string, defined by the user.

When an instance of **Worker D** is loaded, it will be connected to **Worker A** the same way that the statically-linked Workers (**Worker B** and **Worker C**) are.

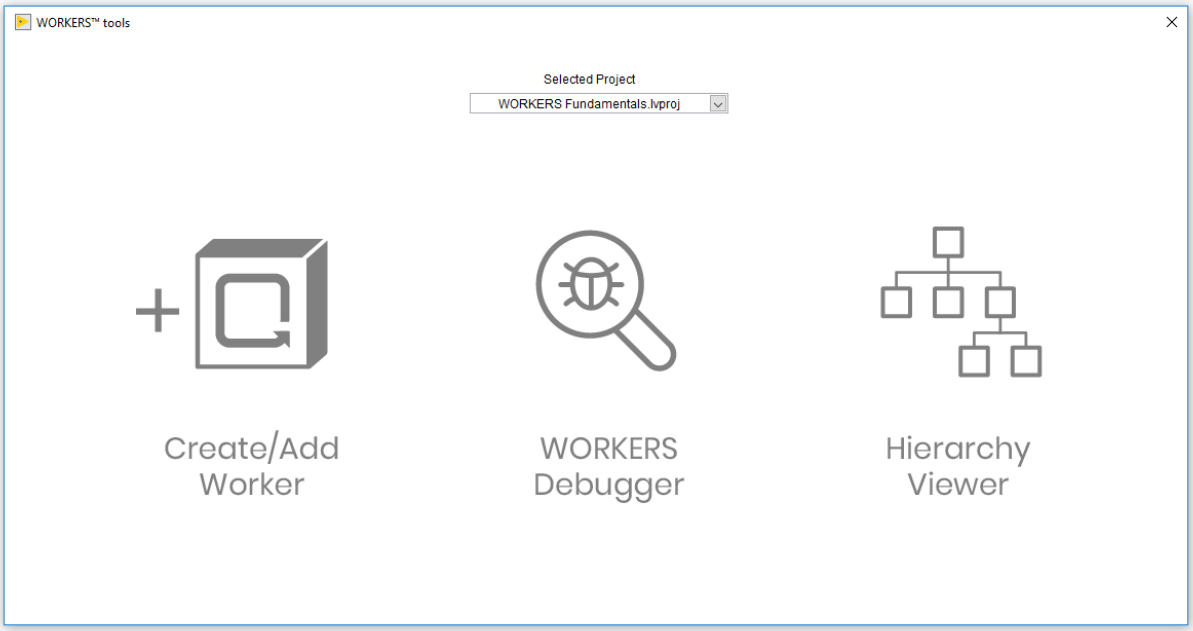
You can load as many instances of **Worker D** as you like and they will be automatically integrated into your application, as shown in the diagram below.



Dynamically loaded instances of **Worker D**

Workers™ tools

The *Workers™ tools* window (shown below) can be accessed through the LabVIEW 'Tools' menu.



Create/Add Worker

The Create/Add Worker tool allows you to quickly build up the frame of your application (an application's Worker call-chain hierarchy) by instantly adding new Workers to your application whenever and wherever you need them.

Workers™ Debugger

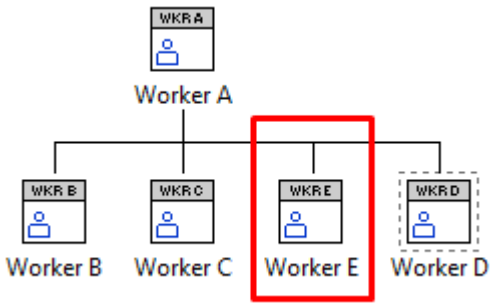
The Workers™ Debugger allows you to see chronological the flow of messages sent in your application. I.e. the location of where a message is enqueued, and where it is dequeued.

Hierarchy Viewer

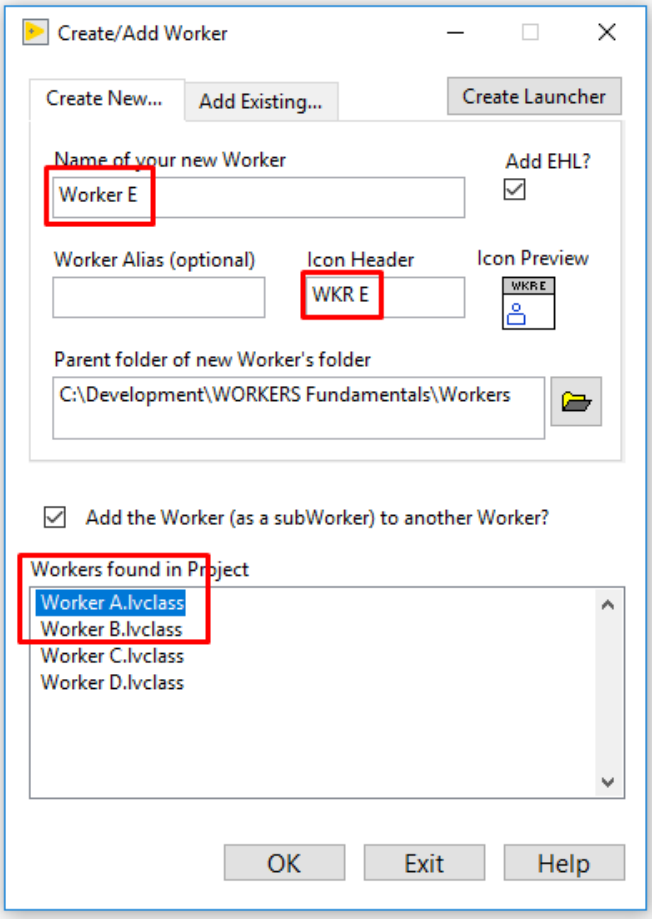
This tool shows you the Worker call-chain hierarchy of all the Workers in an application. Both statically-linked and dynamically loaded Workers are shown.

Step 5. Add a new Worker to your application

In this step, we are going to create and add a new Worker to your application (let's call it **Worker E**) to make your Worker call-chain hierarchy look like:

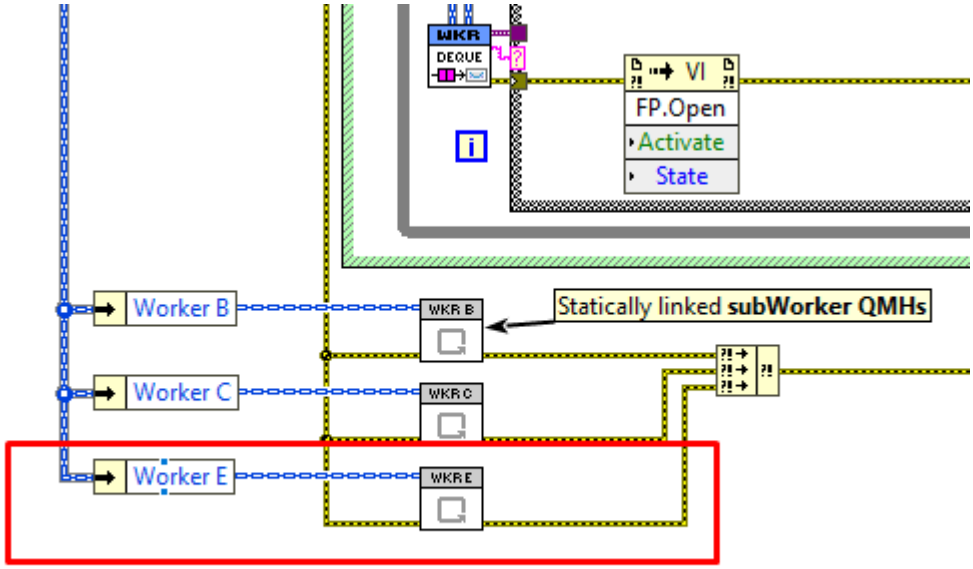


To do this, open up the *Workers™ tools* window and select "Create/Add Worker". The following window will appear. Fill in the elements as shown below, and then press OK.



After the tool completes, close the tool. (Note: this process should take between 8 to 30 seconds, depending on your CPU performance and LabVIEW version.)

Take a look at **Worker A**'s *Main.vi*. On its block diagram, you should notice that **Worker E** has been added to it, as shown below.



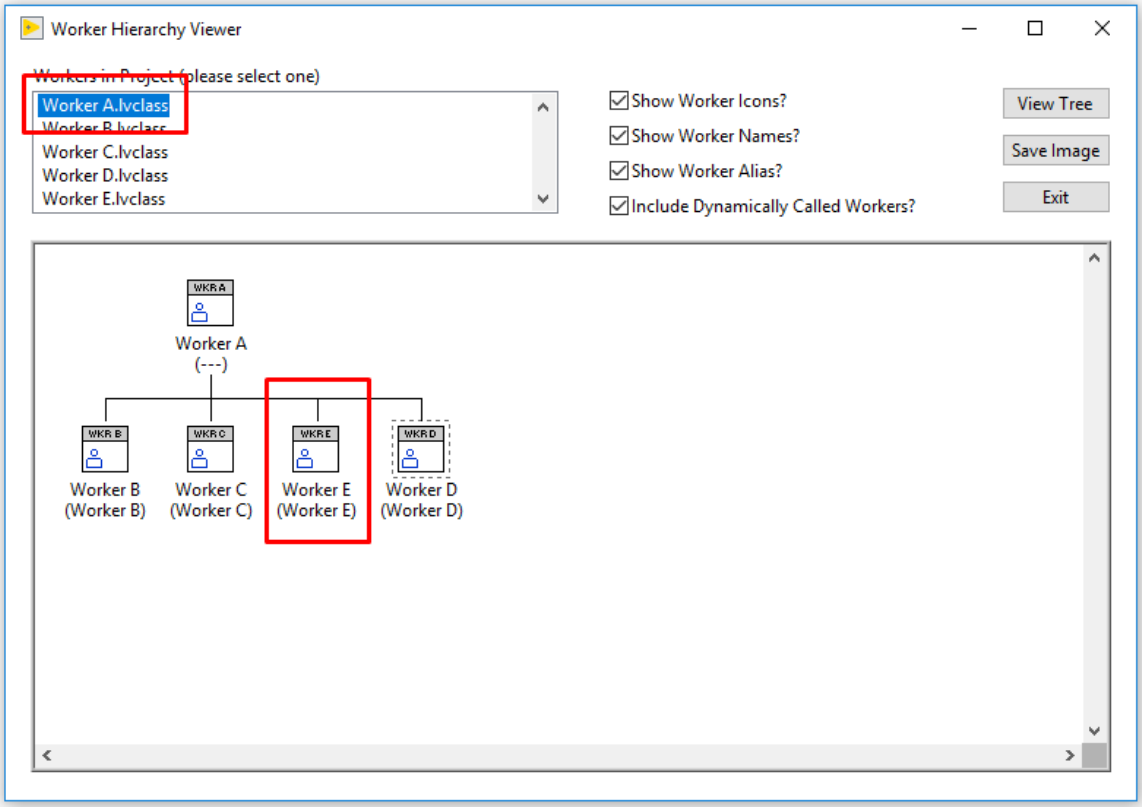
Assuming you had no errors in your application before you added **Worker E** to it, you should be able to run your application now with **Worker E** included as well.

Worker E is now integrated into your application but doesn't actively do anything, apart from being initialized and shutdown with the rest of your application. You will also see it appear in the Workers™ Debugger when you run the application.

Feel free to add any other cases, send messages to it, etc., as you like.

Step 6. View your application's Worker call-chain Hierarchy

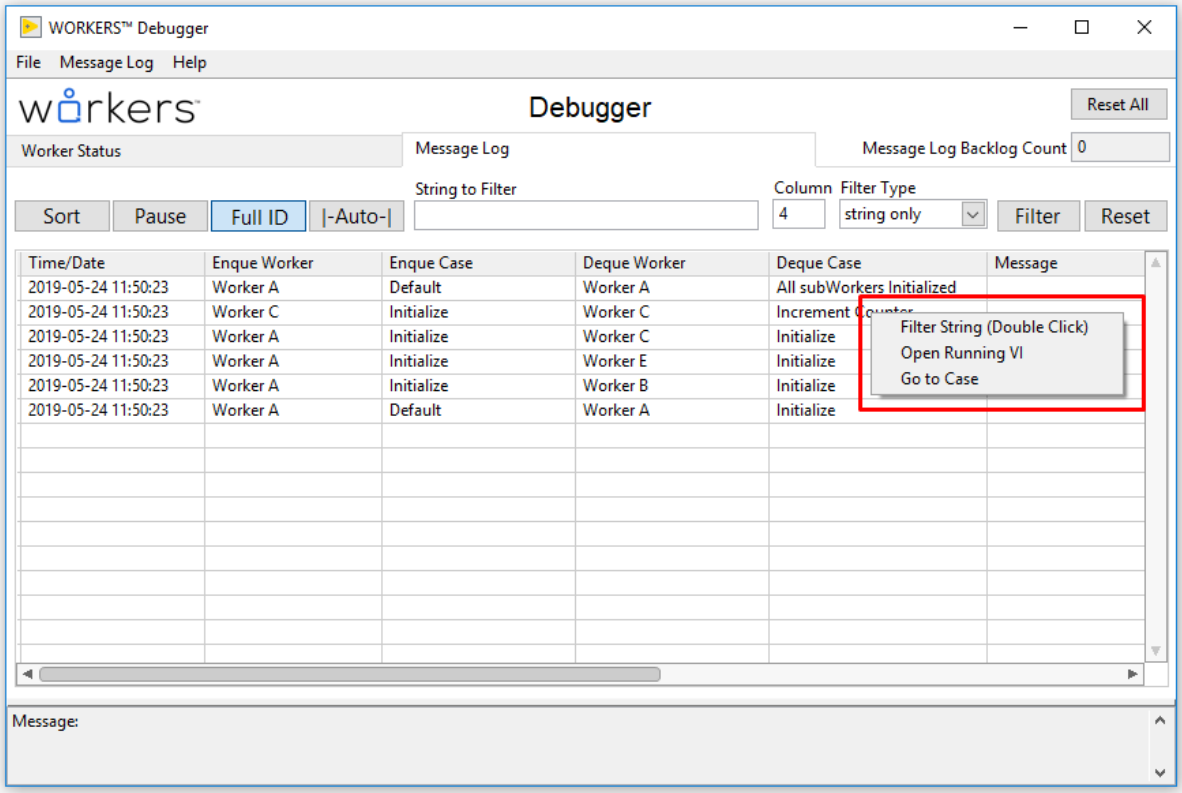
In this step, we are going to view the Worker call-chain Hierarchy of your application. To do this, open up the *Workers™ tools* window and select "Hierarchy Viewer". The following window will appear. Select **Worker A.**lvclass and press 'View Tree'.



The Worker call-chain hierarchy of your application should appear, now with the inclusion of **Worker E**.

Step 7. Workers™ Debugger

When you run your application, the Workers™ Debugger window should automatically appear. Select the Message Log tab, as shown below.



Right-click with the mouse over one of the cells in the 'Deque Case' column. A pop-up menu will appear. With this menu, you can jump directly to the selected case of a particular Worker, or even jump directly to the running instance of that Worker. Try this.

For further details, the 'Help' menu in the Debugger window should take to you to relevant section in the *Workers™ User Guide.pdf*.

Step 8. Continue at your own pace…

Now that you are empowered with what you have learnt from the previous steps, you can easily add additional Workers to your application, create new cases, send new messages, etc. Also please take a look at the comments written all over the block diagrams of the VIs in this sample project. They will help you understand the various features of the framework. Also make sure you take note of the Quickdrop shortcuts that are provided. They are described on **Worker A**'s *Main.vi*. They contain some great features designed to help you be more productive.

To see the framework in action, please take a look at the Workers™ version of the well-known 'Continuous Measurement and Logging' sample project that ships with LabVIEW. We created this so that you can compare side-by-side the classic LabVIEW version of this project with the Workers™ version of this project, to make obvious the similarities and differences between the two methods.

For further details, please refer to the *Workers™ User Guide.pdf*.

Good luck and happy programming. ☺