



USER GUIDE

June 2020



**Copyright 2020, Scarfe Controls.
All Rights Reserved.**

Disclaimer

The information in this document is subject to change without prior notice in order to improve reliability, design, and function and does not represent a commitment on the part of the author. In no event will the author be liable for direct, indirect, special, incidental, or consequential damages arising out of the use or inability to use the product or documentation, even if advised of the possibility of such damages.

This document contains proprietary information protected by copyright. All rights are reserved. No part of this manual may be reproduced by any mechanical, electronic, or other means in any form without prior written permission of the author.

CONTENTS

Preface	iv
Nomenclature.....	v
1 Introduction	7
1.1 Create/Add Worker tool	7
1.2 Workers Debugger	7
1.3 Priority Queue.....	8
1.4 Case Labels.....	8
1.5 Worker Hierarchy Viewer	8
2 The Worker	9
2.1 Worker Relationship Terms	10
2.2 Framework Required Cases (Common to all Workers).....	11
2.3 Framework Required Cases (Common only to Managers)	12
3 Initialization / Shutdown Sequences.....	14
3.1 Initialization Sequence.....	14
3.2 Shutdown Sequence	15
4 Messaging	17
4.1 Regular (asynchronous) Messaging	17
4.2 Callback Messaging	17
4.3 Message Contents.....	18
4.4 Message Priorities.....	19
4.5 Case Labels.....	19
5 Statically Linked / Dynamically Loading Workers	21
5.1 Statically Linked Workers.....	21
5.2 Dynamically Loading Workers.....	21
6 Quickdrop shortcuts	23
6.1 Show Worker's Private Data (Ctrl + 0)	23
6.2 Create Case Label (Ctrl + 9)	23
6.3 Show case of selected Case Label (Ctrl + 8)	24

7	Debugger.....	25
7.1	Task Manager Tab	25
7.2	Message Log Tab.....	27
8	Creating/Adding/Removing Workers.....	30
8.1	‘Create/Add Worker’ tool (recommended way).....	30
8.2	Manually Adding a Worker as a subWorker	33
8.3	Removing a subWorker from its Manager.....	40
9	Launcher VIs.....	41
10	Worker Hierarchy Viewer	42
11	Creating a New Application	43
11.1	Starting from a Blank Project	43
11.2	Sample Projects.....	43
12	Additional Features	44
12.1	Automatic Error Logging	44
12.2	Application Global Attributes.....	45
12.3	Custom Worker Statuses in the Debugger.....	46
12.4	Creating your own Worker Templates.....	46
12.5	Using your own Worker Templates	47
13	Advanced Users (LVOOP).....	48
13.1	Class inheritance of Workers	48
13.2	Worker User Class	49
13.3	Priority Queue.....	49
14	Appendix A	50
14.1	Error Codes	50

PREFACE

LabVIEW has been around for over 30 years now and over time has become an industry standard for the quick and easy acquisition, visualization, logging and control of data from real world hardware. Yet while small and individual tasks can be completed in a very time efficient manner, the development of more complex multi-process applications in LabVIEW remains a difficult and time-consuming process. Applications that require several independent loops, are robust, scalable and easy to debug take time to develop. The development of such an application is an extremely difficult task for a novice LabVIEW developer and can still take considerable amount of time for advanced developers. Features that allow scalability and code reuse and debugging features need to be built directly into quality code from the very beginning, but often there is not time for this to be done properly in projects that have tight deadlines to meet.

While LabVIEW provides single-use design pattern templates, the Actor Framework provides enhanced scalability and priority queues through the use of LVOOP, and third party vendors provide additional solutions, I still felt that a lot more could be done to produce applications quickly and easily in LabVIEW in a minimalistic way without requiring developers to learn Object-Oriented Programming, a new design pattern, or an unfamiliar framework architecture that deviates from the standard LabVIEW training courses.

The backbone of many multi-process LabVIEW applications uses a Queued Message Handler (QMH) based architecture whereby applications are built up using a hierarchy of QMHs which are either statically linked to or loaded dynamically by their caller. While this is one of the most widely used, understood and accepted way of developing applications in LabVIEW, it is a slow and inefficient process to create, add and integrate many QMHs together manually. To help solve this problem, Workers™ was created.

The goal of Workers™ was to provide a scalable QMH replacement of the LabVIEW Queued Message Handler template. It needed to be as easy to use as the QMH template and provide a quick and simple way of building a functional hierarchy (application) out of many QMHs.

Each QMH had to be as minimalistic as possible, providing only the basic yet fundamental features of a QMH: namely a QMH itself and a data structure to define its customizable initialization variables. Code duplication of VIs that perform the same function in every QMH was avoided at all costs. Any additional features that could be integrated to improve a developers coding efficiency, flexibility and data flow tracking, without bloating the basic LabVIEW QMH coding style, were added.

And thus Workers™ was born, the name simply implying that an application can be built out of a hierarchy of QMHs, with each QMH assigned a single process.

I hope that Workers™ will dramatically increase your productivity when developing new and future applications in LabVIEW and that you enjoy the time developing applications with the framework and integrated tools too.

Peter Scarfe, Creator of Workers™ for LabVIEW.

NOMENCLATURE

EHL

Event Handling Loop. A loop of a Worker whose cases are called by registered events.

Head Worker

Functionally identical to any other Worker, this is the name for the Worker that sits at the very top of the Worker call-chain hierarchy. It can exist alone or call additional subWorkers but requires a Launcher VI to run. It is always the first Worker that will run and the last Worker that will exit in a Workers™ application.

Launcher VI

An external VI that you run, to launch the Head Worker of an application (see Chapter 9).

Main Data Wire

The blue-colored data wire that runs through every Worker.

Manager

A Worker that calls one-or-more subWorkers is known as the subWorker's Manager.

MHL

Message Handling Loop. The loop of a Worker whose cases are called by dequeuing priority messages from a dedicated message queue.

QMH

Queued Message Handler. A common design pattern used in the creation of LabVIEW applications, consisting of a loop that can communicate with other QMHs another via the use of message queues.

Silent Mode (in the context of enqueueing a message)

When silent mode is enabled, a record of the message is not sent to the Debugger, and thus the message will not appear in the Debugger's Message Log.

subWorker

A Worker that is called by another Worker is known as its subWorker.

Worker

Simple definition: A modular Queued Message Handler.

Advanced definition: A Queued Message Handler and its supporting code, encapsulated in a class (that is inherited from `Worker.lvclass`).

Worker Hierarchy

The Worker call-chain hierarchy, often referred to as simply the ‘Worker Hierarchy’, showing the call chain of all the Workers in an application.

1 INTRODUCTION

IMPORTANT: Workers™ is a framework based on the well-known LabVIEW Queued Message Handler (QMH). It is assumed that you are familiar with the QMH before you further proceed with this document.

Workers™ provides LabVIEW developers with a modular Queued Message Handler based framework API, along with support tools that are designed for the **quick and easy development of scalable QMH based applications**, both small and large in size, by novice (LabVIEW Core 3 level), advanced, and/or teams of LabVIEW developers.

This document will provide detailed information on the many aspects of the framework and the tools that have been created to maximize the efficiency of your code development when using the framework. The main features that Workers™ provides to the developer include the following:

1.1 CREATE/ADD WORKER TOOL

By far this is the most time saving feature that Workers™ provides to the developer. No longer is it necessary to manually create QMHs from a template and then manually integrate the new QMH into your existing application. With the 'Create/Add Worker' tool, it is possible to completely **add and integrate** a new QMH directly into your application in a matter of seconds. This means that you can create a large LabVIEW application consisting of many QMHs in a matter of minutes. And not only can you do this for the initial architecting of your application, but you can continue to add additional QMHs to your application whenever you need them at any time in the future. This tool will be further discussed in Section 8.

1.2 WORKERS DEBUGGER

The Workers Debugger provides a vital piece of information to the developer that LabVIEW does not provide natively. This piece of information is: **"The flow of messages between QMHs."**

The debugger provides a list of not only which QMH a message was enqueued and dequeued in, but also in which **case** of the QMH the message was, providing point-to-point tracking of messages sent between QMHs. Through the debugger it is also possible to jump directly to the case where a message was enqueued or dequeued.

The debugger also allows you to see the status of each QMH in your application, and the position of a QMH in the application's QMH call-chain hierarchy. In addition, the debugger allows the developer to jump directly to the running instance of a QMH in memory... which is useful since each Worker QMH is a clone, giving you full debugging access to every QMH in your entire Workers™ application.

The complete set of features of the Debugger will be discussed in Section 7.

1.3 PRIORITY QUEUE

Workers™ has adopted the Priority Queue of the LabVIEW Actor Framework for the sending of messages between QMHs in the framework, providing additional messaging flexibility over the standard FIFO LabVIEW queues. Three priorities are available to developers: Low, Normal and High. More about the use of the Priority Queue is discussed in Section 4.4 and Section 13.3

1.4 CASE LABELS

Case Labels provide the developer with a **single location** to define a QMH's MHL case as a string, and an easy way to create them. Normally when sending a message to a case in a QMH the developer would define the case by using a string constant. There are many disadvantages to this approach and therefore the concept of the 'Case Label' was created. Figure 1 illustrates this.

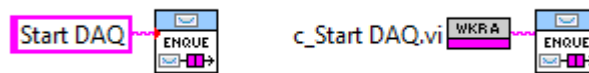


Figure 1. String constant (left) vs 'Case Label' (right) to enqueue a message to a QMH's MHL case 'Start DAQ'

Case Labels are easy to create for any given case that exists in any Worker's QMH by use of a quickdrop shortcut and are automatically added to your Worker.

Another useful feature of the Case Label is the ability **to jump directly to the case that a Case Label defines**, again by use of another Quickdrop shortcut. This feature allows the developer to follow the flow of messages when developing, while the Debugger allows the developer to follow the flow of messages during run-time. Case Labels will be further discussed in Section 4.5.

1.5 WORKER HIERARCHY VIEWER

The Worker Hierarchy Viewer shows the call-chain of all the QMHs in an application and whether a QMH is either statically linked to or loaded dynamically by its caller, providing a high-level visualization of the structure of an application. In Figure 2, the QMH hierarchy is shown of an application that consists of three QMHs: **Worker A**, **Worker B** and **Worker C**.

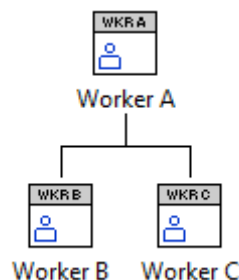


Figure 2. An application's QMH call-chain Hierarchy

Worker B and **Worker C** are called via static links on **Worker A**'s block diagram, in the same way as the QMHs are called in the 'Continuous Measurement and Logging' sample project that ships with LabVIEW. More about this tool will be discussed in Section 9.

2 THE WORKER

IMPORTANT: A Worker is simply a superset of a QMH and its associated parts. For this reason, the term *QMH* as used in the Introduction of this document, will be now referred to as a *Worker* (unless specifically referring to the QMH part of a Worker).

A Worker is defined as:

Simple definition: A modular Queued Message Handler

Advanced definition: A Queued Message Handler and its supporting code, encapsulated in a class (that is inherited from *Worker.lvclass*).

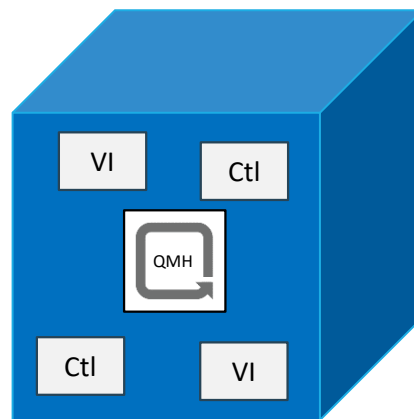


Figure 3. Visual representation of a Worker, showing a class (the blue cube) that contains a QMH and its supporting VIs and controls

Every Worker, as seen in the LabVIEW Project Explorer, contains only these common items:

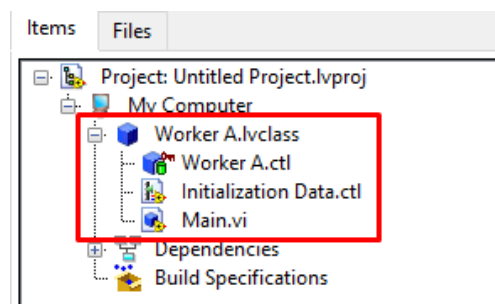


Figure 4. Default parts of a Worker

- (Worker's name).lvclass

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

- **(Worker's 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 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.

TIP: Each *Main.vi* is created as a shared clone, and thus is designed to be reusable. Keep this in mind when you develop your Workers. You can create a Worker to be loaded more than once in a single application or re-use it between projects. Each Worker should be designed to possess certain skills, yet it is the task of the Worker's Manager (see Section 2.1) to decide how the Worker behaves. The dynamically loaded Worker in the Workers™ Sample Application Project (section 11) is a good example of this.

2.1 WORKER RELATIONSHIP TERMS

A typical Workers™ application is built up out of a hierarchy of Workers, whereby one Worker will call one-or-more additional Workers from its *Main.vi*'s block diagram. Let's take an example where a Workers™ application is built from three Workers: **Worker A**, **Worker B**, and **Worker C** as expressed in Figure 5.

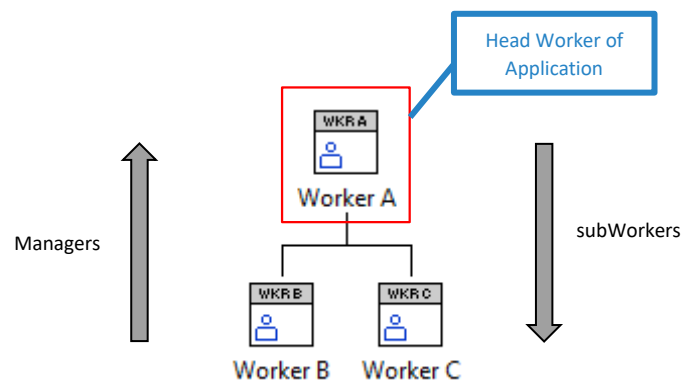


Figure 5. Worker Hierarchy diagram of Figure 6, with relationship terms

Worker A is the first Worker at the very top of the Worker Hierarchy and is therefore known as the 'Head Worker' of the application. Its *Main.vi* is the VI which you launch (by a Launcher VI) to run the application.

Worker B and **Worker C** are called from **Worker A**'s *Main.vi*, and therefore are 'subWorkers' of **Worker A**.

Since **Worker B** and **Worker C** are subWorkers of **Worker A**, we say that **Worker A** is the ‘Manager’ of **Worker B** and **Worker C**, since **Worker A** manages the initialization and shut down of its subWorkers.

These two terms ‘subWorker’ and ‘Manager’ are used to describe the callee/caller relationship of Workers respectively in the Worker call-chain hierarchy and are used throughout the remainder of this document.

Figure 5 is simply a representation of the LabVIEW code presented in Figure 6, showing how **Worker B’s Main.vi** and **Worker C’s Main.vi** are called via static links from **Worker A’s Main.vi**.

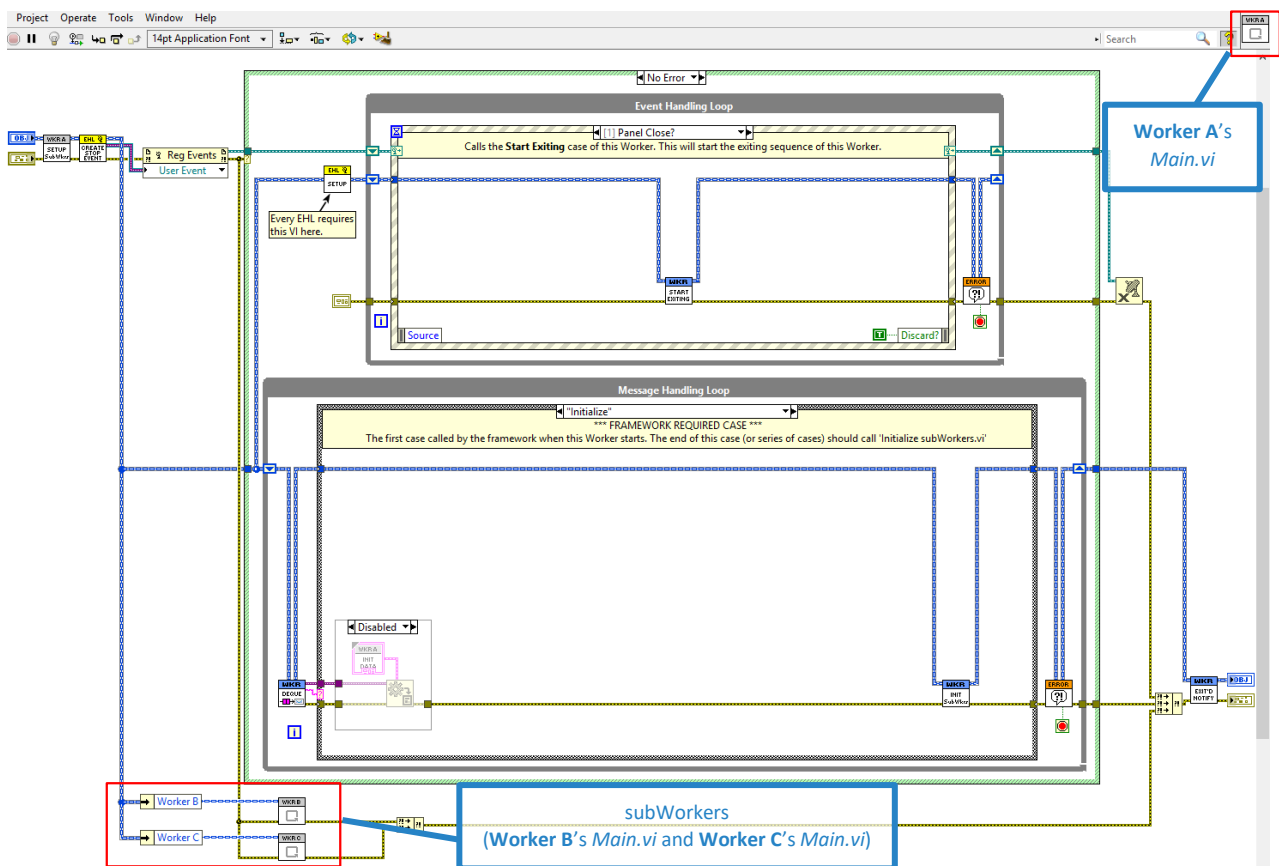


Figure 6. **Worker A’s Main.vi**

2.2 FRAMEWORK REQUIRED CASES (COMMON TO ALL WORKERS)

Each Worker is required to have these cases for successful operation of the framework. They are created for you automatically when you create a new Worker with the ‘Create/Add Worker’ tool (see section 8.1).

2.2.1 Default

The default case of the Message Handling Loop (MHL) in a Worker’s *Main.vi* contains the framework cases within a single VI. These cases are required for the operation of the framework and thus the VI in this case must not be removed.

2.2.2 Initialize

The 'Initialize' case of the MHL is the first case that is called when you run your application. This case is called internally by the framework and can receive an *Initialization Data.ctf* data cluster from a Worker's Manager.

Worker with subWorkers

In this case you can perform any tasks that are required to be completed by the Worker **before** the 'Initialize' case of its statically linked subWorkers is called.

Part of a Worker's initialization responsibility is to send initialization data to its subWorkers via their *Initialization Data.ctf*, if they require it. You can use the *Write subWorker Initialization Data.vi* to load into a buffer all the Initialization Data clusters of each subWorker.

Finally, you are required to call *Initialize subWorkers.vi*. Calling this VI will then call the 'Initialize' case of all statically linked subWorkers, and if any initialization data has been buffered, it will be sent automatically with the message to the 'Initialize' cases of the subWorkers.

Worker without subWorkers

After you have performed any initialization tasks in this case, you are required to call *Initialized Notify.vi*. This VI will notify the Worker's Manager that the Worker has been successfully initialized.

2.2.3 Start Exiting

The 'Start Exiting' case of the MHL is called when the application wants to shut down. Using the *Start Exiting Worker.vi* will call this case directly, or this case will be called internally by the framework by a Worker's Manager.

Worker with subWorkers

In this case you can perform any tasks that are required to be performed **before** the 'Start Exiting' case of the subWorkers is called.

When you are ready to call the 'Start Exiting' case of the subWorkers, *Start Exiting subWorkers.vi* can be called. This will cause the 'Start Exiting' case of all the Worker's subWorkers (that are currently running) to be called.

Worker without subWorkers

After you have performed any shut down tasks in this case, you are required to call *Exit.vi*. This VI will cause the MHL (and EHL if the Worker has one) to stop, and the Worker will then exit.

2.3 FRAMEWORK REQUIRED CASES (COMMON ONLY TO MANAGERS)

When a Worker is added to another Worker by use of the 'Create/Add Worker' tool, two more cases are automatically added to your Worker, now known as the Manager of its subWorkers.

The tool does the following things:

1. Creates the cases 'All subWorkers Initialized' and 'All subWorkers Exited'
2. Replaces *Initialized Notify.vi* with *Initialize subWorkers.vi*.
3. Replaces *Exit.vi* with *Start Exiting subWorkers.vi*.

IMPORTANT: Always leave the *Initialized Notify.vi* and *Exit.vi* VIs on the block diagram of your Worker's *Main.vi*, and do not embed them within a subVI. The reason for this is that the 'Create/Add Worker' tool looks for these VIs and replaces them automatically with the correct VIs. This cannot be done if these VIs do not exist on the *Main.vi*'s block diagram and will result in a warning when using the tool.

2.3.1 All subWorkers Initialized (Common only to Managers)

This case is called by the framework when all a Worker's statically linked subWorkers have been successfully initialized. (A Worker will notify its Manager that it has been successfully initialized when its *Initialized Notify.vi* has been called). You can now perform any tasks that are required to be performed now that all the Worker's subWorkers have been initialized.

When you are ready you are then required to call *Initialized Notify.vi*. This VI will notify the Worker's Manager that the Worker has been successfully initialized.

2.3.2 All subWorkers Exited (Common only to Managers)

This case is called by the framework when all a Worker's subWorkers have been successfully exited. (A Worker will notify its Manager that it has exited when its *Exited Notify and Cleanup.vi* has been called.) You can now perform any tasks that are required to be performed now that all the Worker's subWorkers have been exited.

When you are ready to exit the Worker, you are required to call *Exit.vi*. This VI will cause the MHL (and EHL if the Worker has one) to stop, and the Worker will then exit.

3 INITIALIZATION / SHUTDOWN SEQUENCES

In the previous section, each of the ‘Framework Required Cases’ were described along with the VIs that they are required to call. This section will illustrate this process for you to better understand how the framework handles the initialization and shutdown of all its Workers.

The framework’s initialize and shutdown sequences are performed by a cascade call both down and up the Worker hierarchy. To best explain this, let’s look at an example. Take an application of four Workers, arranged in a call-chain hierarchy as illustrated in Figure 7. The initialization and shutdown sequences of the Workers in this application will be discussed now.

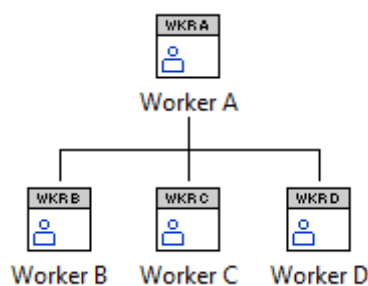


Figure 7. Worker Hierarchy of an application

3.1 INITIALIZATION SEQUENCE

The initialization of an application always starts with the Head Worker’s ‘Initialize’ case being called first. In Figure 7 this means that **Worker A**’s ‘Initialize’ case is the first case that is called in the entire application. The sequence of events that occurs from here onwards is as follows and is illustrated in Figure 8.

0. Launcher VI (section 9) launches **Worker A**.
1. **Worker A**’s ‘Initialize’ case is called. Then by calling *Initialize subWorkers.vi*...
2. ...all of **Worker A**’s subWorkers’ ‘Initialize’ cases are called. *Initialized Notify.vi* is called in each subWorker, telling **Worker A** that each has been initialized. When all of **Worker A**’s subWorkers have reported they have been initialized...
3. ...**Worker A**’s ‘All subWorkers Initialized’ case is called. Finally, **Worker A** calls *Initialized Notify.vi* to tell the framework that it too has been successfully initialized.

Note that if **Worker A** was not the Head Worker, then calling *Initialized Notify.vi* would alert its Manager that **Worker A** had been initialized, and this would continue all the way up to the Head Worker of the application.

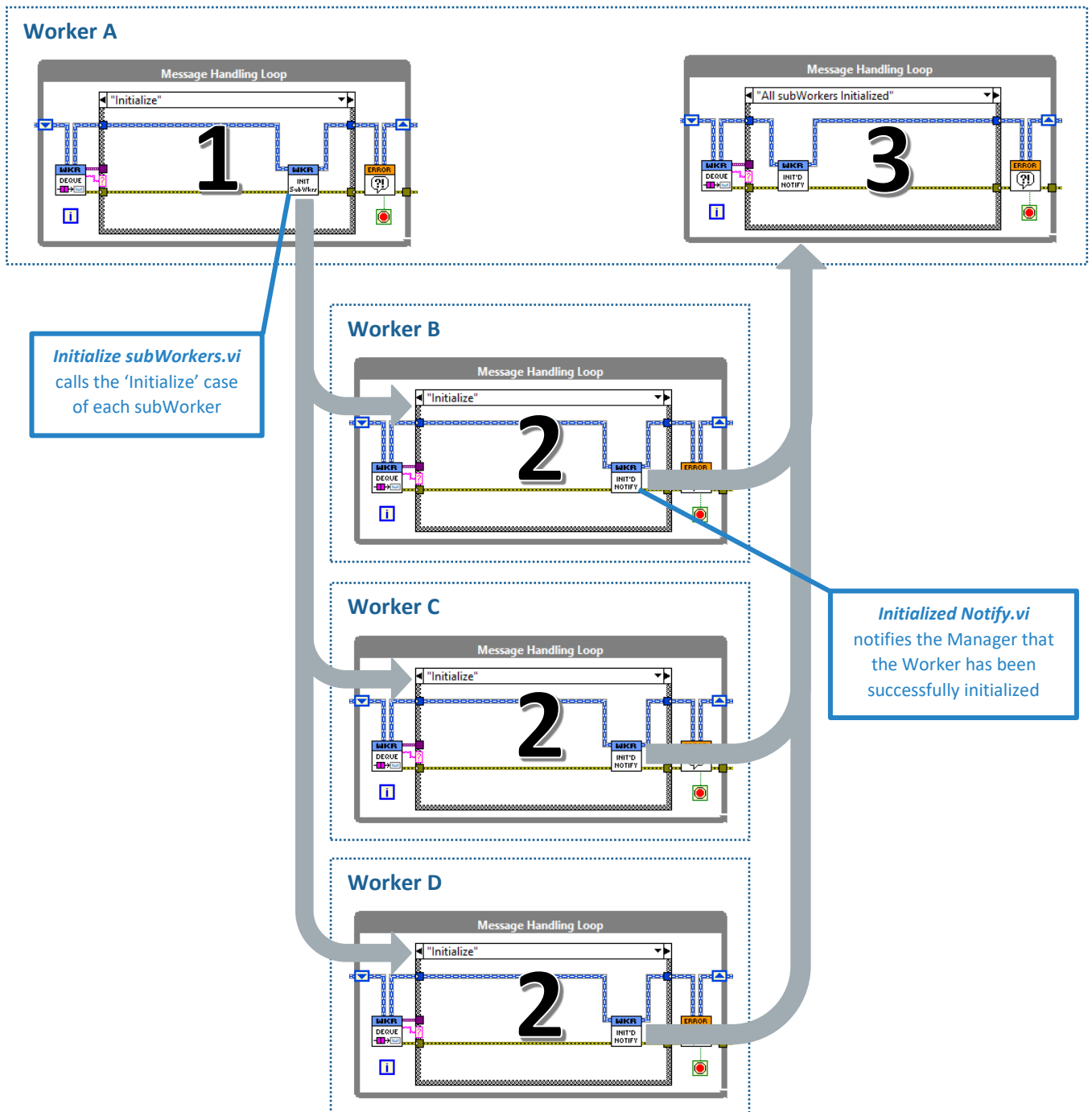


Figure 8. Initialization sequence of Workers in the application of Figure 7.

3.2 SHUTDOWN SEQUENCE

The shutdown of an application always starts with the Head Worker's 'Start Exiting' case being called first. In our example hierarchy, this means that **Worker A's** 'Start Exiting' case is the first case that is called when the application is required to shut down. The sequence of events that occurs from here onwards is as follows and is illustrated in Figure 9.

0. User calls *Start Exiting Worker.vi* (available in the Workers™ palette).
1. **Worker A's** 'Start Exiting' case is called. Then by calling *Start Exiting subWorkers.vi...*

2. ...all of **Worker A**'s subWorkers' 'Start Exiting' cases are called. Finally *Exited Notify and Cleanup.vi* is called by each subWorker, telling **Worker A** that each has exited. When all of **Worker A**'s subWorkers have reported they have exited...
3. ...**Worker A**'s 'All subWorkers Exited' case is called. Finally, **Worker A** calls *Exited Notify and Cleanup.vi* to tell the framework that it too has successfully exited.

Note that if **Worker A** was not the Head Worker, then calling *Exited Notify and Cleanup.vi* would alert its Manager that **Worker A** has exited, and this would continue all the way up to the Head Worker of the application.

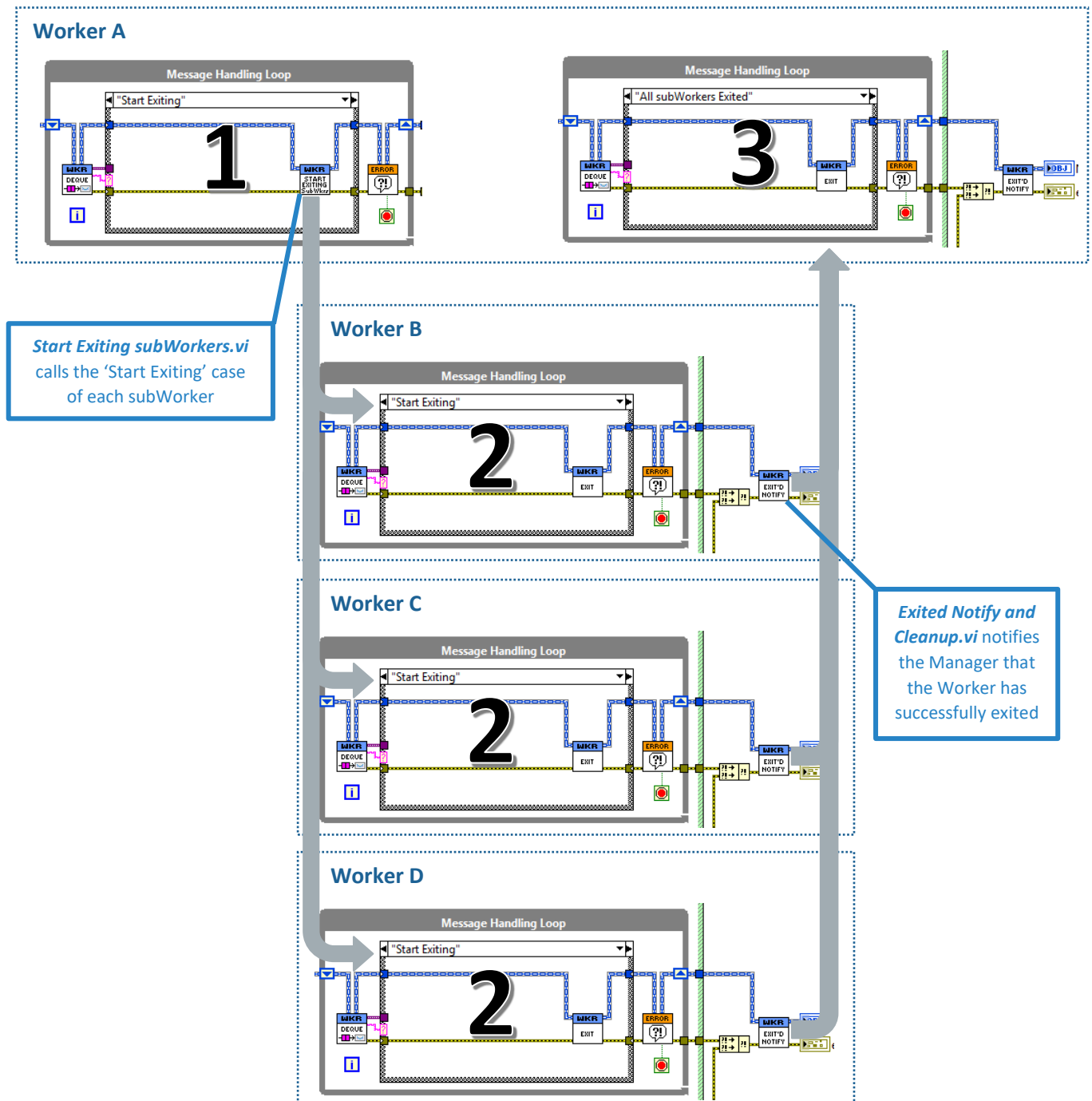


Figure 9. Shut down sequence of Workers in the application of Figure 7.

4 MESSAGING

The flow of data in a QMH based application is achieved by the sending of messages between the Message Handling Loops of numerous QMHs. This is achieved by each QMH being assigned a message queue, whereby messages are enqueued in one QMH and dequeued by another.

There are two types of default messaging systems that Workers™ employs. A regular message (asynchronous) and a callback message (asynchronous or synchronous).

Note: the point of the descriptions here are not to provide comprehensive descriptions of the VIs used in the framework (see individual VI context help for this) but instead to explain the operations behind the framework's messaging API.

4.1 REGULAR (ASYNCHRONOUS) MESSAGING

A regular asynchronous message is the preferred way to send messages between Workers. This is achieved by using the *_Enqueue Message.vi*.

4.1.1 *_Enqueue Message.vi*

This VI is a polymorphic VI, with the possible selection of the following VIs:

[Enqueue Message to Self.vi](#)

Enqueues an asynchronous message to the Worker that this VI is called from.

[Enqueue Message to Queue.vi](#)

Enqueues an asynchronous message to a Worker, defined by its Queue.

4.2 CALLBACK MESSAGING

The difference between a callback message and a regular message is that a callback message contains additional information about the Worker that sent the message. This way, it is possible for the recipient Worker to send a reply message directly back to Worker that sent the Callback Message, without explicitly needing access to its Queue.

There are two types of callback messages that exist in the framework. They are accessed by the use of *Enqueue Callback Message.vi*.

4.2.1 *Enqueue Callback Message.vi*

This VI is a polymorphic VI, with the possible selection of the following VIs:

[Enqueue and Collect.vi](#)

Enqueues a synchronous message to a Worker, defined by its Queue. This VI will block execution until it receives a 'Callback Reply' message or the timeout duration of the VI is exceeded. Note that a synchronous enqueue is a blocking function (with a timeout), and one needs to be careful with

its use since it pauses the execution of the code while it waits for data to be received from another Worker. However, it can be useful if caution is heeded.

Enqueue and Forget.vi


Enqueues an asynchronous message to a Worker, defined by its Queue.

4.2.2 Replying to a callback message

It is possible to retrieve the callback message from the Worker that received the message using the *Get Callback Message.vi*. The Callback Message can then be stored for later use or used immediately to send a reply message back to the Worker that sent the Callback Message, by use of the *Callback Reply.vi*.

TIP: the use of both regular messaging and callback messaging are demonstrated in the Workers™ Fundamentals Sample Project.

4.2.3 Silent Mode

All the enqueue VIs described above also have a 'Silent Mode' option. The VI will appear the same however its icon it will have an additional  symbol on it. When an enqueue VI with 'Silent Mode' is used, the message will not appear in the Debugger's Message Log. Using this mode is useful when you have an MHL case that is called many times, possibly at high speed, and you wish not to fill the Debugger's Message Log with redundant messages.

4.3 MESSAGE CONTENTS

Every message contains several specific data types which are packaged together and sent with the message. These include:

4.3.1 Queue

The Queue of the Worker to whom the message is to be sent.

4.3.2 Case Label

A string identifier of the case which will receive the message in the recipient Worker's Message Handling Loop.

4.3.3 Data (optional)

Any data type of your choice. It will be sent as a variant and therefore will be required to be converted back to the specific data type when received by the recipient Worker.

4.3.4 Auxiliary (optional)

An additional data type of your choice. It will be sent as a variant and therefore will be required to be converted back to the specific data type when received by the recipient Worker.

4.3.5 Metadata (hidden)

This is data that is always sent by the framework along with the message, used to identify the sender Worker and recipient Worker of the message. This data is sent to the Debugger when the message is dequeued by the recipient Worker.

4.4 MESSAGE PRIORITIES

Workers™ uses a priority queue for the sending of messages between Workers. The priority queue has been adapted from the priority queue of the Actor Framework. Thus, the performance and limitations of the queue should be identical to that of the Actor Framework. For additional information about the priority queue please see section 13.3.

The priority queue contains four possible priorities, three of which are made available to the developer and one which is only available to the framework. These include:

Priority	Access
Critical	Framework Only
High	Developer
Normal (default)	Developer
Low	Developer

Table 1. Message Priorities

The initialization and shut down sequences of the framework use 'Critical' priority messages, which are not made available to the developer. The reason for this is that if a Worker's queue is overloaded with messages, the framework will be able to pre-empt these messages and shut down the application rapidly and correctly, increasing the reliability and performance of your application.

4.5 CASE LABELS

Traditionally, when sending a message to a QMH, the developer would use a string constant to identify the recipient case of the message. If a case is called from many different locations in your code, many string constants must be created individually with the name of the case. Misspelled strings, changes in a MHL's case name, etc. can create many forgotten and untraceable errors in your code. To help avoid these problems, the 'Case Label' was created.

A Case Label offers the developer a single location for the string definition of a case (in addition to the case name within the case structure itself).

A Case Label is simply a VI that contains a string indicator as its sole output, with the string being the name of a case in a Worker's MHL. While you can create a Case Label manually, the framework has a Quickdrop shortcut that does this for you and will produce a VI with the file name "c_(Case Name).vi". Figure 10 depicts this. In addition, the Case Label's icon header will take the icon header of the Worker that it belongs to.

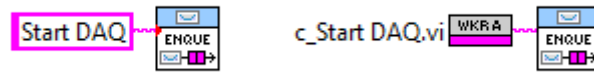


Figure 10. String constant (left) vs 'Case Label' (right) to enqueue a message to a Worker's MHL case 'Start DAQ'

Here are the advantages of using Case Labels over a string constants.

Single Location Definition

If you wish to change the name of a case in your Worker's MHL, you only have to change the string in the associated Case Label, and its filename.

Finding the Enqueue Locations to a Particular MHL case

If you wish to find all the locations in your code that enqueue a message to a particular case of an MHL, you can simply use 'Find all Instances' of the Case Label VI to do this.

Instantly Create your Case Labels

We have created a Quickdrop shortcut that can instantly create a Case Label for you from the visible block diagram case of your Worker's MHL. See Quickdrop Shortcuts section 6.2 for how to do this.

Jump directly to the Case Label's Case

By use of another Quickdrop Shortcut, if you select a Case Label on a VI's block diagram, you can jump directly the Case Label's corresponding MHL case. This is very useful for following the **flow of messages between cases and Workers** while developing. See Quickdrop Shortcuts section 6.3 for how to do this.

5 STATICALLY LINKED / DYNAMICALLY LOADING WORKERS

A Worker can be either statically linked to / or dynamically loaded by another Worker. The following section describes the difference between both methods. Both static linking and dynamic loading of Workers is demonstrated in the Workers™ Fundamentals Sample Project (section 11).

5.1 STATICALLY LINKED WORKERS

The majority of the time, you will build up the core Worker call-chain hierarchy of your application by using statically linked Workers. Every time you add a subWorker to a Worker with the 'Create/Add Worker' tool, the subWorker will be added as a static link and thus it will load at the same time its Manager is loaded.

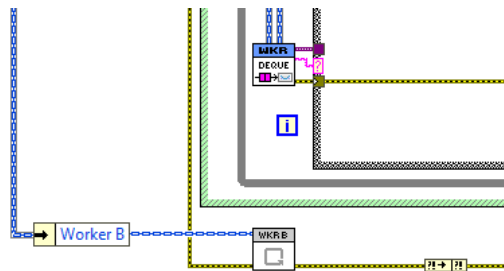


Figure 11. **Worker B** is a statically linked Worker

Worker B in Figure 11 is a statically linked Worker. **Worker B** is called at the same time that **Worker B**'s Manager is loaded. Every statically linked Worker is loaded automatically at runtime when the application runs. This is the most common way of calling a Worker.

5.2 DYNAMICALLY LOADING WORKERS

There are times where you want to call an unspecified amount of Worker's programmatically, in addition to the statically linked Workers that make up the majority of the Worker call-chain hierarchy of your application. For these times, loading a Worker dynamically is useful.

Unlike statically linked Workers, dynamically loaded Workers are not loaded until they are loaded with the *Dynamically Load Worker.vi*. This is a very easy process and is shown in Figure 12. By wiring a Worker Class Object Constant and a cluster of initialization data to the VI, the Manager is able to load **Worker C** (Figure 12) at any time, and load as many instances of **Worker C** as desired. Once the Worker has been dynamically loaded, it is automatically integrated into the Manager and will behave the same way as a statically linked Worker. Dynamically loading Workers is demonstrated in the Workers™ Fundamentals Sample Project.

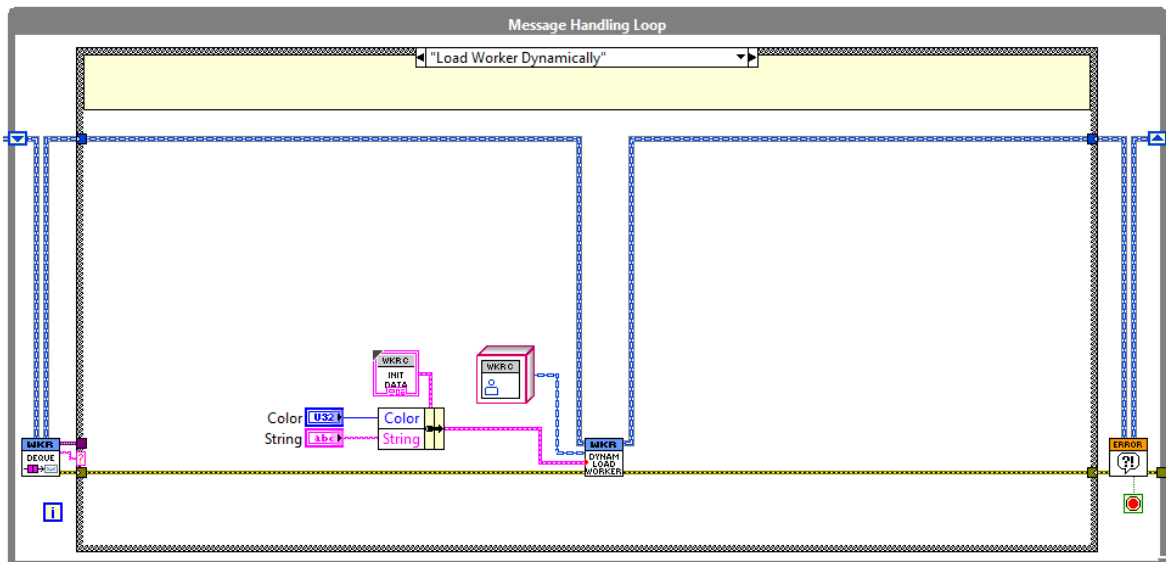


Figure 12. Dynamically loading **Worker C**.

Note: If you are loading Workers dynamically from a Worker that has not been set up as a Manager (i.e. contains statically linked subWorkers), then you will need to complete steps **9** and **10** of section 8.2 so that the shutdown sequence of the Manager will be propagated to the Dynamically Loaded Workers.

5.2.1 Troubleshooting

Error 1003 appears when loading a Worker Dynamically

A Worker can only be dynamically loaded if **all** the Workers' *Main.vi*'s in a project are not broken. Therefore, make sure that none of the Workers' *Main.vi*'s in your project are broken before you attempt to load a Worker dynamically.

6 QUICKDROP SHORTCUTS

A few Quickdrop shortcuts have been created to help automate common tasks and increase your development efficiency while using Workers™.

6.1 SHOW WORKER'S PRIVATE DATA (CTRL + 0)

To see the private data cluster of a Worker from any of the Worker's VIs. You can also access the private data cluster from the LabVIEW project explorer, but this is a faster way to directly access the private data cluster of a Worker while you are developing.

6.2 CREATE CASE LABEL (CTRL + 9)

To create a Case Label, follow the following steps.

1. Make a Worker's *Main.vi's* block diagram the active window. Make the MHL case visible that you want to create the Case Label for. E.g. 'Start DAQ' (Figure 13).

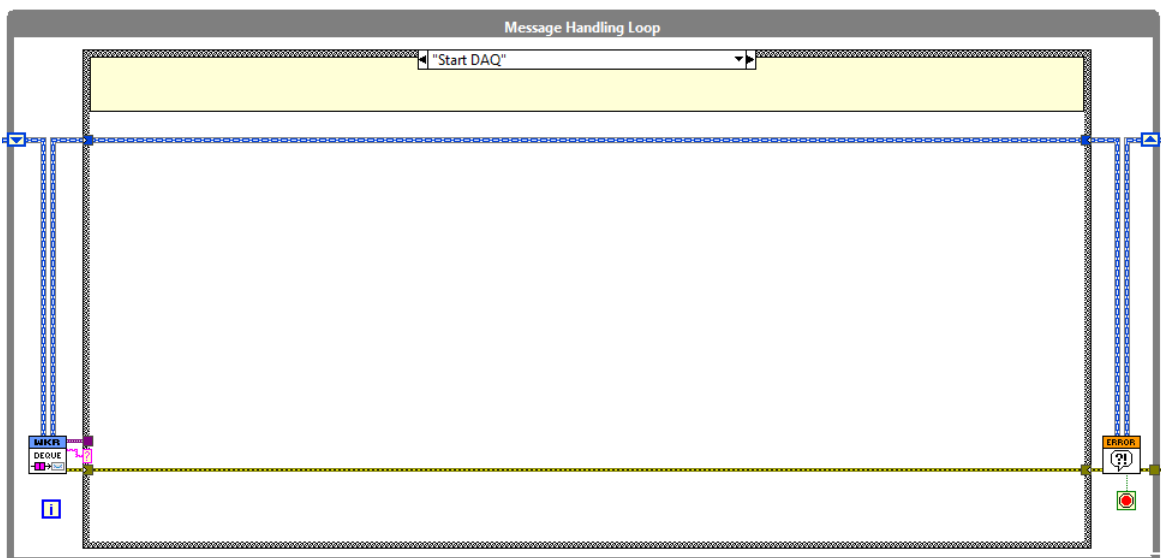


Figure 13. Case 'Start DAQ' of a Worker's MHL visible on block diagram

2. Activate the Quickdrop shortcut (Ctrl + 9). It will ask you to confirm before you create it (Figure 14).

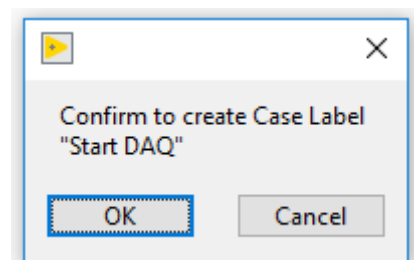


Figure 14.

- Once created, you will be notified of success or failure via another dialog box. The Case Label will be added to your Worker under the virtual Folder “Case Labels” (Figure 15). Make sure you save the Case Label VI as this is not automatically done for you.

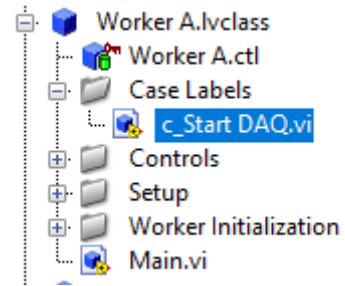


Figure 15. Case Label VI added to Worker

6.3 SHOW MHL CASE OF CORRESPONDING CASE LABEL (CTRL + 8)

- Select a Case Label VI that has been placed on a block diagram (Figure 16).
- Activate the Quickdrop shortcut (Ctrl + 8).

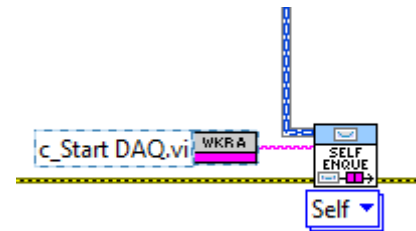


Figure 16. Case Label VI selected on block diagram

- The Worker’s MHL case (block diagram) that is associated with the Case Label should become visible (Figure 17).

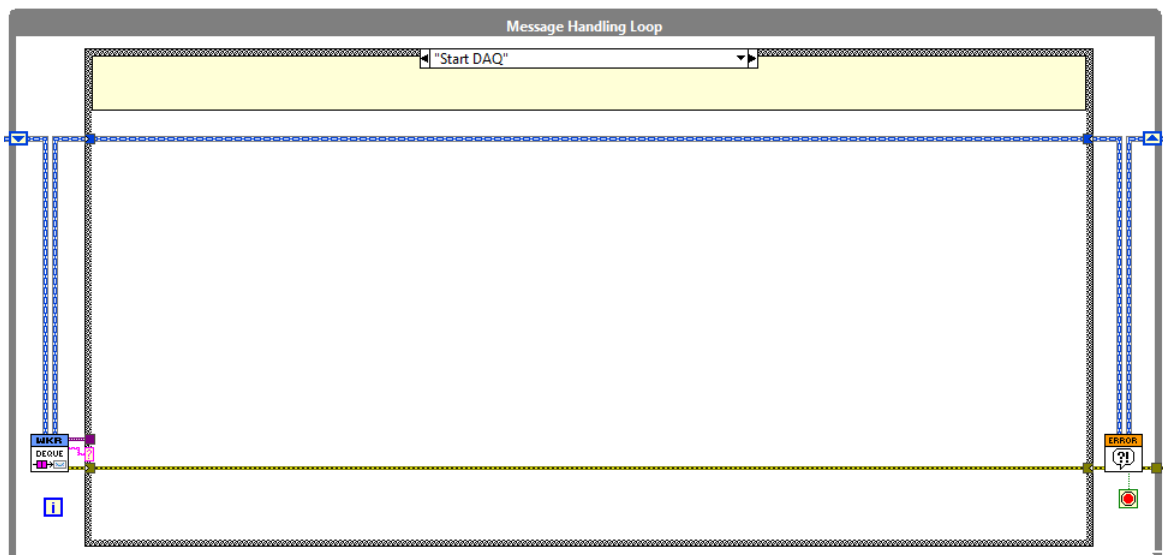


Figure 17. Case 'Start DAQ' of a Worker's MHL has become visible

7 DEBUGGER

The Debugger plays an integral part in the development of applications using Workers™ and its importance should not be understated. No matter how useful a framework is, if the code that it creates is difficult to understand and debug then the time-saving benefits that the framework brings will be significantly reduced.

The Debugger must be running **before** a Workers™ application is run and can be accessed directly from the *Workers™ tools* menu (*Tools>>Workers tools*) or directly loaded from a Launcher VI with the Debugger *Loader.vi*. The Workers™ Fundamentals Sample Project demonstrates how to do this.

Note: The Debugger uses the same framework code that every Worker uses. Therefore, whenever the Debugger is running you will see a ‘locked’ symbol over your Worker’s *.lvclass icons in the project explorer. Closing the Debugger will remove this ‘locked’ symbol.

7.1 TASK MANAGER TAB

The Debugger’s ‘Task Manager’ tab allows the developer to see a hierarchical list of all the loaded Workers in an application showing the call-chain of each Worker in relation to the application’s Head Worker, the Worker’s Clone ID (since every Worker’s *Main.vi* is a shared-clone), and the current status of each Worker. Figure 18 shows the ‘Task Manager’ tab of a running Workers™ application with five Workers.

Task Manager		Message Log	Message Log Backlog Count 0
Worker ID	Clone ID	Status	
Worker A	4140001	Initialized	
Worker B	4140001	Initialized	
Worker C	4140001	Initialized	
Worker D	4140001	Initialized	
Worker E	4140001	Initialized	

Figure 18. Debugger ‘Task Manager’ tab, showing status and call-chain list of an applications loaded Workers

Table 2 lists all the framework Worker statuses that currently exist, along with what they mean. The Workers API also allows you to add your own custom statuses. This is explained in Section 12.3.

Status	Description
Queue Created	The Queue for the Worker has been created by the <i>Setup subWorkers.vi</i> of the Worker’s Manager (or by the <i>Setup Head Worker.vi</i> if the Worker is the Head Worker.)
	<u>If you see this:</u> the Worker’s <i>Main.vi</i> does not exist on the block diagram of its Manager, and therefore messages sent to it will never be dequeued.

Pre-initialized	<p>The Worker's <i>Main.vi</i> has been loaded and it is able to dequeue messages that are sent to it.</p> <p><u>If you see this:</u> the Worker has not yet had its 'Initialize' case called by its Manager. This is done by <i>Initialize subWorkers.vi</i> in the Manager.</p>
Initialized called	<p>The Worker's 'Initialize' case has been called by the Worker's Manager.</p> <p><u>If you see this:</u> the Worker has not yet notified its Manager that it has been successfully Initialized. This is done in the Worker by <i>Initialized Notify.vi</i>.</p>
Initialized	<p>The Worker has notified its Manager that it has been successfully initialized by using <i>Initialized Notify.vi</i>.</p> <p><u>If you see this:</u> this is the status you will see when your Worker has been initialized correctly. No other framework statuses will appear for this Worker until the Worker starts to shut down.</p>
Start Exiting called	<p>The Worker's 'Start Exiting' case has been called by use of either the <i>Start Exiting Worker.vi</i>, or by the Worker's Manager use of <i>Start Exiting subWorkers.vi</i>.</p> <p><u>If you see this:</u> the Worker has not yet notified its Manager that it has successfully exited. This is done in the Worker by <i>Exited Notify and Cleanup.vi</i>, and therefore should be the very last VI that is called by the Worker's <i>Main.vi</i>.</p>
Exited	<p>The Worker has notified its Manager that it has successfully exited by using <i>Exited Notify and Cleanup.vi</i>.</p> <p><u>If you see this:</u> this is the status you will see when your Worker has exited correctly. This is the final framework status of the Worker.</p>
Stopped (Critical Error)	<p>A critical error has occurred in the Worker. The Worker has stopped and can no longer be accessed by the framework.</p> <p><u>If you see this:</u> The Worker's queue has been destroyed unexpectedly.</p>
(Aborted)	<p>This string is appended to any of the above statuses, whenever the application is terminated pre-maturely.</p>

Table 2. Worker status definitions

7.1.1 Right Click Menu

Right clicking on the table in the Task Manager tab provides the developer with some useful features. The right click menu of the Task Managers tab's table is shown in Figure 19.

Task Manager		Message Log	Message Log Backlog Count
		0	
Worker ID	Clone ID	Status	
Worker A	4140001	Initialized	
Worker B	4140001	Initialized	
Worker C	4140001	Initialized	
Worker D	4140001	Initialized	
Worker E	4140001	Initialized	

Figure 19. Task Manager tab right-click menu

Open Running VI

Shows the block diagram of the Worker's *Main.vi* that is currently running in memory. If the VI is no longer running, the VI of the Worker's editable *Main.vi* will open.

Filter as Enqueue Worker

Filters the Worker's ID as the 'Enqueue Worker' in the Debugger's Message Log tab.

Filter as Dequeue Worker

Filters the Worker's ID as the 'Dequeue Worker' in the Debugger's Message Log tab.

7.2 MESSAGE LOG TAB

Every time a message is dequeued by a Worker, metadata about the message can be sent to the Debugger. This information is shown in the Debugger's 'Message Log' tab. The main purpose of the Message Log tab (shown in Figure 20) is to:

- Show the chronological flow of messages between Worker MHL cases.
- Show errors that occur in an application, and identify when and where they originated
- Allow the user to jump directly to any case shown in the Message Log.

Task Manager

Message Log

Message Log Backlog Count 0

String to Filter

Column 4

Filter Type string only

Filter

Reset

Sort

Pause

Full ID

-Auto-

Time/Date	Enque Worker	Enque Case	Deque Worker	Deque Case	Message
2019-05-05 11:45:09	Worker A	Get Value	Worker B	Get Value	
2019-05-05 11:45:09	Worker A	EHL Case	Worker A	Get Value	
2019-05-05 11:45:07	Worker A	EHL Case	Worker A	Receive String	
2019-05-05 11:44:56	Worker A	Load Worker Dynamically	DLW01	Initialize	
2019-05-05 11:44:56	Worker A	EHL Case	Worker A	Load Worker Dynamically	
2019-05-05 11:44:50	Worker A	Get Value	Worker B	Get Value	
2019-05-05 11:44:50	Worker A	EHL Case	Worker A	Get Value	
2019-05-05 11:44:29	Worker A	Default	Worker A	All subWorkers Initialized	
2019-05-05 11:44:29	Worker A	Initialize	Worker E	Initialize	
2019-05-05 11:44:29	Worker B	Initialize	Worker B	Increment Value	
2019-05-05 11:44:29	Worker A	Initialize	Worker D	Initialize	
2019-05-05 11:44:29	Worker A	Initialize	Worker B	Initialize	
2019-05-05 11:44:29	Worker A	Default	Worker A	Initialize	

Figure 20. Debugger 'Message Log' tab, showing the flow of messages between Worker MHL cases

7.2.1 Columns

Each row of the Message Log contains several columns. These columns can be disabled/enabled by use of the Debugger's toolbar menu 'Message Log'. The Message Log's columns include:

Count (sec)

This column is used as a high precision timestamp of the message. This timestamp is given to the message when it is enqueued. The value is provided by the LabVIEW VI: *High Resolution Relative Seconds.vi*. It is not possible to disable this column.

Time/Date

Shows the date and time, to the nearest second, of when the message was enqueued.

Enque Worker

The ID of the Worker that enqueued the message.

Enque Case

The case of the Worker where the message was enqueued.

Deque Worker

The ID of the Worker that dequeued the message.

Deque Case

The case of the Worker that received the message.

Message

A string that was sent to the debugger along with the message. Messages that appear here are sent to the Debugger the following ways:

1. By the user, by use of the *Send Debugger Message.vi* (available in the Workers™ palette).
2. By the framework, when an error is detected by the *Error Handler.vi*.

If a row of the Message Log is selected (mouse left-click), the message of the row will appear in the bottom frame of the Debugger (as in Figure 21).

7.2.2 Error Messages

An example of an error sent by an application to the Debugger is shown in Figure 21. The error row has been selected and information about the error is shown in the bottom frame of the Debugger.

Time/Date	Enque Worker	Enque Case	Deque Worker	Deque Case	Message
2019-03-25 11:50:56	Worker A	Flags Set	Worker E	Open Connection	Error -2132869110 occurred at Worker E.lvclass:Main.vi:414

Message:

Error -2132869110 occurred at Worker A.lvclass:Main.vi

Possible reason(s):

LabVIEW: (Hex 0x80DF000A) A communication error between the host and target occurred.

Complete call chain:

Worker A.lvclass:Main.vi

Figure 21. Error received by the Message Log

The timestamping of the error is performed by the *Error Handler.vi* when it receives an error. The 'Enque' and 'Deque' columns identify the message-call-chain of the case that the error occurred in.

7.2.3 Controls

For information about the individual controls on the Message Log tab, please see the tip-strip information that appear when you hover the mouse over a control.

7.2.4 Right Click Menu

Right clicking over a cell in the Message Log provides the developer with some useful features. The right-click menu of the Message Log is shown in Figure 22.

Time/Date	Enque Worker	Enque Case	Deque Worker	Deque Case	Message
2019-03-25 11:53:16	Worker A	Load Worker Dynamically	DLW01	Initialize	
2019-03-25 11:53:16	Worker A	EHL Case	Worker A	Load Worker Dynamically	
2019-03-25 11:53:09	Worker A	Get Value	Worker B	Get Value	
2019-03-25 11:53:09	Worker A	EHL Case	Worker A	Get Value	
2019-03-25 11:53:03	Worker A	Default	Worker A	All subWor	
2019-03-25 11:53:03	Worker A	Initialize	Worker E	Initialize	
2019-03-25 11:53:03	Worker B	Initialize	Worker B	Increment	
2019-03-25 11:53:03	Worker A	Initialize	Worker D	Initialize	

Figure 22. Message Log tab table right-click menu

Filter String (Double Click)

The column will be filtered by the string in the cell. You may also double click on a cell in the Message Log to perform the same function.

Open Running VI

Shows the block diagram of the Worker's *Main.vi* that is currently running in memory.

Go to Case

Shows the specific case of the Worker's editable *Main.vi*.

7.2.5 Message Log Backlog Count

The 'Message Log Backlog Count' indicator (on the Debugger's Front Panel) shows how many elements are in the Debugger's queue awaiting to be displayed to the Message Log. The Message Log can only be updated so fast, and when the Debugger is sent more elements than it can display, then the backlog count will start to increase. This can be avoided, by using the 'Silent Mode' option for enqueue VIs, for MHL cases that are called at high speed, potentially clogging up the Message Log's backlog. See section 4.2.3 for how to use 'Silent Mode' enqueue VIs to avoid this.

8 CREATING/ADDING/REMOVING WORKERS

The recommended way of creating and/or adding a Worker (as a subWorker) to another Worker is via the use of the 'Create/Add Worker' tool, which can be found under the *Workers™ tools* menu (*Tools>>Workers tools*). However, if the tool fails you can still add a Worker manually to another Worker. Both methods are described in this section.

8.1 'CREATE/ADD WORKER' TOOL (RECOMMENDED WAY)

With this tool it is very easy to create and/or add a Worker (as a subWorker) to another Worker, making it quick and easy to create the entire frame of an application in very little time. It is recommended that you use this tool over the manual method described in the next section, since the tool automates many processes for you which would otherwise take considerable time. It is also possible to add an existing Worker to another Worker using the 'Add Existing...' tab of the tool.

When you run the tool, you will see the following window (Figure 23).

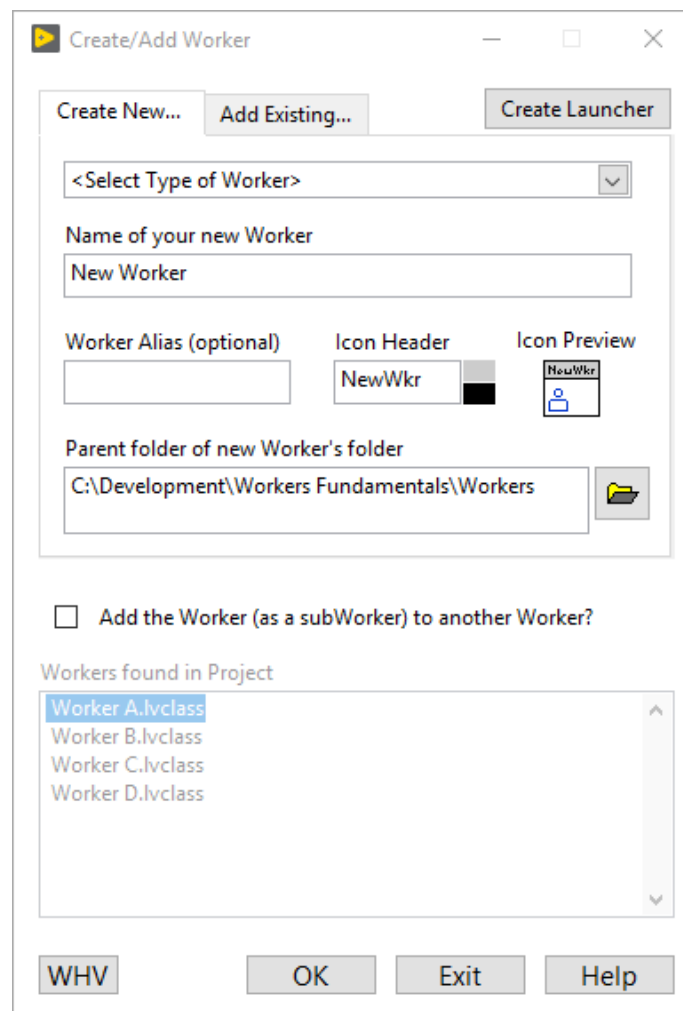


Figure 23. Create/Add Worker tool window

8.1.1 Control Descriptions

Creating a new Worker is a straightforward process. The developer has the following options:

<Select Type of Worker>

Select if you want your new Worker to contain an Event Handling Loop (EHL) or not. It is recommended that you choose 'Worker with EHL' so that you do not have to manually add one yourself later. There is no performance difference and the framework will automatically stop the EHL for you when you exit your Worker. Additionally here you can also choose to create a new Worker based on your own custom Worker templates (see section 12.4).

Name of your new Worker

Select a name for your Worker. A *(SelectedName).lvclass* file will be created, so make sure that the name you choose can also be used as a file name. Periods '.' are not allowed in the Worker's name.

Worker Alias (optional)

The name that the Worker will be identified as within your Workers™ application and by the Debugger. If left blank, the Worker's name will be taken as the alias. This field is useful to provide a shorter name for your Worker within your code if your Worker's name is long and descriptive.

Icon Header

A header string for your Worker's icon. Here you can also select the background and text color.

Icon Preview

The icon of your Worker, containing your text header and the standard Worker icon logo.

Parent folder of new Worker's folder

Every time a Worker is created, a folder (with the same name as the Worker) is created to store its VIs. The 'parent folder' is the folder one level above your Worker's folder.

Add the Worker (as a subWorker) to another Worker?

If you want to add the Worker as a subWorker to another Worker, then select this option. For detailed information about what happens when you select this option, please see section 8.1.2.

Workers found in Active Project

A list of the Workers that are found in your active project that you can add another Worker to. This control is disabled if 'Add the Worker (as a subWorker) to another Worker' checkbox is not checked.

8.1.2 Tasks performed by the tool

For your interest, a summary of the automated tasks performed by the tool will now be provided.

Creating a Worker WITHOUT adding it to another Worker

The tool creates a new Worker from a template. The Worker's class icon will be changed, and the VIs of the Worker will adopt the Worker's icon header. Labels of the Worker's control constants on the VIs will take the Worker's given name. The Worker is then added to your project.

Creating a Worker AND adding it to another Worker

First the tool creates a new Worker from a template, etc. as described directly above.

In addition, changes must occur to the Worker's Manager (the Worker that you are adding the new Worker to). Changes to the Manager include:

- Additional cases are added to the MHL ('All subWorkers Initialized' and 'All subWorkers Exited')
- *Initialized Notify.vi* is replaced with *Initialize subWorkers.vi* on the *Main.vi*'s block diagram
- *Exit.vi* is replaced with *Start Exiting subWorkers.vi* on the *Main.vi*'s block diagram
- A control *subWorkers.ctl* is created and added to the Manager
- The new Worker's object control is added to the Manager's *subWorkers.ctl* cluster
- *subWorkers.ctl* is added to the Manager's private data cluster
- *Setup subWorkers.vi* is created and added to the Manager
- *Setup subWorkers.vi* is added to the Manager's *Main.vi*'s block diagram
- An initialization VI titled *Set Initialization Data - (new workers name).vi* is created and added to the Manager
- The new Worker's *Main.vi* is added to the Manager's *Main.vi*'s block diagram, beneath the QMH. The VI's terminals are wired up.

The result of the above process is shown in Figure 24. Here, a Worker was created by the tool (**Worker B**) and added as a subWorker to an already existing Worker (**Worker A**). **Worker B**'s *Main.vi* will appear on **Worker A**'s *Main.vi* Figure 24 (left). **Worker B** will be added to the project and new VIs will appear in **Worker A.lvclass** Figure 24 (right).

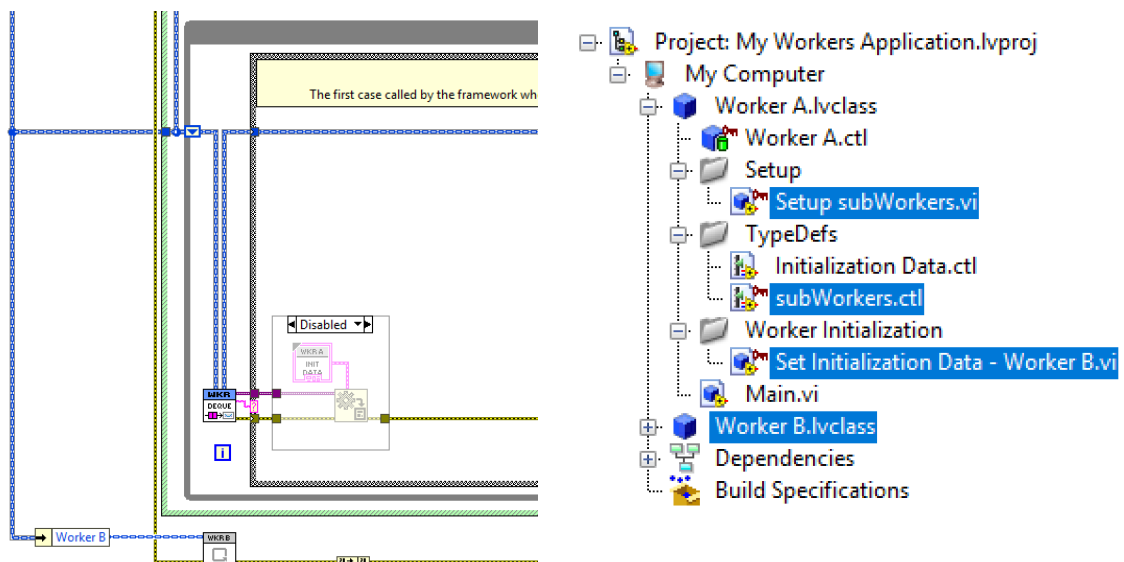


Figure 24. **Worker B** has been added to another Worker's block diagram (left) and added to the project (right).

8.1.3 Troubleshooting

If you run into problems with the 'Create/Add Worker' tool, it is possible to manually add a Worker to another Worker (see section 8.2). However, a few reasons may exist for the tool to fail.

Note: Do not place framework VI's (those that exist by default on a *Main.vi*'s block diagram) in subVIs. This ensures that when you add a Worker to another Worker, everything will go smoothly as the tool can find and replace any framework VI's on the Manager's *Main.vi* as required.

Strange Wiring of newly placed VI's on Manager's Main.vi

This can occur when the tool tries to add items but is unsure exactly where to place them. The tool does its best to place items neatly, but occasionally it may be required to clean up the placing of the new items. This can occur if the Manager's *Main.vi* already contains a lot of other items that may be in the way of where the tool would place the new items by default.

Manager class is locked

New files cannot be added to the Manager if it is locked. Make sure the Manager is unlocked before adding a new Worker to it.

Manager's Main.vi is broken

This can cause problems when adding new Workers. Make sure that the Manager's *Main.vi* is not broken before adding a new Worker to it.

Worker or files already exist on disk

If the tool fails, you may have already created a Worker with the same name previously and removed it from your project, without deleting the files from the disk. Delete the Worker from the disk before attempting again or use another name for the new Worker you wish to create.

VI's aren't wired correctly

It is possible that if the tool cannot find items on the Manager's *Main.vi* required for wiring up the new *Main.vi*, then the tool will fail. For example, the 'Merge Error' node is required to exist for the new *Main.vi* to be automatically wired up correctly.

A Worker's subWorkers.ctl is empty

If any of the Worker's in your project have a *subWorkers.ctl* that contains an empty cluster, it is possible the 'Create/Add Worker' tool won't load at all. Either remove the cluster from the Worker or add an element into it, apply changes, and then save it. When it is no longer broken, run the tool again.

8.2 MANUALLY ADDING A WORKER AS A SUBWORKER

The following process will take you from creating a new Worker from a template, to adding it to another Worker as a statically linked subWorker. It is assumed that you already have one Worker (**Worker A**) in your project, which you will now add a new Worker to.

8.2.1 Create a new Worker

1. This can be done simply by copying a Worker (*.lvclass) that you have already created, or from a template. The Worker templates used by the framework can be found under the LabVIEW installed directory at:

...\(LabVIEW 20xx) \vi.lib\addons_Workers\Templates\Worker with EHL

In this folder, drag and drop the *Worker with EHL Template.lvclass* file into the Project Explorer of your Workers™ Application.

2. Make a copy of the template by right clicking on it in the project and selecting 'Save As'. The dialog box should appear.

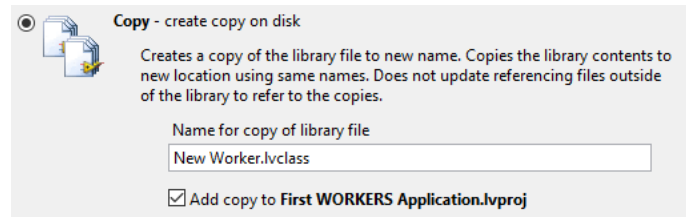


Figure 25. Right-click 'Save As...' dialog box

Give your Worker a new name, click OK, and create and then select a dedicated folder for the Worker and its VIs. Make sure you remove the template Worker from the project afterwards so that you do not accidentally modify it.

8.2.2 Add the new Worker to another Worker

If the Worker you are adding your new Worker to does not already contain any statically linked subWorkers, you will need to first make it a Manager so that it can accept your new Worker. If it already contains at least one statically linked subWorker, then you can jump directly to **step 12**.

Adding Manager features to a Worker

The Worker you want to add the new Worker to will be known here as the Manager.

Find the library file *Setup subWorkers Template.lvlib* and drag and drop it into your project. This can be found under the LabVIEW installed directory at:

...\ (LabVIEW 20xx) \vi.lib\addons_Workers\Templates\Setup subWorkers Template

3. Make a copy of this library (same as in Step 2) and don't forget to remove the template library.

Move the contents of your new library into the Manager, and then remove the library file you created. You can also delete this from disk as it is no longer needed. You should have two new files now in your project, as shown in Figure 26.

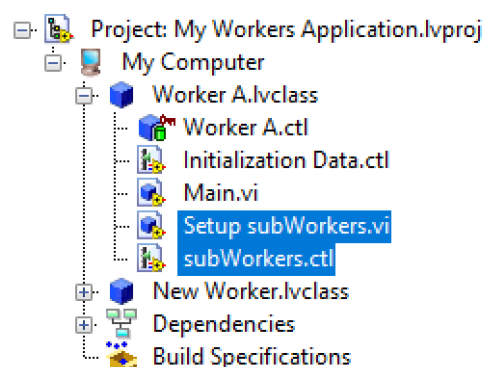


Figure 26.

4. Add the *subWorkers.ctl* to the Manager's private data cluster (Worker A.ctl). It is normal that the VI will be broken since the 'subWorkers' cluster is currently empty, as shown in Figure 27.

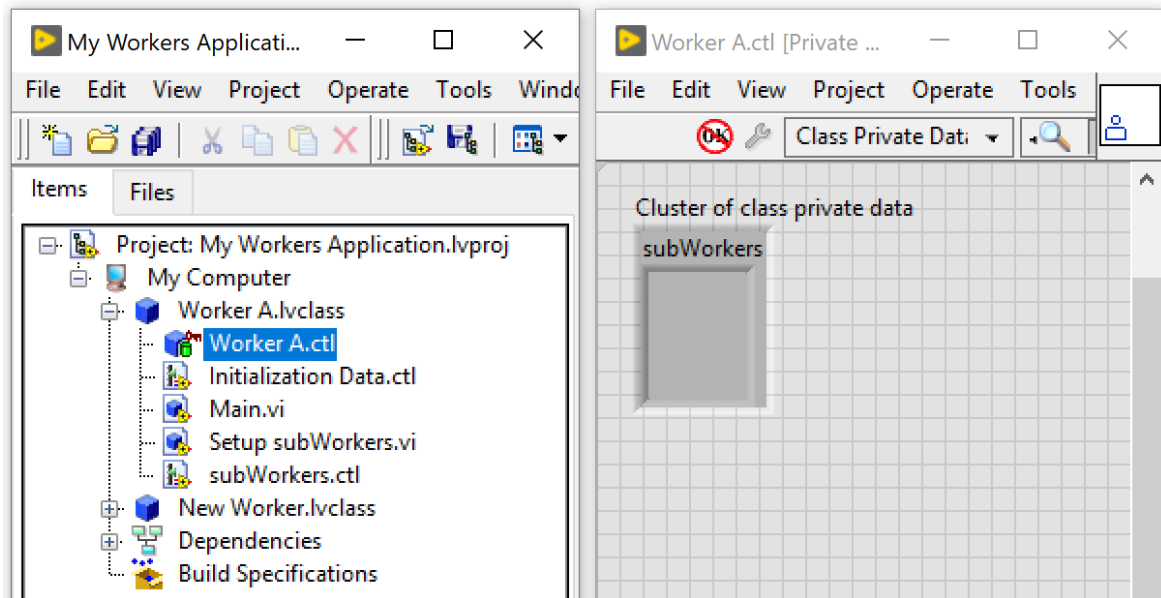


Figure 27. *subWorkers.ctl* added to Manager's private data cluster

5. Open *Setup subWorkers.vi*. Replace the Worker.lvclass object controls with the Manager's object controls and change their labels appropriately. The VI's front panel should end up looking like that in Figure 28. Make sure the wiring on the VIs block diagram hasn't changed.

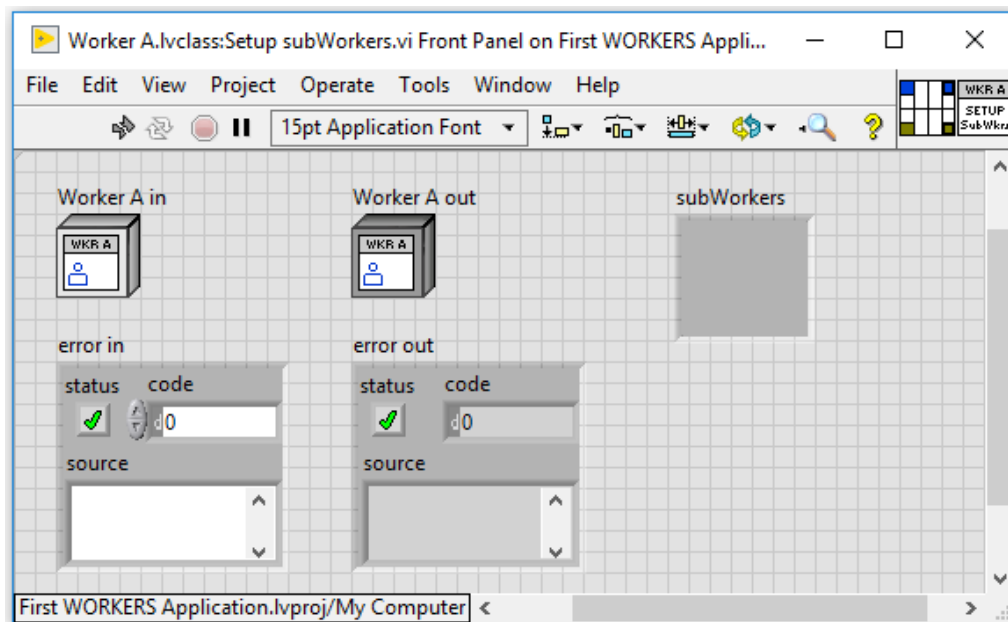


Figure 28. *Setup subWorkers.vi* after front panel changes

6. Add *Setup subWorkers.vi* to the Managers *Main.vi* block diagram, as shown in Figure 29.

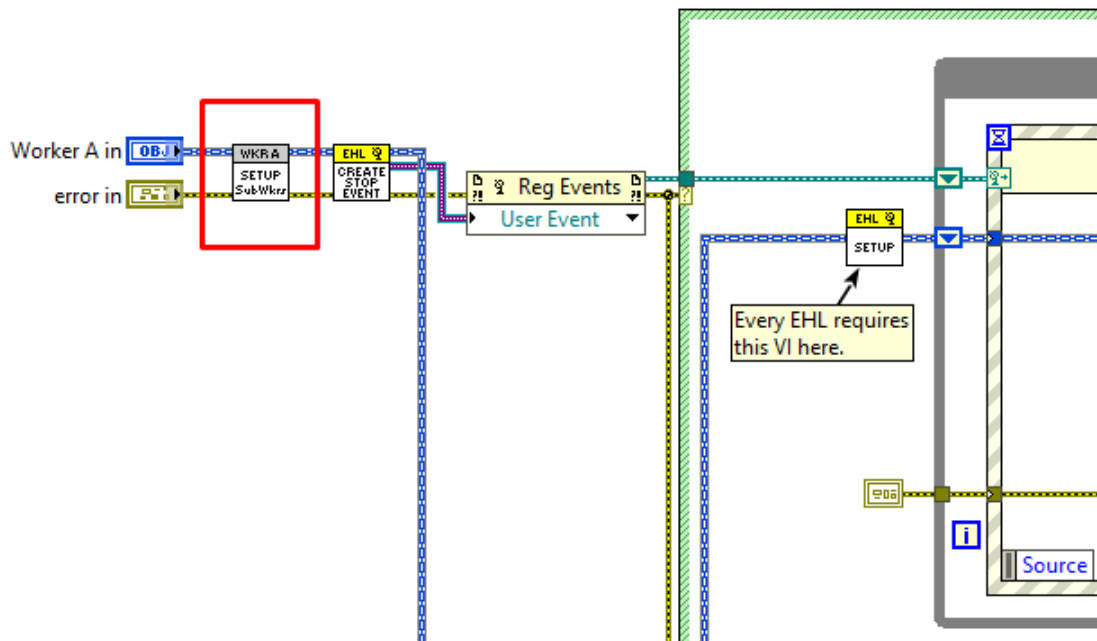


Figure 29. Setup subWorkers.vi added to Manager's Main.vi block diagram

7. In the Manager's 'Initialize' case of its MHL, replace the *Initialized Notify.vi* with *Initialize subWorkers.vi* as shown in Figure 30. This VI is available in the Workers™ palette.

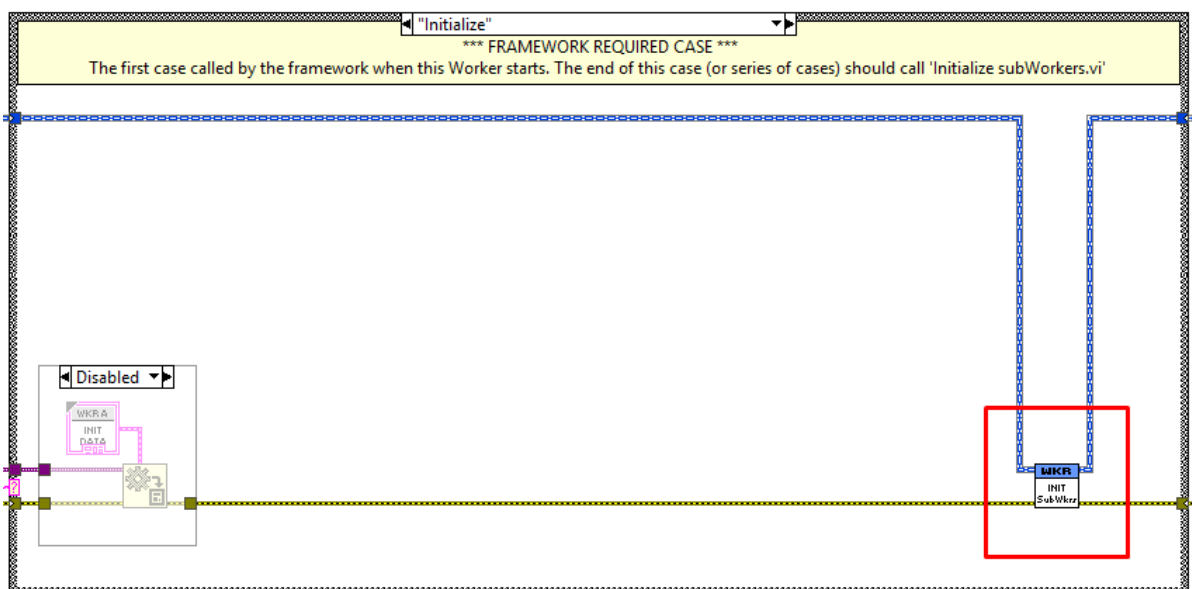


Figure 30. 'Initialize' case with Initialize subWorkers.vi

8. Add a new case to the Manager's MHL, called 'All subWorkers Initialized'. Add *Initialized Notify.vi* as shown in Figure 31. This VI is available in the Workers™ palette.

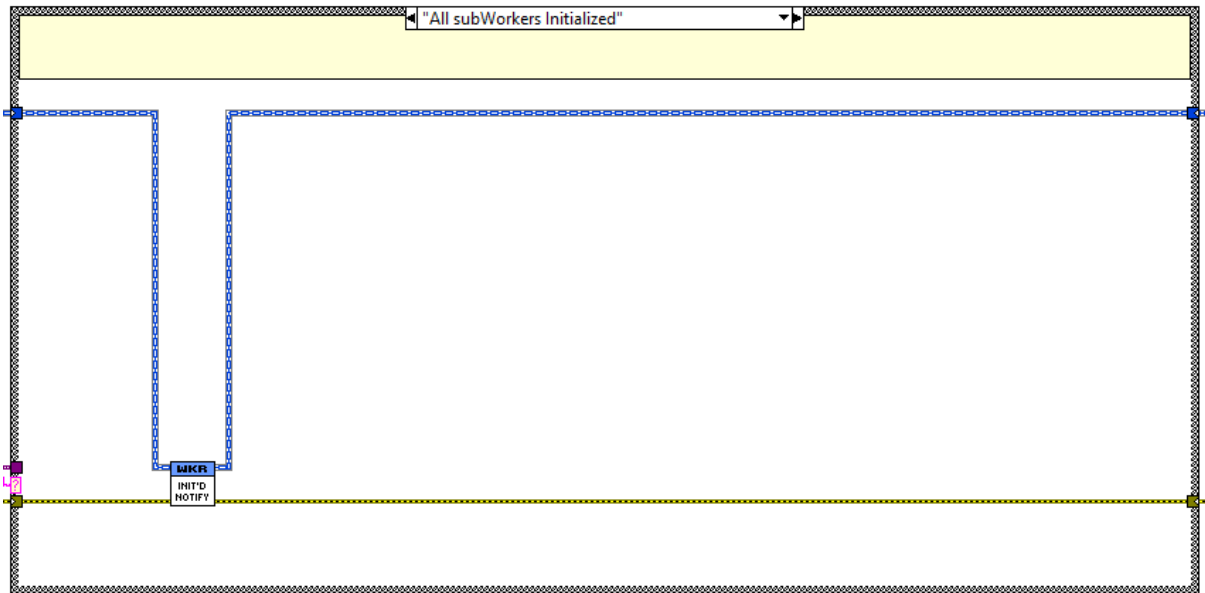


Figure 31. 'All subWorkers Initialized' case with *Initialized Notify.vi*

9. In the Manager's 'Start Exiting' case of its MHL, replace the *Exit.vi* with *Start Exiting subWorkers.vi* as shown in Figure 32. This VI is available in the Workers™ palette.

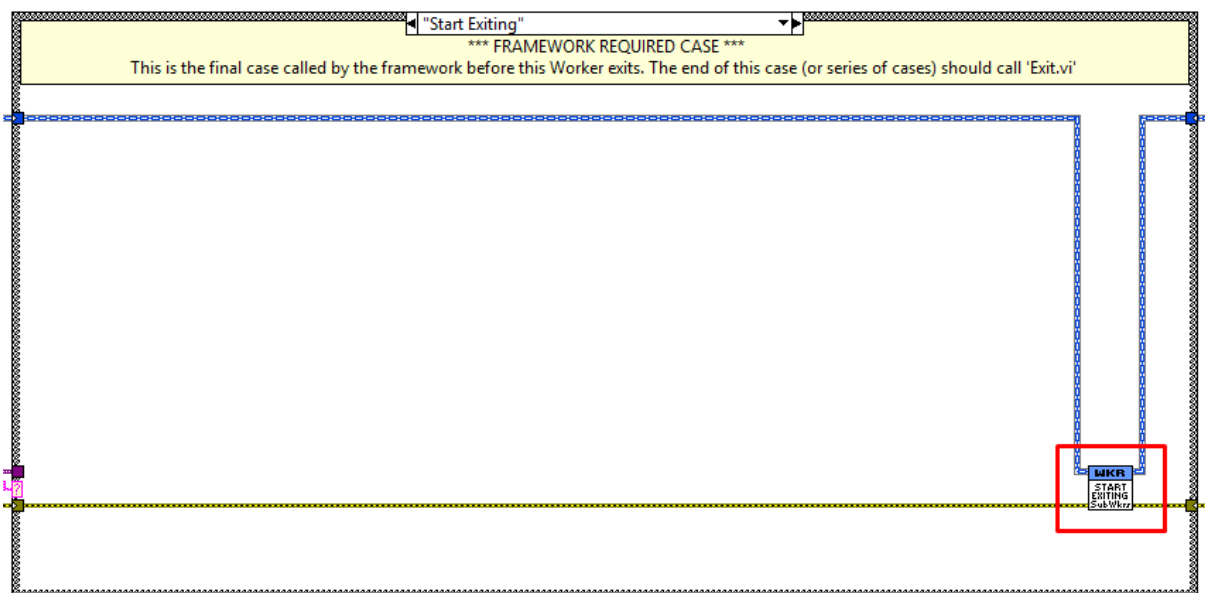


Figure 32. 'Start Exiting' case with *Start Exiting subWorkers.vi*

10. Add a new case to the Manager's MHL called 'All subWorkers Exited'. Add *Exit.vi* as shown in Figure 33. This VI is available in the Workers™ palette.

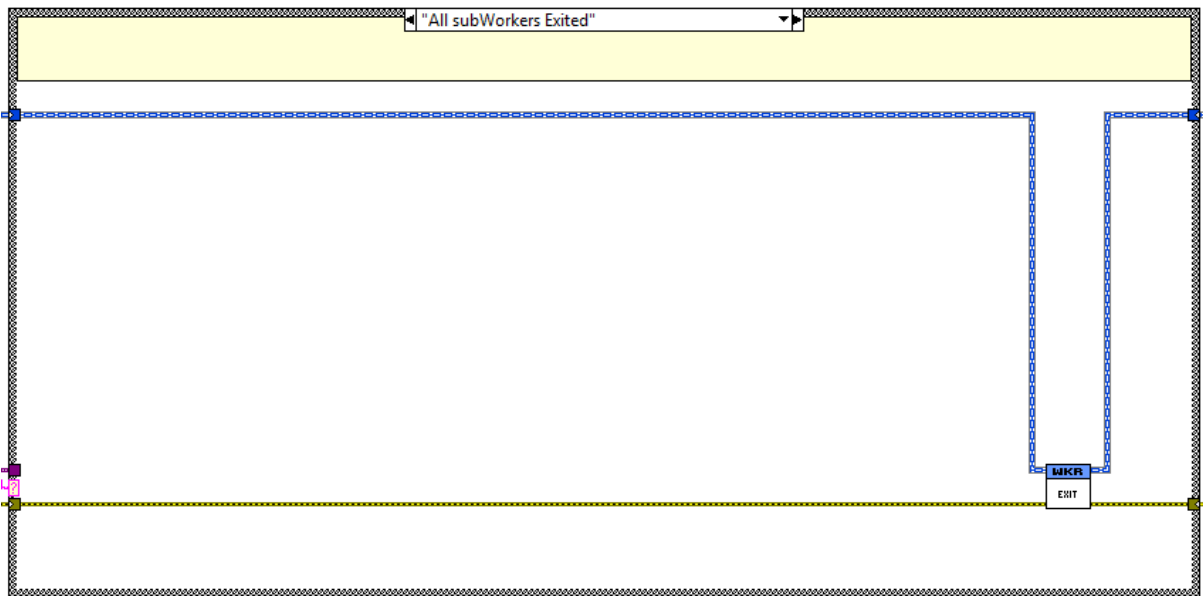


Figure 33. 'All subWorkers Exited' case with *Exit.vi*

11. This completes the addition of the Manager features to the Worker to allow the Worker to now accept subWorkers.

Adding a subWorker to the Manager

We will now add the Worker (created in **steps 1 and 2**) to the Worker that has been setup as a Manager.

12. Add the Worker to the Manager's *subWorkers.ctl* (i.e. drag and drop the **New Worker.lvclass** object from the project explorer into the *subWorkers.ctl* cluster on its front panel). **IMPORTANT: Remove the ".lvclass" from the Control's label.** Update the control. You should notice now that the Manager's private data cluster is no longer broken as shown in Figure 34.

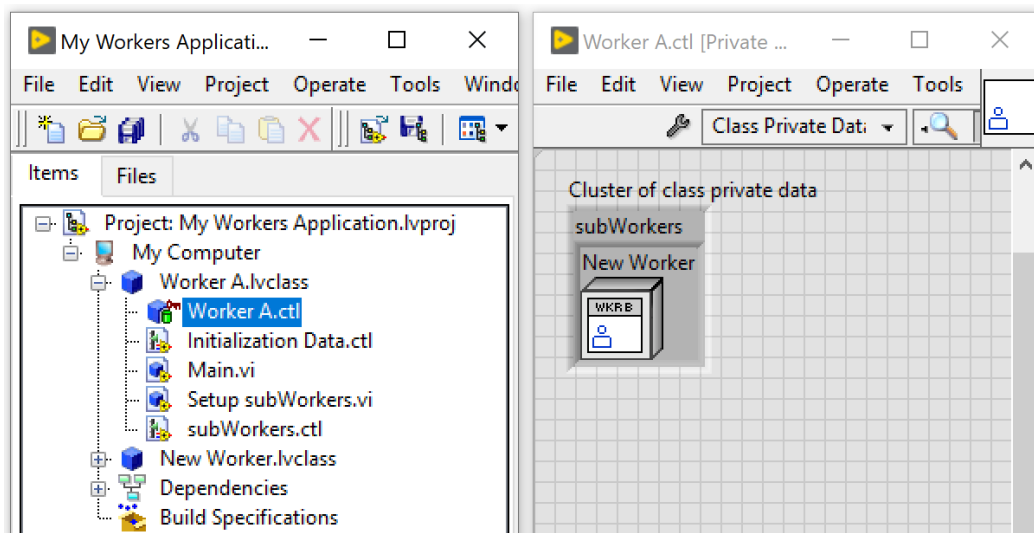


Figure 34. New Worker added to *subWorkers.ctl*. Manager's private data cluster is no longer broken.

- Double check that in the Manager's *Setup subWorkers.vi*, that the named-bundler's selected element is set to 'subWorkers' as shown in Figure 35.

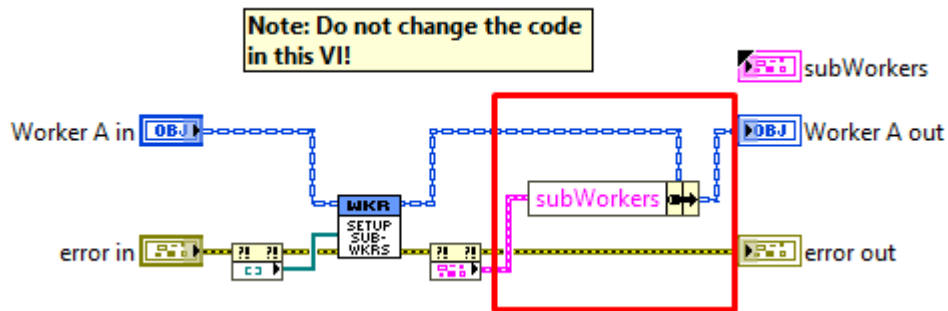


Figure 35. Setup subWorkers.vi named-bundler node selected as 'subWorkers'

- Finally, add the New Worker's *Main.vi* onto the block diagram of the Manager's *Main.vi*. Wire up its terminals as shown in Figure 36.

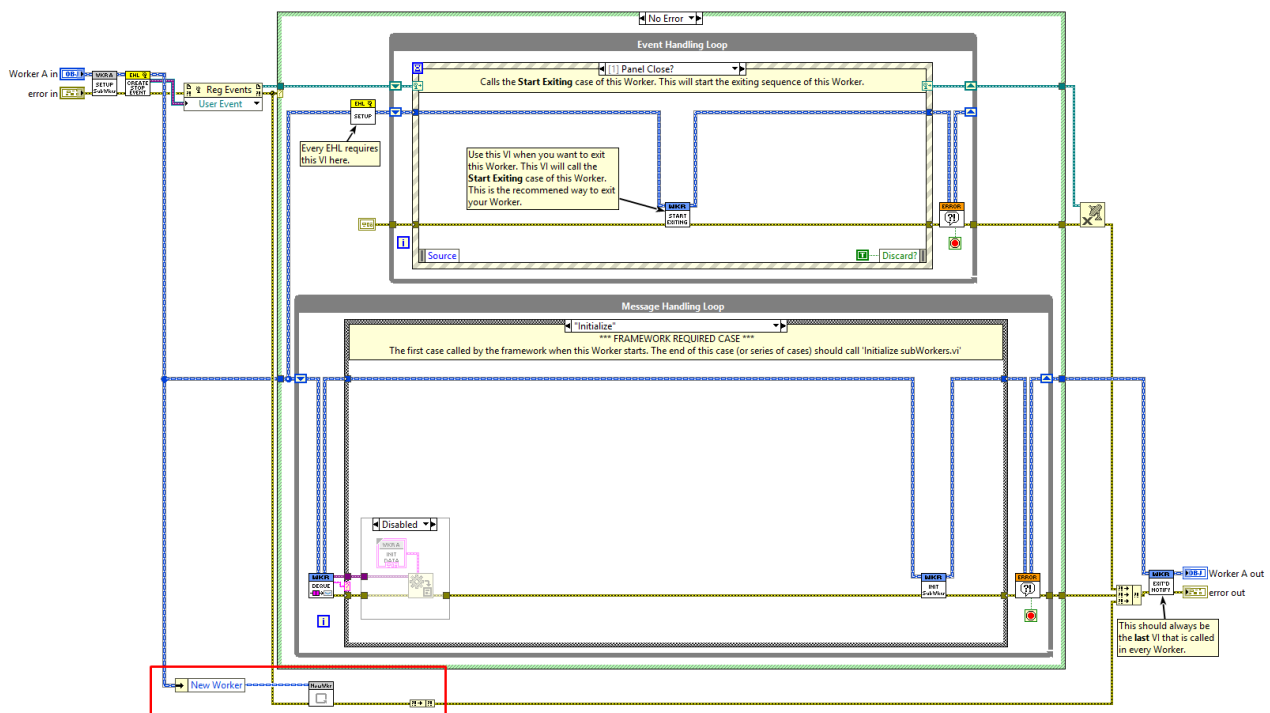


Figure 36. New Worker's Main.vi added to Manager's Main.vi

- This completes the addition of a Worker (as a subWorker) into another Worker. If you launch **Worker A** with a Launcher VI (see Chapter 9), the application should now run with the addition of the new Worker.

8.3 REMOVING A SUBWORKER FROM ITS MANAGER

Currently, there is no tool to automate this process, but it is simple to do manually. The following steps will help you remove a subWorker from its Manager:

1. Remove the Worker's object control from the Manager's *subWorkers.ctl*.
2. If there are no more Workers in the *subWorkers.ctl*, then the *subWorkers.ctl* must be removed from the Manager's private data, otherwise the Manager will be broken.
3. If there are no more Workers in the *subWorkers.ctl*, remove the *Setup subWorkers.vi* from the Manager's *Main.vi*'s block diagram.
4. Remove the Worker's *Main.vi* from the Manager's block diagram.
5. This completes the steps required to remove a subWorker from its Manager.

9 LAUNCHER VIs

A Worker's *Main.vi* cannot run as a stand-alone application, since a Worker cannot create its own message queue. (A Worker's message queue is always created by its Manager or Launcher VI.) Also, Workers are designed to be re-useable and can exist anywhere within an application's Worker call-chain hierarchy. It is for these reasons that every Worker's *Main.vi* run button is disabled by default.

However, the Head Worker of an application needs to be called somehow, and for this you use a Launcher VI. A Launcher VI creates a message queue for the Head Worker and then calls the Head Worker's *Main.vi*. An example of a Launcher VI's block diagram that is to launch **Worker A** is illustrated in Figure 37.

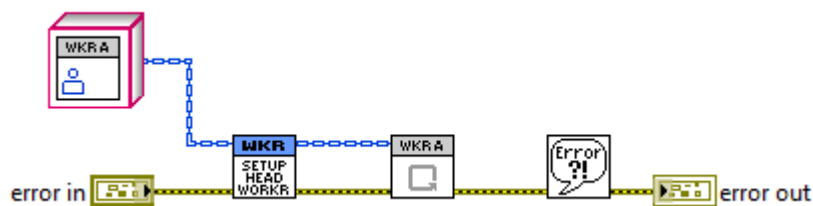


Figure 37 - Contents of a Launcher VI

Every time you create a new project and add your first Worker to it, you will have the option to also create a Launcher VI for it. You may also create Launcher VIs for any Worker with the Create Launcher tool, which is accessible through the 'Create/Add Worker' tool, shown in Figure 38.

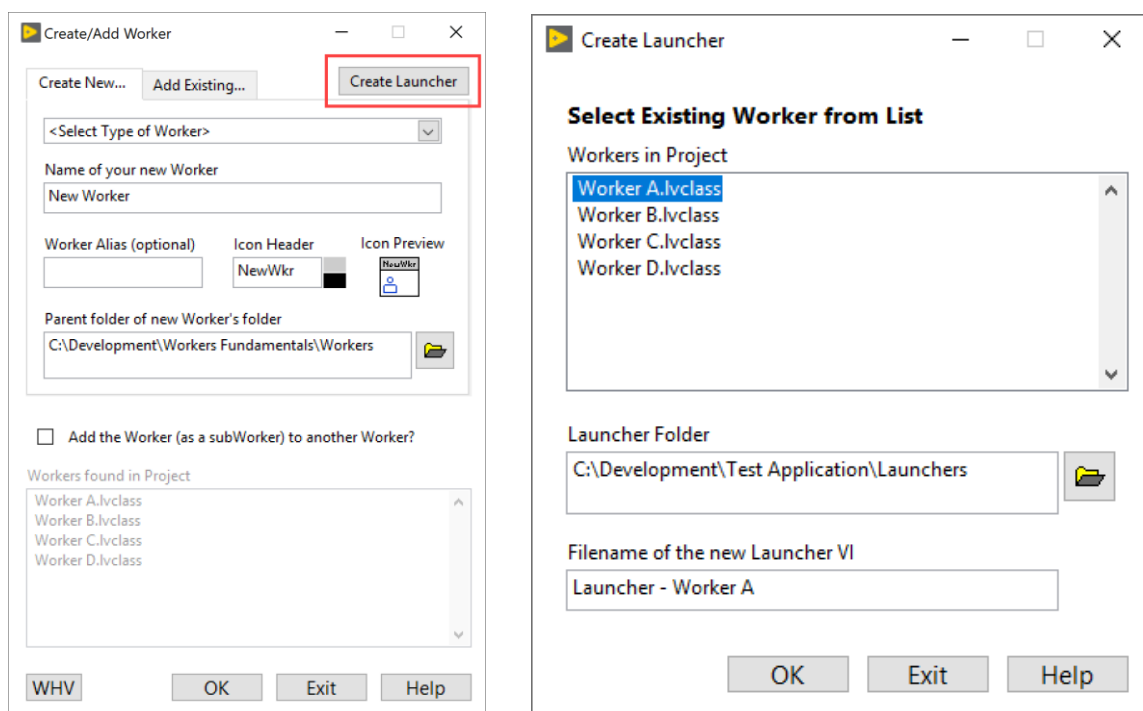


Figure 38 - (left) Create Launcher button in Create/Add Worker tool. (right) Create Launcher tool interface

10 WORKER HIERARCHY VIEWER

The Worker Hierarchy Viewer provides the developer with a high-level overview of all the Workers in an application, by showing their call-chain hierarchy. Similar to the information that is provided in the 'Task Manager' tab of the Debugger, the Worker Hierarchy Viewer provides the development mode Worker hierarchy of both statically linked and dynamically loaded Workers. The Worker Hierarchy Viewer can be accessed from the *Workers™ tools* menu or from the Create/Add Worker tool.

When the Worker Hierarchy Viewer is opened, its window appears as in Figure 39.

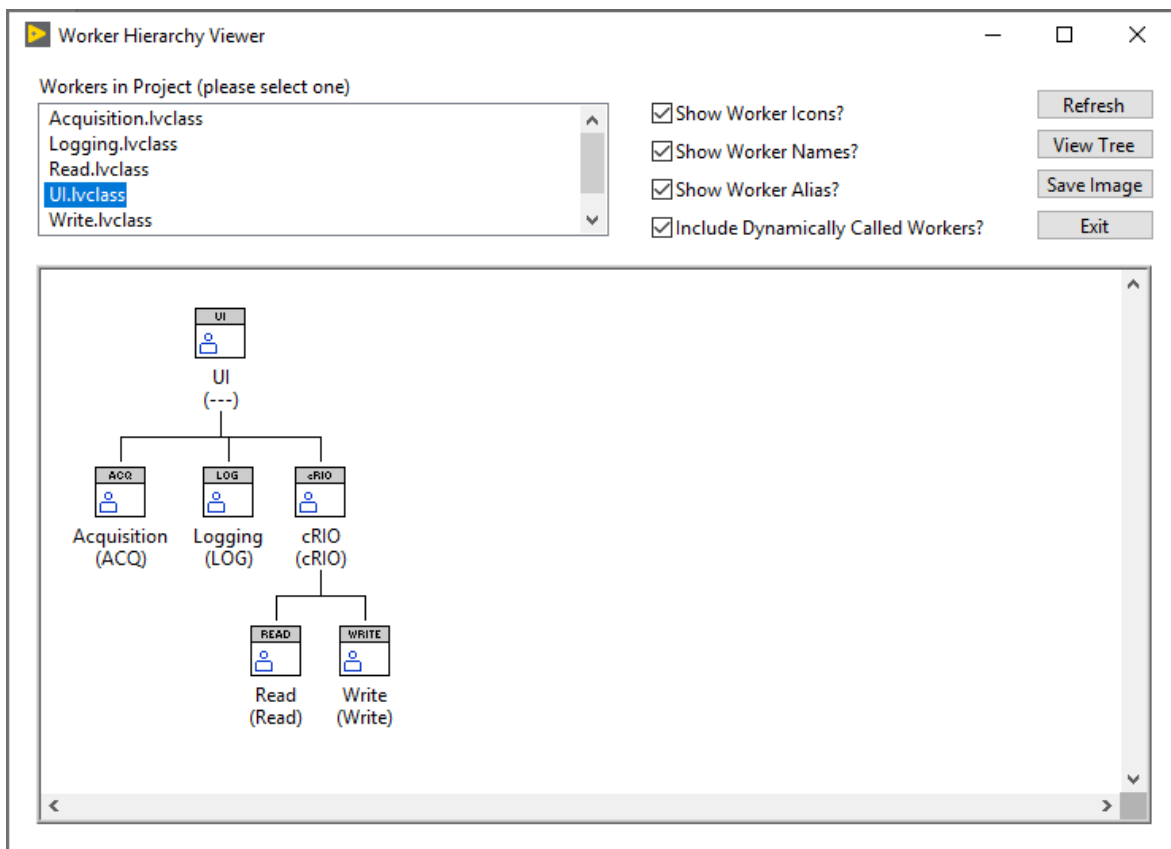


Figure 39. Worker Hierarchy Viewer window

Figure 39 shows the Worker call-chain hierarchy of all the Workers in an application, starting from the Worker that you select. Workers that are loaded dynamically are indicated with a dashed line around the Worker's icon, otherwise the Workers are statically linked. Head Workers do not show their alias.

It is possible to export the diagram created to a PNG, JPEG, or BMP file by use of the 'Save Image' button.

11 CREATING A NEW APPLICATION

11.1 STARTING FROM A BLANK PROJECT

The recommended way to create a new Workers application is by starting from a blank project. After a blank project has been created and saved, you can then use the 'Create/Add Worker' tool (see Section 8) to start adding new Workers to it. The first time you add a Worker to a blank project, the tool will ask you if you also want to create a Launcher VI for this Worker. If you want to create Launcher VIs for any subsequent added Workers, you can do this manually with the 'Create Launcher' tool (see Chapter 9).

11.2 SAMPLE PROJECTS

Two sample projects are provided to help you understand the framework and the usage of the API.

11.2.1 Workers™ Fundamentals Sample Project

The Workers™ Fundamentals sample project demonstrates the basic functionality of the framework, the API and the *Workers™ tools*, and comes with its own Support Guide. This sample project is the recommended starting point for anyone that is new to Workers™.

11.2.2 Workers™ Continuous Measurement and Logging Sample Project

This sample project was created to provide a side-by-side comparison with the original 'Continuous Measurement and Logging' sample project that ships with LabVIEW. It demonstrates how the same functionality is achieved using Workers™ as it is using the classic LabVIEW QMH.

12 ADDITIONAL FEATURES

This section will discuss additional features that have been created to supplement the main features of the framework discussed in this document.

12.1 AUTOMATIC ERROR LOGGING

The error messages that appear in the Workers™ Debugger can be automatically logged to an error log file. This can be done by use of the *Add Error Log.vi* (entirely independent from the Debugger).

The *Add Error Log.vi* must be placed in a Launcher VI, **after** the *Setup Head Worker.vi* and **before** the Worker's *Main.vi*. The correct placement of the VI's is shown in Figure 40.

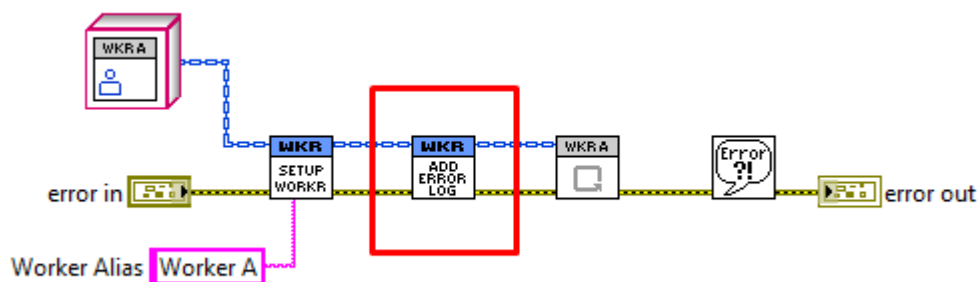


Figure 40. Correct placement of *Add Error Log.vi*

The *Add Error Log.vi* will load the Error Logger dynamically (hidden from the user) before the rest of your Workers™ application is loaded, ensuring that any errors that occur after the *Add Error Log.vi* will be logged. When you exit your application, the Error Logger will automatically be terminated by the framework. Thus, there is nothing more required from the developer than the addition of this single VI as shown in Figure 40.

By default, the error log file will be saved under the project's directory, in a newly created folder called 'Log Files'.

The error log file will be created as a comma-separated text file. The maximum default file size is 1Mbyte. When the file reaches the maximum size, the first half of the errors in the file are deleted, to make room for new errors.

It is possible to change the maximum file size and path (with file name) of the log file. This can be done by two 'write attribute' VIs (available in the Workers™ palette) as shown in Figure 41.

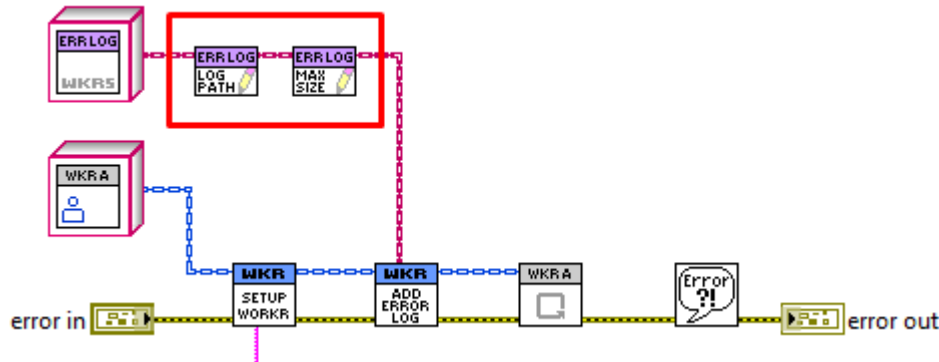


Figure 41. Error Log Write Attribute VIs

12.2 APPLICATION GLOBAL ATTRIBUTES

It is useful to have a single storage location for the sharing of global variables that can be accessible from any Worker, in an instance of any Workers™ application. This feature has been added to the framework via the use of a variant that is accessed by a data value reference, created when the Head Worker is launched. Behind the scenes, the variant uses LabVIEW's 'set/get variant attribute' VIs to write and read data from the variant.

To write data to the variant, use the *Set Application Global Attribute.vi*, shown below.



Figure 42. Set Application Global Attribute.vi

To read data from the variant, use the *Get Application Global Attribute.vi*, shown below.

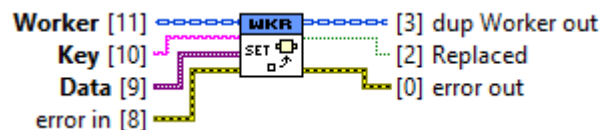


Figure 43. Get Application Global Attribute.vi

These VIs are available in the Workers™ palette. For more information about these VIs, please see their VI context help through LabVIEW.

12.3 CUSTOM WORKER STATUSES IN THE DEBUGGER

While Workers™ provides each Worker with a set of pre-defined statuses in the Debugger's Task Manager tab (described in section 7.1) you can also set custom statuses for your Workers. To set a custom status for a Worker, use the *Update Worker Status.vi* shown below.



Figure 44. Update Worker Status.vi

This VI is available in the Workers™ palette. For more information about this VI, please see its VI context help through LabVIEW.

12.4 CREATING YOUR OWN WORKER TEMPLATES

A Worker template is simple a Worker that contains the extension **template.lvclass* at the end of its filename.

You can create your own Worker templates by creating a new Worker with the 'Create/Add Worker' tool, and then customizing the Worker as you like. Then you can use it to create new Workers based on your own customized template, explained in section 12.5. To create a new template, follow the following steps:

1. Create a blank project and save it.
2. Open the 'Create/Add Worker' tool found under *Tools>>Workers tools*.
3. Select whether you want your template to have an Event Handling Loop or not.
4. Give the Worker a name. Make sure that the Worker's name ends with the word **Template**. This is shown in Figure 45.
5. Press OK.
6. When asked if you want to create a Launcher VI for this Worker, select No.
7. Customize your Worker as you like. You can now use this Worker as a Worker Template.

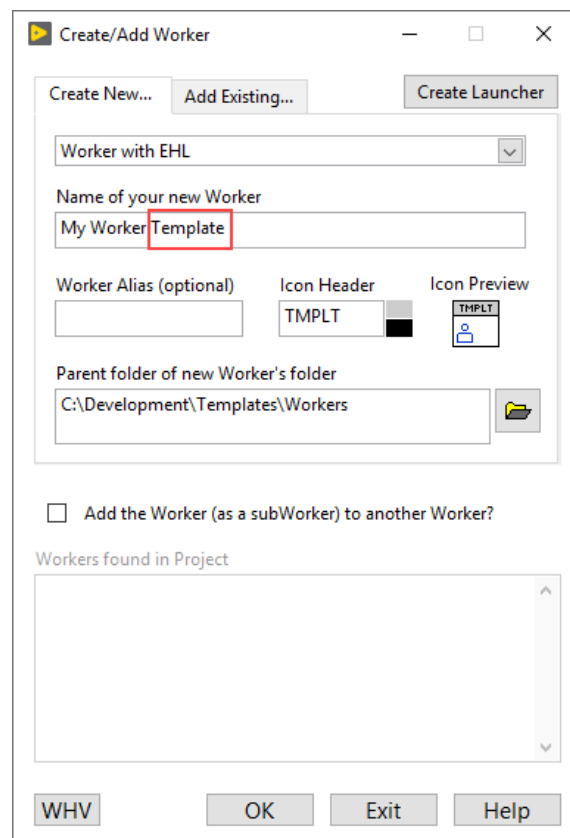


Figure 45 - Creating a new Worker Template

12.5 USING YOUR OWN WORKER TEMPLATES

Now that you have created your own Worker Template as outlined in section 12.4, you can now create new Workers based on your template using the 'Create/Add Worker' tool.

If you choose to create a new Worker from a Worker template, the Create/Add Worker tool will do the following:

1. Create a new Worker (which is a copy of your Worker template).
2. Take the Worker template's icon header colors for your new Worker's icon header.
3. Replace all the template VIs control and indicator labels with the name of your new Worker.

To use your new template, follow the following steps:

1. Create a blank project and save it.
2. Open the 'Create/Add Worker' tool found under *Tools>>Workers tools*.
3. Under <Select Type of Worker> select *Worker from template*, as shown in Figure 46.
4. Select the Worker template you just created in section 12.4, as shown in Figure 46.
5. Proceed as usual with the 'Create/Add Worker' tool. The tool will create a new Worker based on your template and integrate it directly into your Workers application.

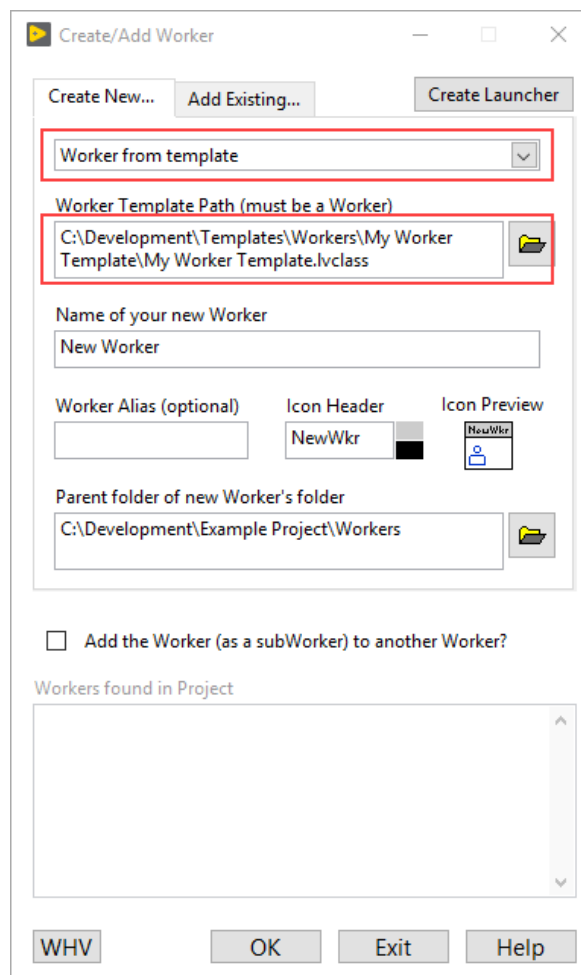


Figure 46 – Creating Workers from your own Worker Templates

13 ADVANCED USERS (LVOOP)

This section contains information for advanced users about the Object-Oriented aspects of the framework and provides information about how the framework functionality can be extended by use of LabVIEW Classes. The classes referred to in this section are all contained within **Workers.lvlib**.

13.1 CLASS INHERITANCE OF WORKERS

Every Worker that is created by the 'Create/Add Worker' tool is a concrete class (**Worker's name**).lvclass that has the following inheritance:

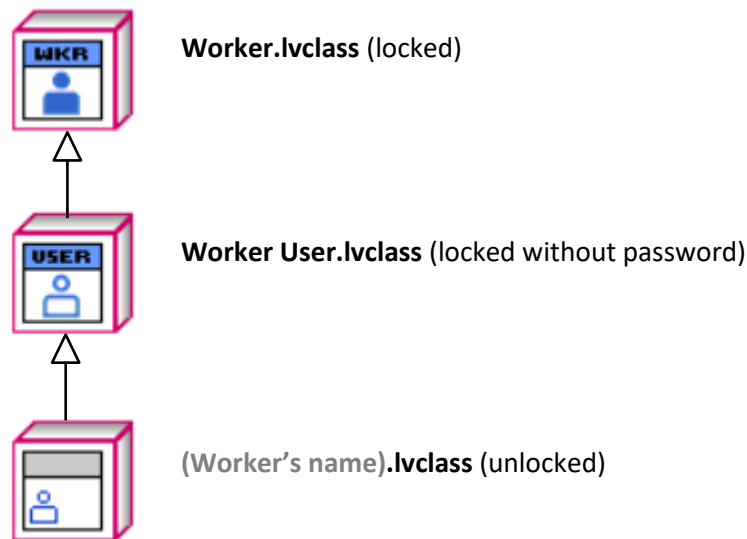


Figure 47. Worker class inheritance hierarchy

The **Worker.lvclass** is a locked class, and contains the methods and data required for the functioning of the framework. The only VI in the class that is designed to be overridden (dynamic dispatch VI) is *Worker.lvclass:Main.vi*.

The **Worker User.lvclass** will be discussed in Section 13.2 below.

The creation of (**Worker's name**).lvclass creates an instance of a Worker through the dynamic overriding of *Worker.lvclass:Main.vi*. It is within this class where all the code for your Worker should be placed.

13.2 WORKER USER CLASS

The **Worker User.lvclass** was created so that you can extend the Workers™ API with your own custom features.

Instead of inheriting from either **(Worker's name).lvclass** or from **Worker.lvclass**, which would create a branch in the Worker inheritance hierarchy, the **Worker User.lvclass** was provided as a parent class to all **(Worker's name).lvclass** classes. This class has been left empty, allowing developers to add methods and data to this class which can be used by all **(Worker's name).lvclass** classes. In this way, libraries of your own 'add-ons' can be created for the framework, added to **Worker User.lvclass** and used by any **(Worker's name).lvclass** without creating branches of Workers each with different features.

Thus, it is recommended to use this class to extend the functionality of the framework instead of creating branches in the Worker inheritance hierarchy (as this would limit your new features only to descendant Workers of your inheritance branch).

13.3 PRIORITY QUEUE

Workers™ has utilized the priority queue of the Actor Framework for the sending of messages between Workers within the framework. The priority queue is contained within **Message Queue.lvclass**.

The **Message Queue.lvclass** is locked without a password. All the VIs contained within **Message Queue.lvclass** are also unlocked. This provides developers with direct access to the priority queue. It is therefore possible for developers to access and modify the priority queue if one wishes to do so. E.g. you can create a VI to query the number of remaining elements in a queue(s).

To provide easy access to the priority queue, the Data Value Reference of the priority queue has been exposed through the **Read Priority Queue.vi** (Figure 48). Through this VI, it is possible to access the contents of the priority queue within your own VIs, external to the **Message Queue.lvclass**. This VI is accessible through the Workers™ palette.

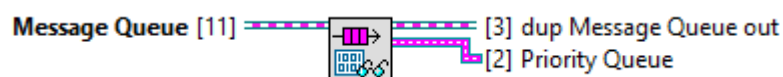


Figure 48. Read Priority Queue.vi

For examples of how the Data Value Reference is used to query/modify the contents of the priority queue, please look at the VIs contained within **Message Queue.lvclass**.

14 APPENDIX A

14.1 ERROR CODES

5000	<p>The <i>Create/Add Worker</i> tool could not complete step 7 of section 8.2.2 (in this document). Therefore, you will need to perform this step manually.</p> <p>This error occurs when the framework VI "Initialized Notify.vi" is not found on the Manager's block diagram. This VI needs to be replaced with "Initialize subWorkers.vi".</p>
5001	<p>The <i>Create/Add Worker</i> tool could not complete step 9 of section 8.2.2 (in this document). Therefore, you will need to perform this step manually.</p> <p>This error occurs when the framework VI "Exit.vi" is not found on the Manager's block diagram. This VI needs to be replaced with "Start Exiting subWorkers.vi".</p>

www.workersforlabview.com