

Remote Module Installation Guide

October 8, 2005

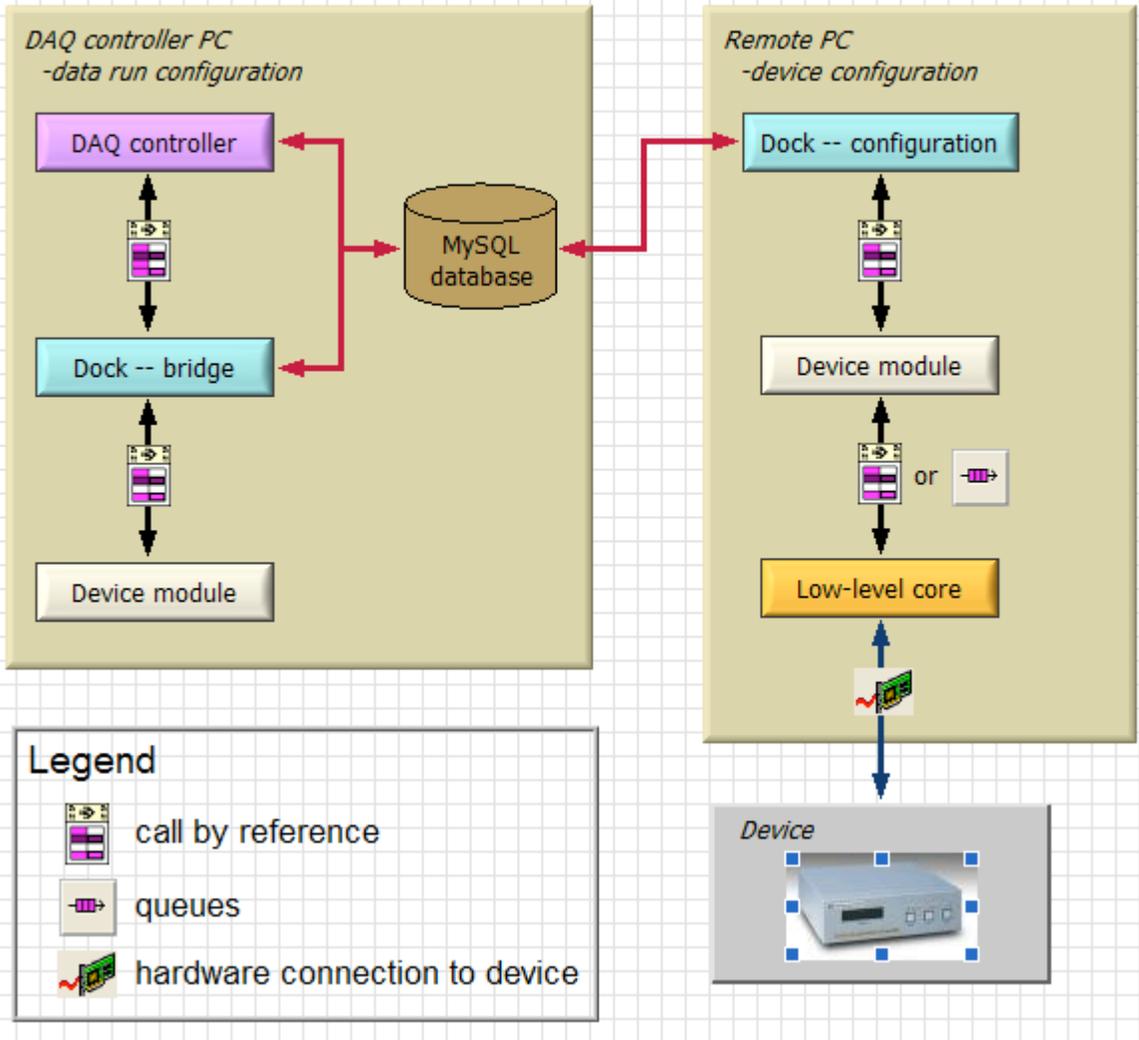
Overview

The “*Remote module dock*” is a LabVIEW-based software package that is designed to allow end users to integrate new devices into the *ACQ II* data acquisition system. This guide describes the process of installing a remote device module and is intended for those end users having at least moderate proficiency in LabVIEW. The goal is to minimize the amount of time and effort required to add devices, as well as to shield the end user from the details of the *ACQ II* system. Functionality common to all device modules has been generalized and collected into the *Remote module dock*. This includes all database-related activity, communication with the *DAQ controller*, and reformatting of data and meta-data (such as for HDF5 conversion). The functionality which the end user must provide is limited to the low-level control of the device and the management of the device configuration and run-time state. These end user responsibilities are discussed in detail below, but first it is necessary to know something about the overall architecture of the *Remote module dock*.

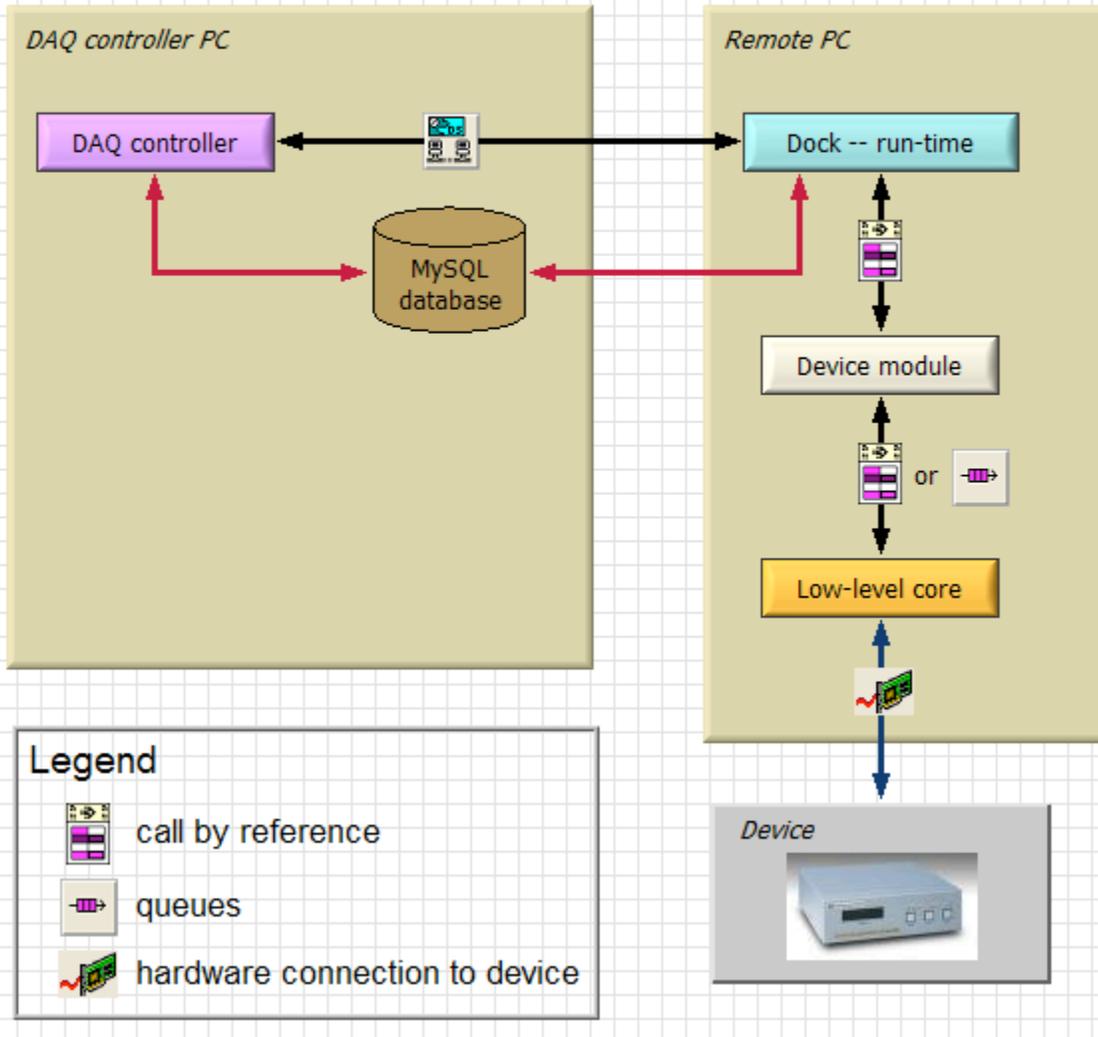
The architecture breaks into two parts, configuration-time and run-time, as shown in the schematic drawings on the next two pages. The existing *DAQ controller* is shown in purple. The three main components of the *Dock*, “*Dock – bridge*”, “*Dock – configuration*”, and “*Dock – run-time*” are shown in blue. At configuration-time data run configuration and device configuration are done independently on the DAQ controller PC and the remote device PC, respectively. On the DAQ controller PC, device related information is provided by the *Dock – bridge*. At configuration-time there is no direct communication between these two PC’s. However, at run-time the *DAQ controller* communicates via DataSockets to the *Dock – run-time* VI. The *Remote module dock* supports one device per PC at the moment. Once it is well-established in this mode, it could be modified to handle multiple devices per PC with a small amount of effort.

The remaining “*Device module*” and “*Low-level core*” parts of the architecture are to be provided by the end user (see detailed instructions below). In general, it would be best to adhere to a convention of naming the *Device module* “<device>.vi” and the *Low-level core* “<device> core.vi”. Note that there is no database access from the *Device module*. This frees the end user from any database-related coding. The database is used to store device configuration data and the related custom database I/O code has proved to be a bottleneck in implementing previous device modules. This bottleneck has now been eliminated in the *Remote module dock* by using a general approach whereby the database tables and I/O are defined by the structure of a LabVIEW data element (typically a cluster). Thus, the *Device module* simply has to pass the device configuration, specified in a LabVIEW cluster, to the *Dock* which then handles any database operation without needing to know the details of the configuration. Another point to note, communication between the *Dock* and the *Device module* is by a call by reference node. This allows the *Dock* to call any device module via a standard interface, using a well-defined request protocol.

Remote module dock: configuration-time architecture



Remote module dock: run-time architecture



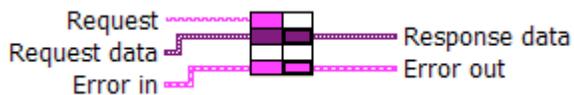
1. Device module

Device module

The best place to start is with the *Device module*. Create a new folder under “ACQ II home\Modules”. The folder should be named after the device. Either copy an existing *Device module* or start a new VI here, again named after the device. Working from a copy of an existing *Device module* will greatly reduce the amount of time required for many of the steps below. For example, replacing the configuration and run-time state clusters with the new ones will save a considerable amount of reconnecting wires.

1.1) Configure the connector pane

Ensure that the connector pane is configured as follows:



Connector pane inputs and outputs

	Request	input	string
	Request data	input	variant
	Response data	output	variant
	Error in	input	error cluster
	Error out	output	error cluster

The basic idea is that the *Device module* is called with a Request and optional Request data, performs the appropriate action, and optionally returns Response data.

1.2) Design the configuration cluster

All of the configuration parameters for the device must be contained within a single configuration cluster. It should be saved as a LabVIEW custom control of the type “Strict Type Def”. An instance of the configuration cluster should be placed, as a control, in the display section of the *Device module*. It is convenient to make the display section be one page of a tab control. The configuration cluster will be used by the end user when interactively changing the device configuration (see the *Configure* request). Since it also determines the structure of the database tables used to store the device configuration, it is subject to some restrictions related to data types and naming.

Valid data types

- boolean
- unsigned (8-, 16-, and 32-bit)
- enum unsigned (8-, 16-, and 32-bit)
- integer (8-, 16-, and 32-bit)
- single, double, extended,
- string
- path
- cluster – may contain any of these data types
- array – may contain any of these data types except cluster and array

Note that while arrays are not allowed to contain arrays or clusters, multi-dimensional arrays are allowed.

Naming conventions

The naming of the configuration cluster and the data elements within determines the naming of corresponding elements in the database so it is important to choose these names carefully.

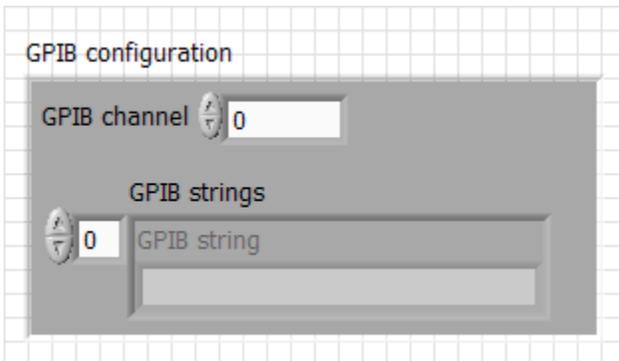
- Names may contain alphanumeric characters as well as “-“, “_“, and “.“.
- The name of the configuration cluster should be “<device name> configuration”, *i.e.* the device name followed by “ configuration”.
- All clusters and arrays within the configuration cluster become separate tables in the database and so their names should begin with “<device name>”. This way all the tables related to the device will be listed consecutively when viewed with a database browser.

Completeness

If possible, identify all necessary configuration parameters from the start. Adding more parameters later is not impossible but the *Remote module dock* will not handle this kind of change automatically; custom intervention will be necessary. Also, do not include the configuration name as part of the configuration cluster; the *Remote module dock* automatically adds the configuration name as a primary key.

Example

Throughout this document I will be using the GPIB interface as an example. Thus, its configuration cluster would be:



Two database tables would be produced:

gpib_configuration

configuration_name	varchar(120)	primary key
gpib_channel	smallint	

gpib_strings

gpib_strings_index0	int	primary key
configuration_name	varchar(120)	primary key
gpib_string	text	

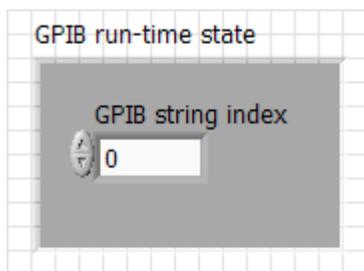
Note that the end user need not be aware of these tables at all. They are completely equivalent to the configuration cluster itself.

1.3) Identify the device-specific run-time state parameters

The run-time state of a device is the minimum set of parameters which completely defines the state of the device at a moment in time. In general, the state of the machine is defined by the configuration. However, in some cases, there are items that can change within a given configuration. These are referred to as the “device-specific run-time state parameters”. A typical example would be the index into an array contained within the configuration. Collect all such device-specific parameters into a cluster and place it in the internal data section of the *Device module*. The name of this cluster does not matter because the *Dock* automatically takes its data elements and adds them to the full run-time state cluster. If there are no device-specific parameters, no cluster is necessary. If there is only a single parameter, it does not have to be placed within a cluster.

Example

The GPIB interface device-specific run-time state would be:



1.4) Setup block diagram

The *Device module* must handle a certain number of standard requests. Also, it must handle at least one request specific to the device (otherwise the module would accomplish nothing). In fact, the list of requests defines the functionality of the *Device module*.

Request list

Get configuration

- Returns the configuration cluster.

Load configuration

- Replaces the internal configuration cluster with the input.

Configure

- Allows the user to interactively modify the device configuration, typically by actually operating the device. Returns the configuration cluster.

Get run-time state

- Returns a cluster or simple data element which represents all device-specific run-time state parameters.

Get device-specific requests

- Returns a list of device-specific requests along with their key-value pairs, used for creating a data run sequence.

Get integer names

- Returns a list of the names of integers available using the *Get integer* request (see below).

Get integer

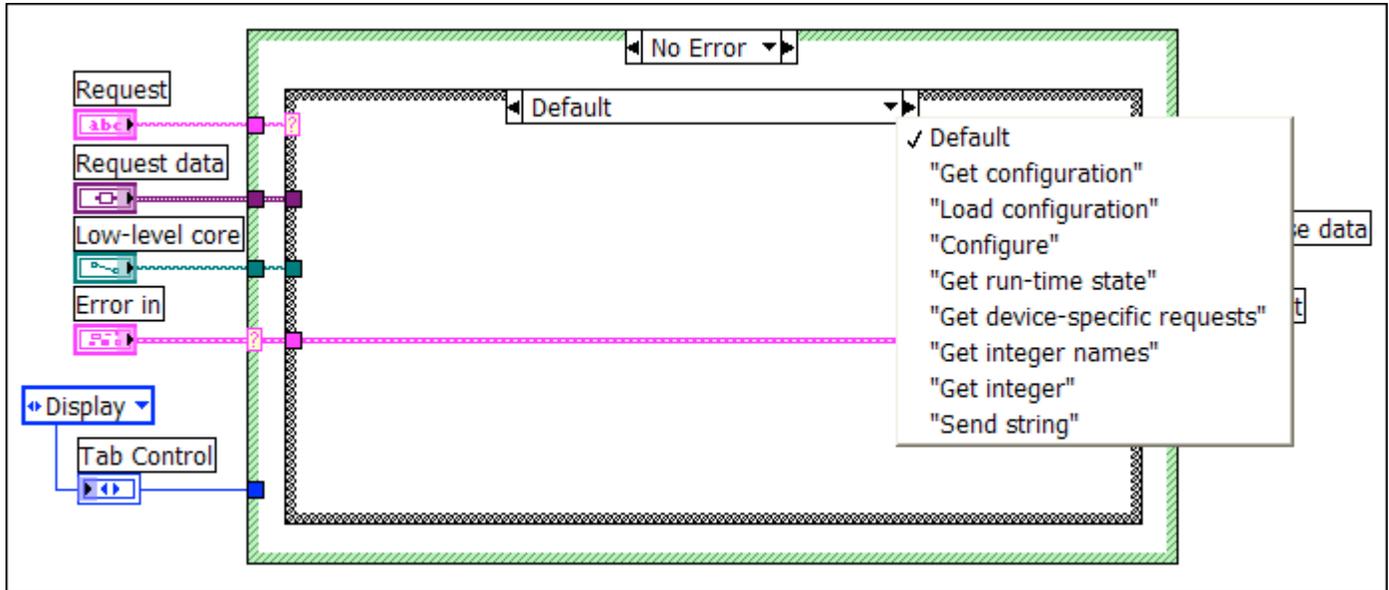
- Returns the value of the integer specified in the Request data. This would typically be the index of the last element in an array; for example, the index of the last GPIB command string in a list.

<Device-specific requests>

- Causes the device to perform one of its defined run-time actions. Response data must not be returned.

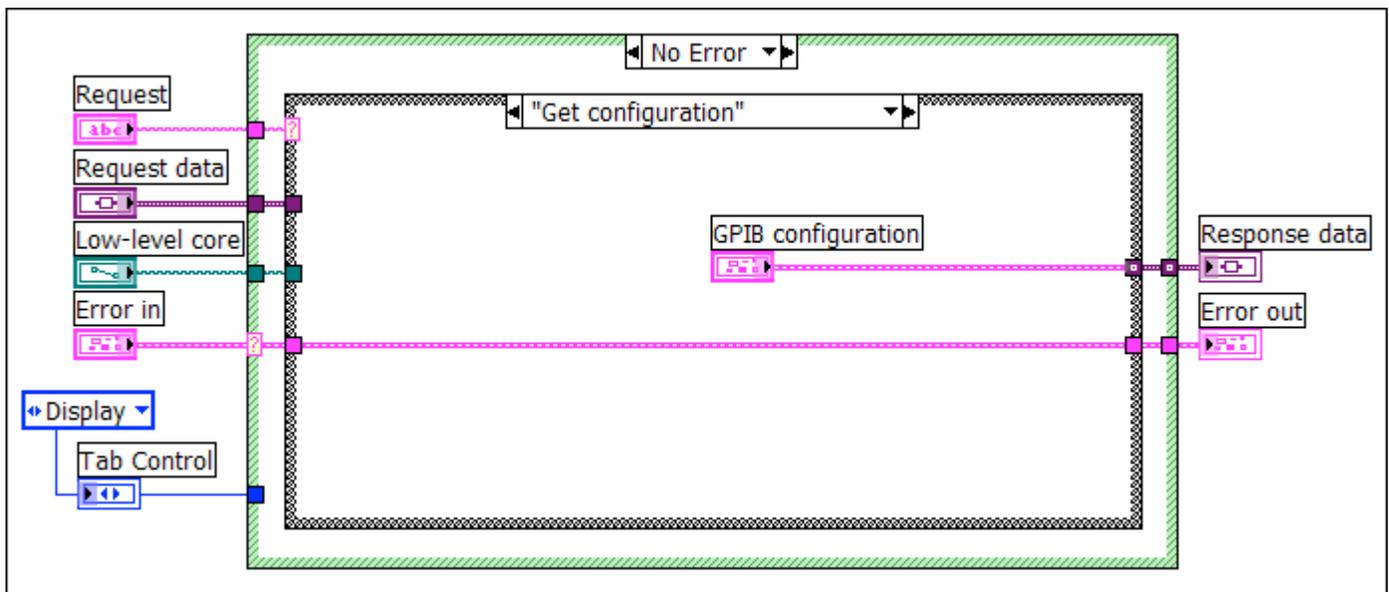
Block diagram structure

A simple way to implement the *Device module* is to use a case structure with the case selector wired to the Request input. Note that the Request case structure is executed only when there is no error. Here is what the block diagram for the GPIB interface would look like, with the case selector values highlighted.



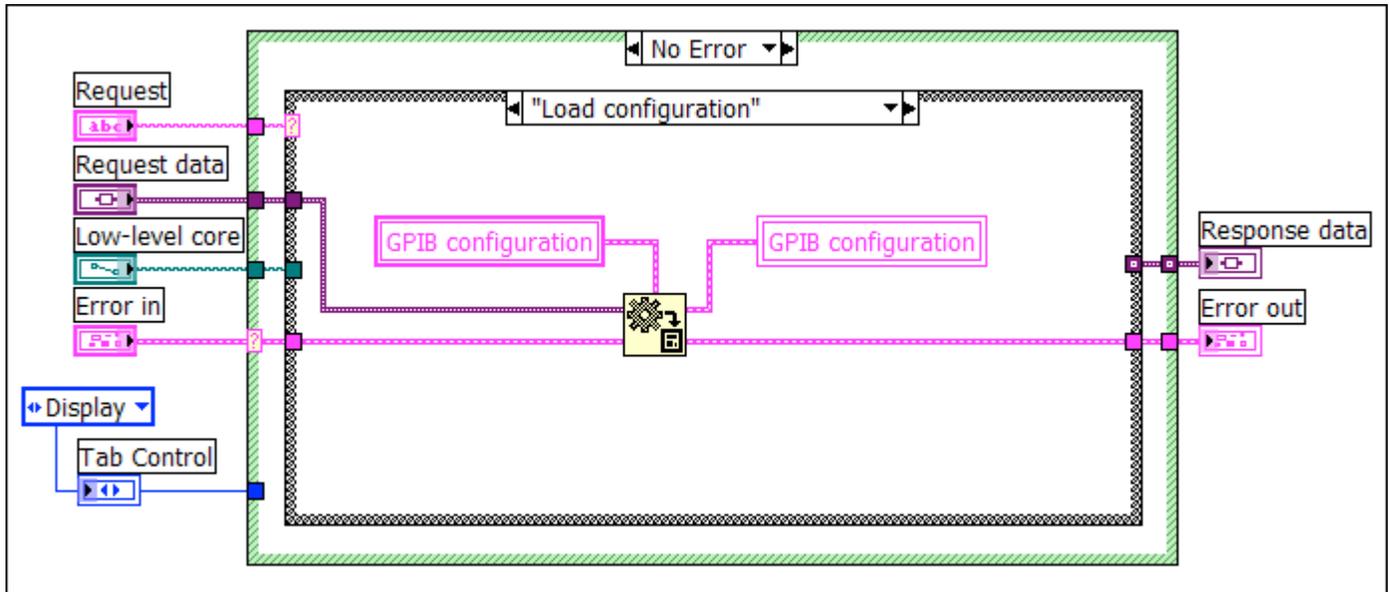
1.5) Implement "Get configuration" request

If the device hardware is capable of storing configuration parameters, then the best approach is to use it as the prime repository of the configuration. In this case the configuration must be read back from the device, copied into the configuration cluster, and returned via Response data. If the device does not store configuration, as in the GPIB example, then the configuration cluster in the *Device module* is the prime repository and is simply returned via Response data.



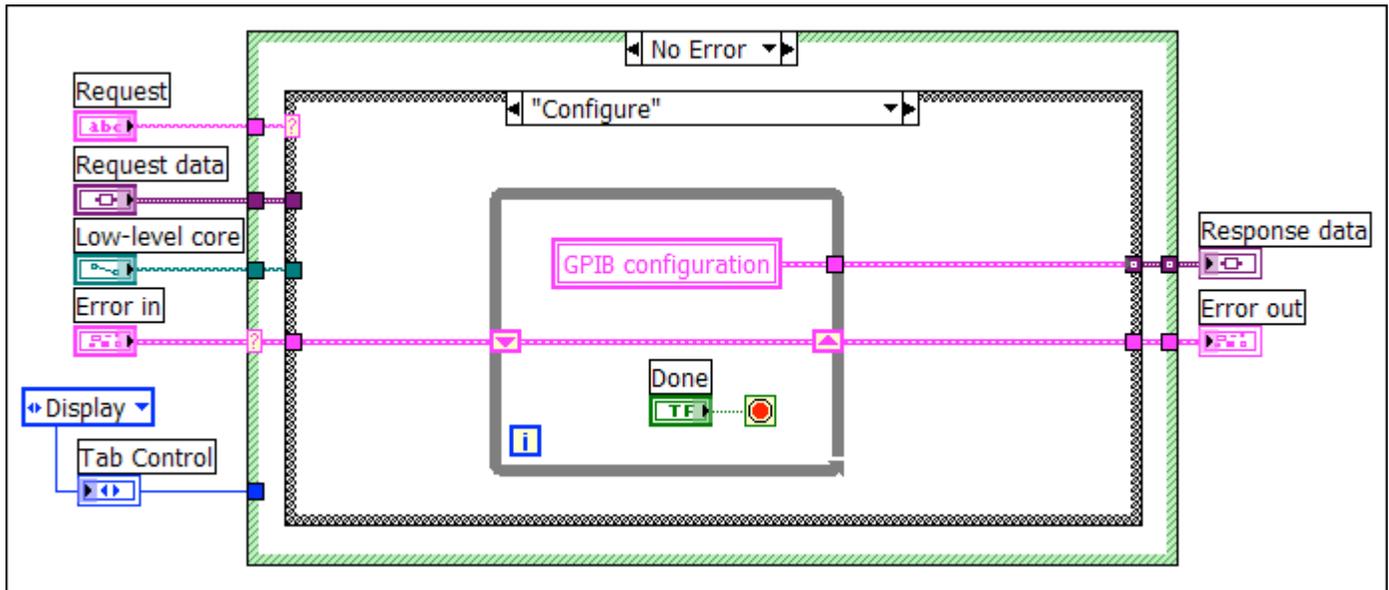
1.6) Implement “Load configuration” request

Use the Variant to data VI to translate the Request data into the configuration cluster. Then load the configuration into the device, if necessary.



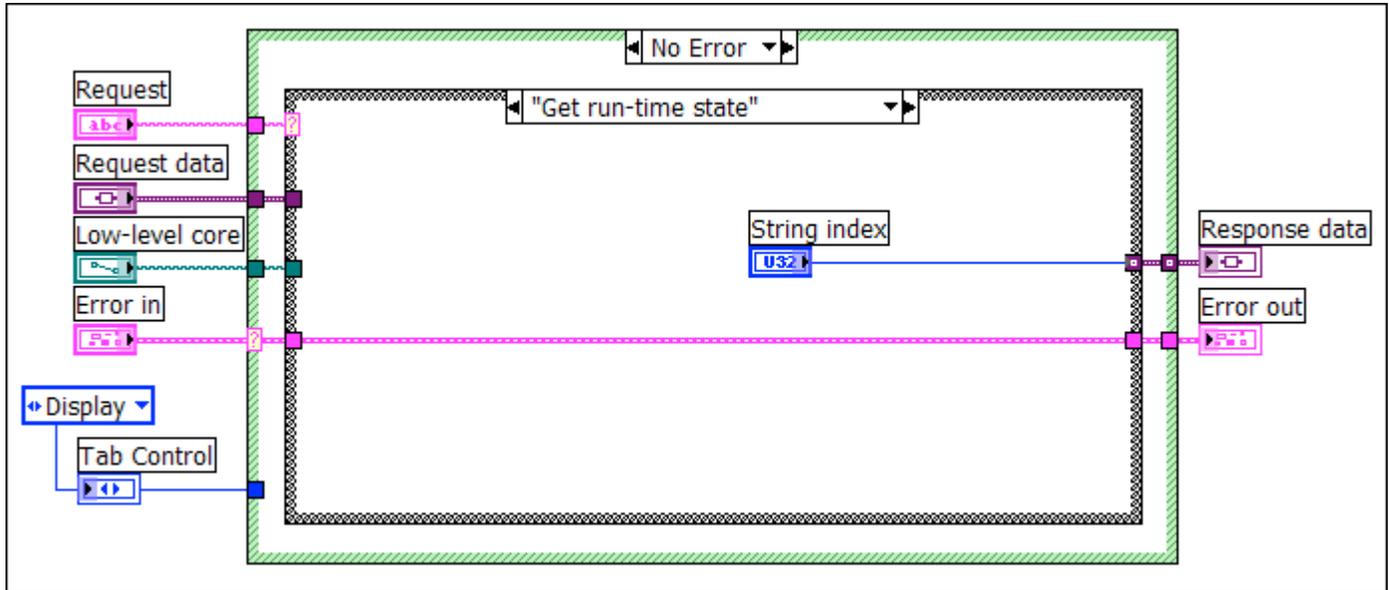
1.7) Implement “Configure” request

With the *Configure* request, the *Device module* is expected to enter an interactive mode where the end user can modify the configuration and interact with the device. Note that the *Dock – configuration* VI takes care of showing and hiding the *Device module* front panel before and after the call. All other user interface details are the responsibility of the *Device module*. The implementation of this request is where the majority of the code for the *Device module* will be. All the rules regarding valid configuration parameters will need to be enforced here and all device functionality should be exercised here as well. All configuration changes should be transmitted to the device as they are made, if appropriate. (Also see “2. Low-level core”.)



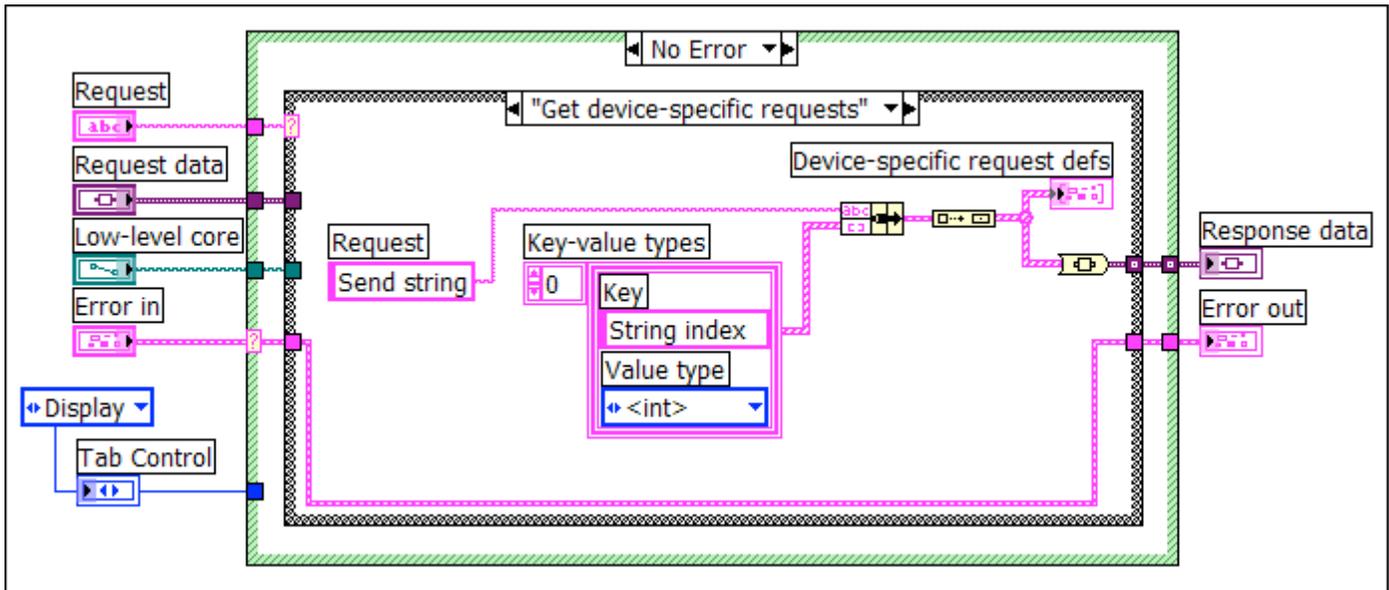
1.8) Implement "Get run-time state" request

Return the run-time state data element in Response data. Clearly, at run-time, this data element must be kept current. For most devices, the prime repository for the run-time state will be the *Device module* but it is conceivable that it be stored in the device itself.



1.9) Implement "Get device-specific requests" request

In order for a remote device to be used in a data run, its run-time request syntax must be published. At the time of data run sequence creation, the *DAQ controller* requests, via the *Dock -- bridge*, the "device-specific requests" and "integer names" from each module. These are formatted, along with the standard requests, into a comprehensive syntax. Thus, the first step is to identify all device-specific requests. They correspond to all the well-defined actions of the device that could be requested during a data run. Typically there will only be one or two. In the case of the GPIB interface, there is only the "Send string" request. This request has the key



“String index” whose value is of type “<int>”. The next step is to proceed, as in the block diagram (above), to package this information, for each request, into the Device-specific request defs data structure. If the request has no key-value pairs associated with it, just use an empty Key-value types array. A typedef of the data structure can be found in “Remote module dock\Typedefs\Device-specific request defs.ctl”.

Device-specific request def cluster

The request definition cluster is summarized as follows:

Cluster of 2 elements

Request (string)

Key-value types (1-D array of)

Key-value type (cluster of 2 elements)

Key (string)

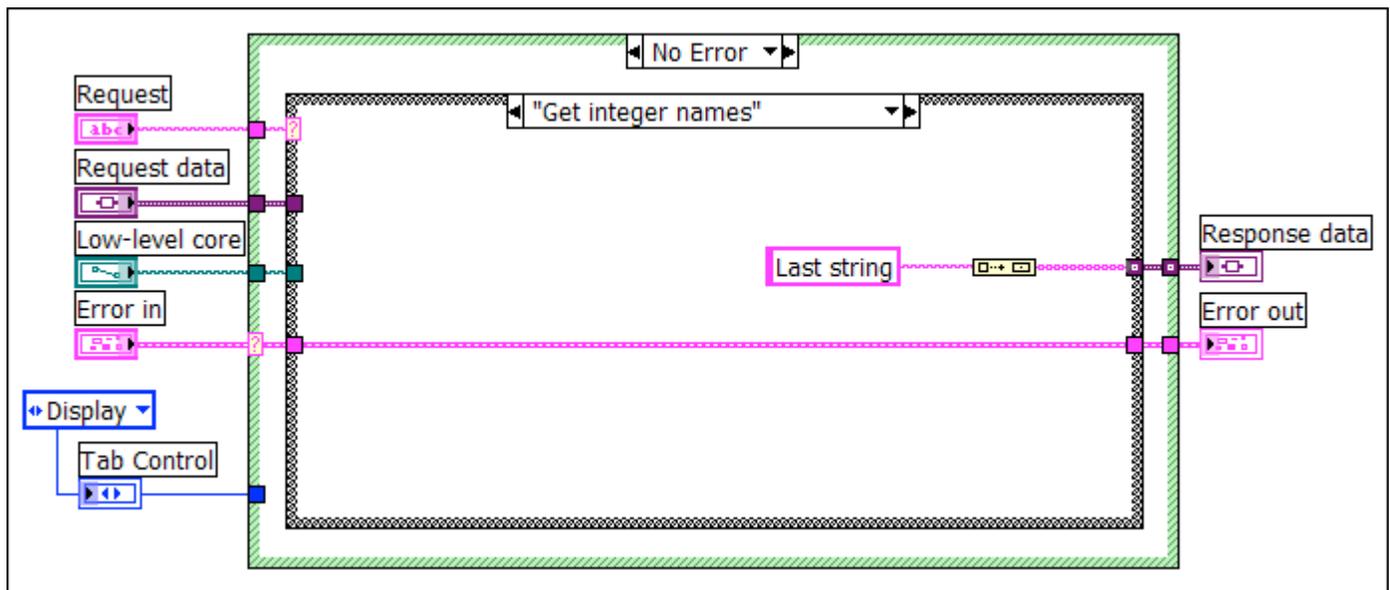
Value type (enum {“<string>”, “<int>”})

1.10) Implement “Get integer names” request

Identify what integers will be necessary for use in a data run sequence. Currently, the data run sequence scripting language supports two uses for these device-specific integers. First, a script variable can be set to the value of the integer; this variable can subsequently be incremented or passed to any device via a key-value pair of a request. Second, the integer can be used as the start, stop, or increment value in a for loop. The for loop stop value is, by far, the most common role of device-specific integers. For example:

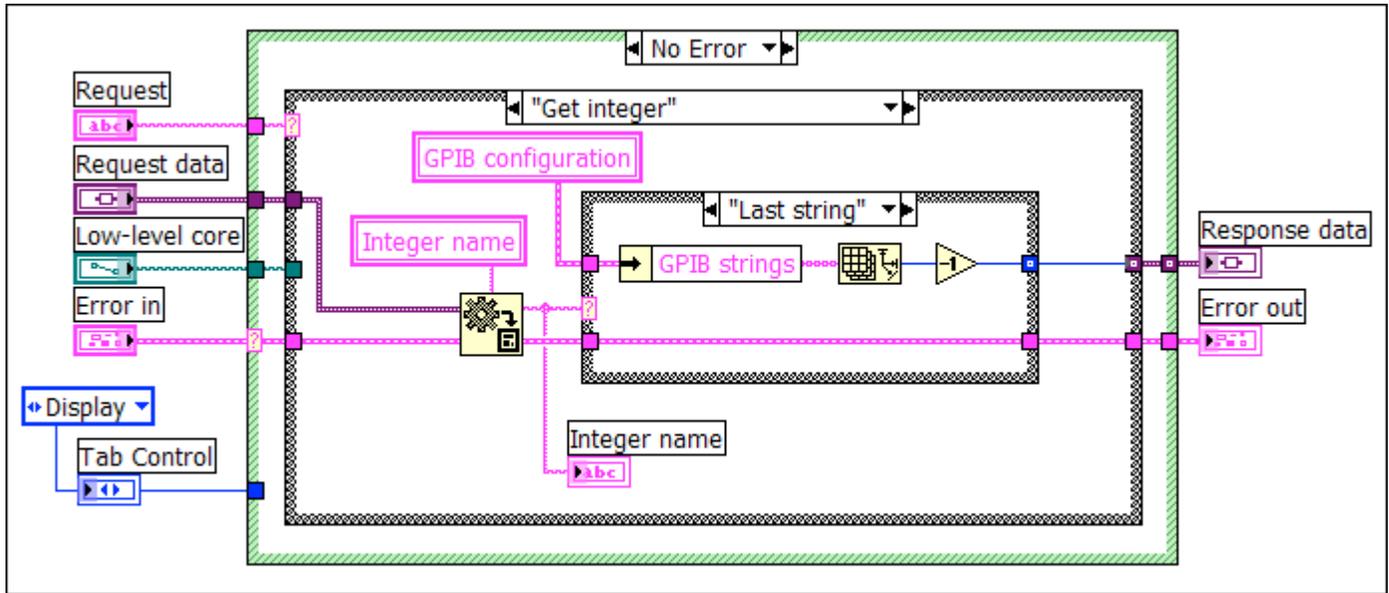
For @i: 0 | GPIB interface:Last string

Build the integer name strings into an array and return the array via Response data. If there are no integers, return a zero-length array of strings.



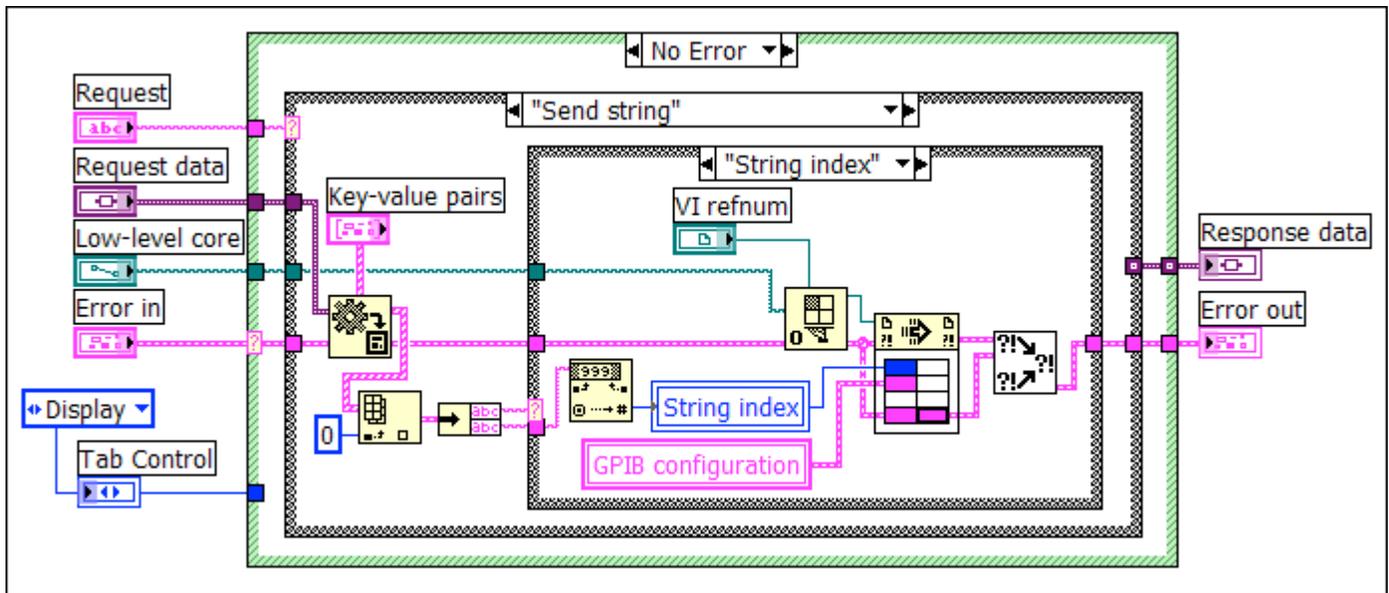
1.11) Implement "Get integer" request

Use the Variant to Data VI to convert the Request data to a string representing the integer name. Connect the integer name to the selector of a case structure and return the corresponding integer. In the example, the "Last string" integer is simply: $\text{length}(\text{GPIB string array}) - 1$.



1.12) Implement "<Device-specific requests>"

It is not possible to discuss the implementation of a device-specific request in details as it depends on what the request is, but some general points can be noted. If the request has any key-value pairs associated with it, they will be passed in as Request data.



The Key-value pairs data structure is a:

1-D array of
cluster of 2 elements -> { Key (string), Value (string) }

In the example, the single key is verified to be “String index” and the value string is converted to an integer before being passed to the *Low-level core*. If there are multiple key-value pairs, parsing them becomes a little more involved. Communication with the *Low-level core* is by call by reference, though other methods may be used, as long as the *Low-level core* is not called directly (see “2. Low-level core” for more on communication). Finally, note that Response data is never returned for device-specific requests. These requests are always performed at run-time and the data run sequencer has no way of handling feedback, other than in the case of an error.

1.13) (Optional) Edit VI icon

If desired, change the *Device module* VI icon to something appropriate for the device. This icon will appear in the upper left hand corner of the *Dock – configuration* VI.

Finally, please note that the example *Device module* and *Low-level core* VI’s, from which the screen shots were taken, are included with the installation guide as “Example device module.vi” and “Example low-level core.vi”, respectively.

2. Low-level core

Low-level core

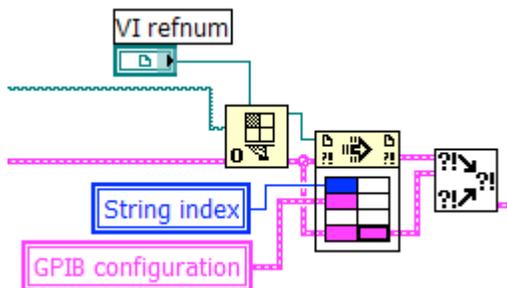
The *Low-level core* should reside under the same folder created for the *Device module*. Before beginning to write or modify any code for the *Core*, it is necessary to consider 1) its form or architecture and 2) the method of communication between it and the *Device module*. Considering communication first, there are two standard methods in LabVIEW that could be used: call by reference and queues.

2.1) Communication methods

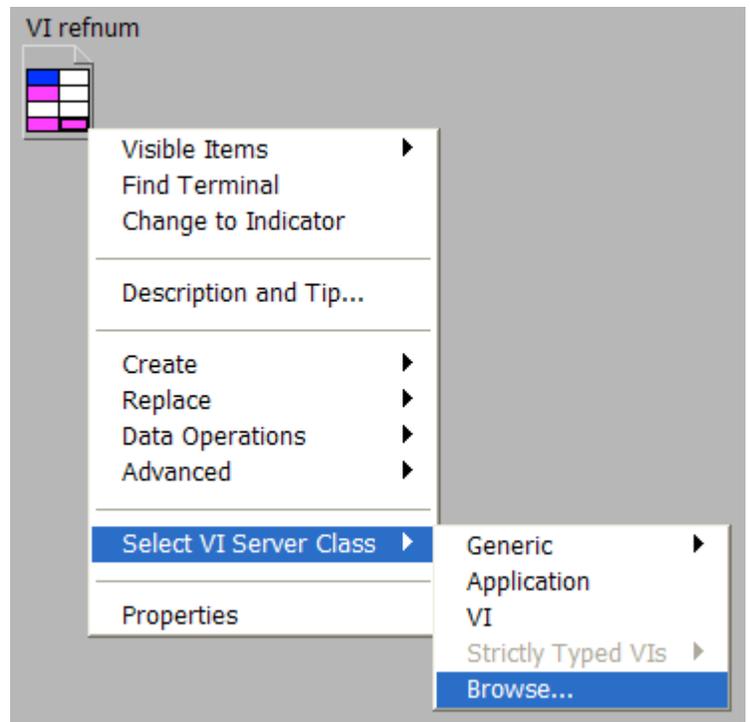
Call by reference

The simplest communication method is call by reference. If there is a choice between the *Core* being callable or free-standing, then it is preferable to have it callable because there is a performance penalty in LabVIEW for running multiple VI's simultaneously. Use a Call by Reference Node, which is fairly straightforward to code except for obtaining the VI reference. For this, place an Open VI Reference VI on the block diagram, right-click the type specifier VI Refnum input and create a control. Then on the front panel, right-click the newly-created VI refnum control and choose **Select VI Server Class >> Browse...**, then browse to the *Low-level core* VI. The purpose of all this is to inform the Call by Reference Node of the connector pane of the *Low-level core*. Finally connect the inputs and outputs as you would with a regular VI. Note that the two error outputs coming from the call by reference are merged.

Block diagram



Front panel

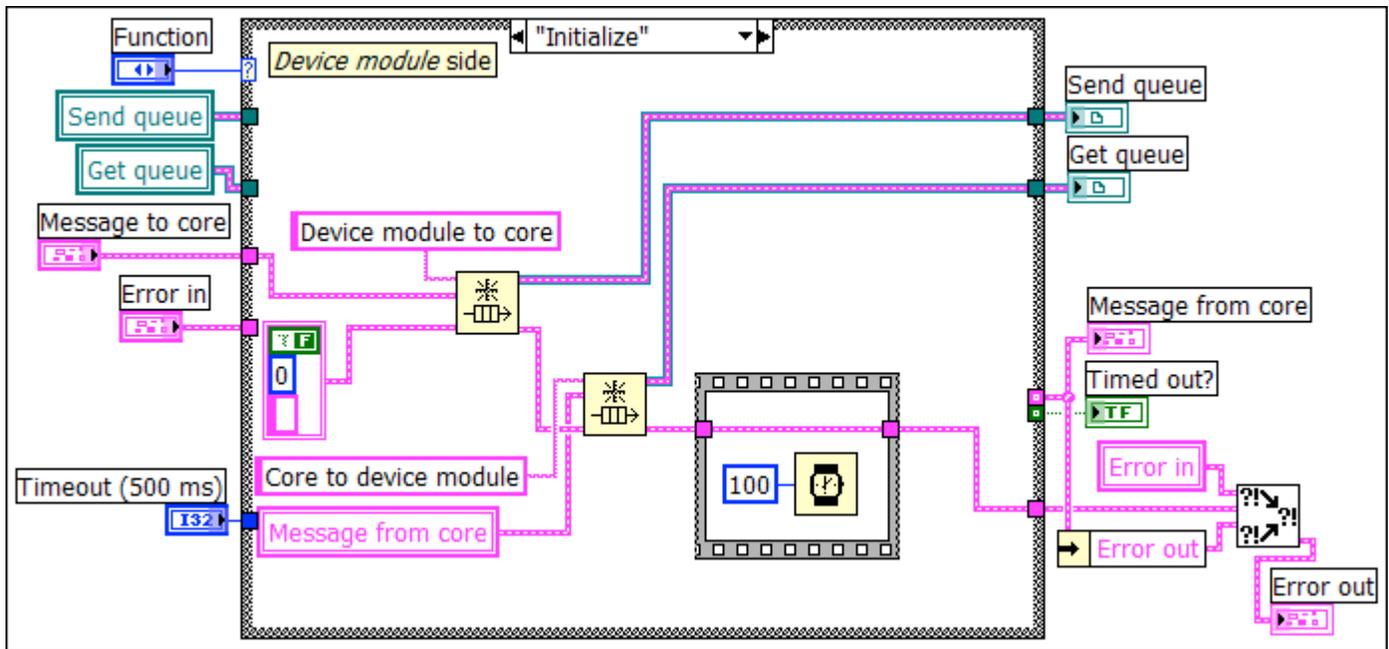


Queues

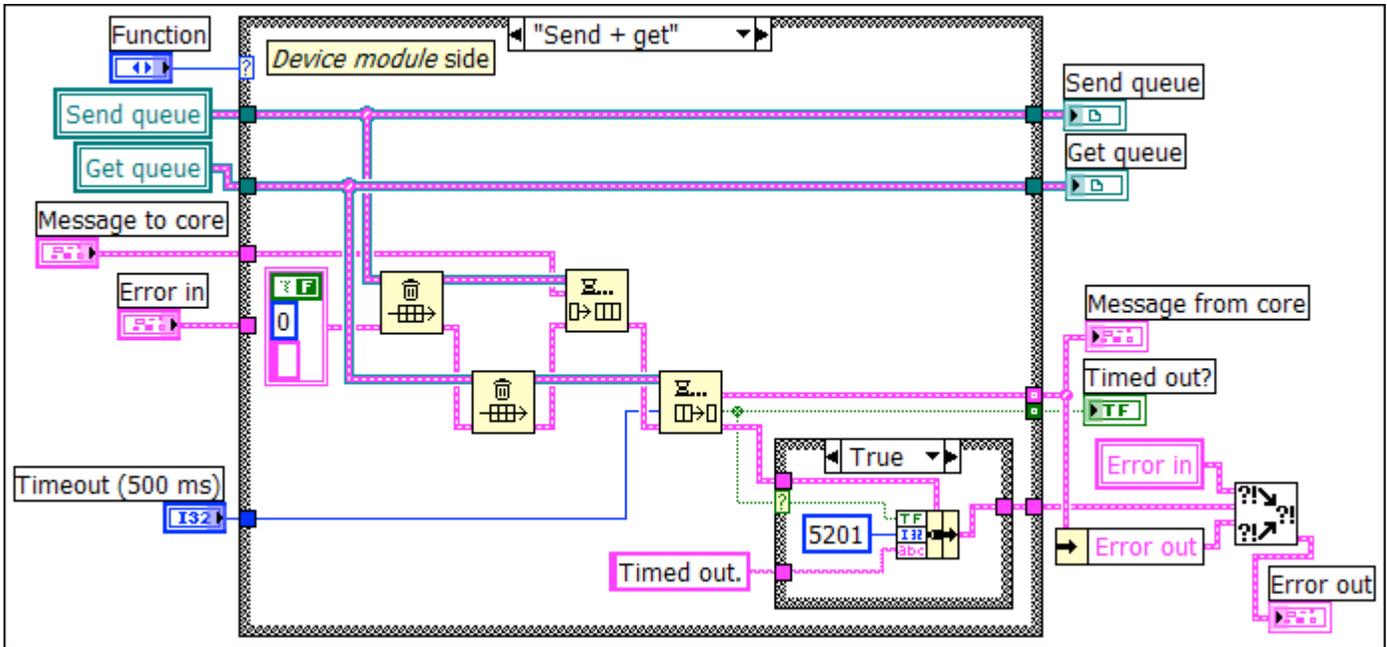
If the *Low-level core* must be a free-standing application that runs independently of the *Device module*, then the best choice of communication is by queues. Note the *Device module* and the *Core* must run on the same machine. It is worth encapsulating the details of the queue communication in a pair of VI's; one for the *Device module* and one for the *Core*. I suggest implementations for these as follows.

Queue communication VI (*Device module side*)

This VI has two functions: "Initialize" and "Send + get". The first, "Initialize", merely creates two queues: one for sending messages to the core and one for getting messages back from the core. This function should not be called repeatedly as it will cause a memory leak. The Send queue is initialized for Message to core, which is a cluster of two elements: { Message (string), Message data (variant) }. The Get queue is initialized for Message from core, which is a cluster of three elements: { Message (string), Message data (variant), Error out (standard error cluster) }. Note that the names of the queues, *i.e.* "Device module to core" and "Core to device module", must be unique on any given PC. Obviously, the actual name of the device should be substituted for "Device module".



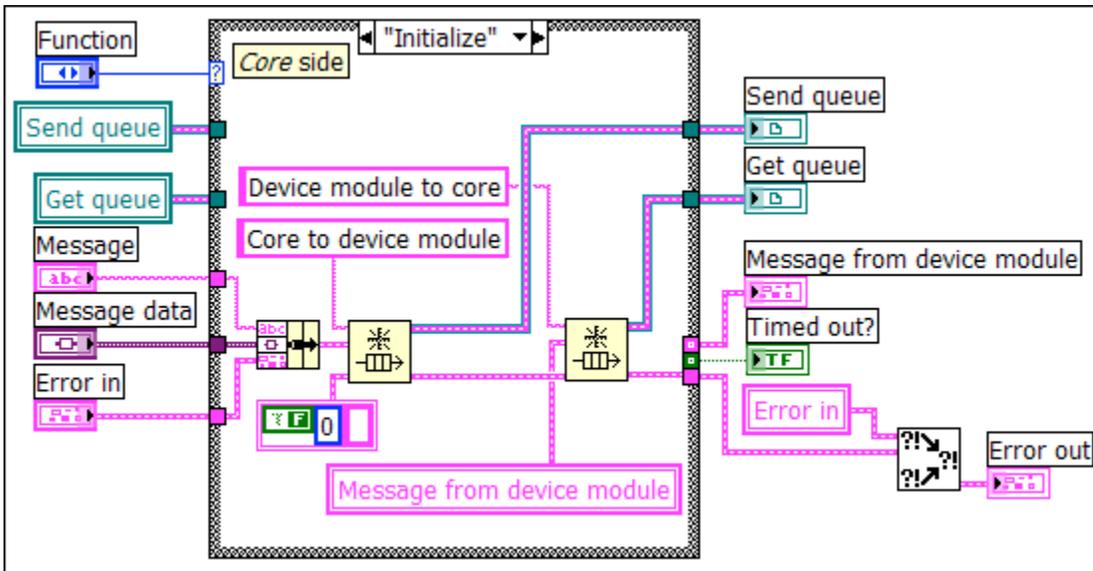
The "Send + get" function (shown on the next page) first clears both the Send and Get queues, then it puts the Message to core on the Send queue and waits for a message back from the *Core* on the Get queue. The message back must occur within the specified timeout period.



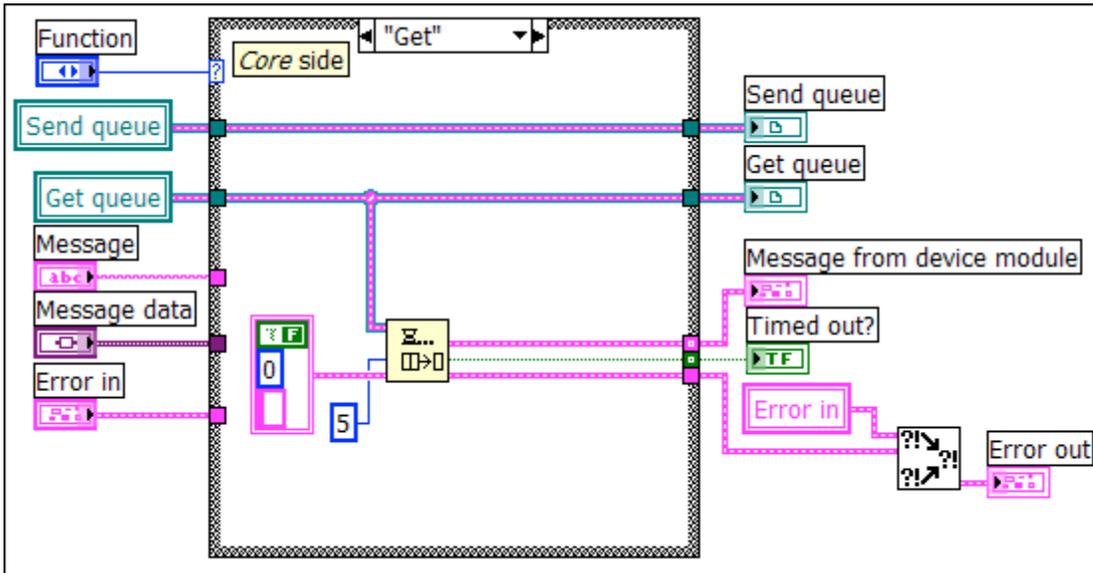
Note that both the “Initialize” and the “Send + get” functions execute regardless of the state of the Error in cluster; however, Error in is merged with any error from queue operations and the error returned by the Core and returned in Error out.

Queue communication VI (Core side)

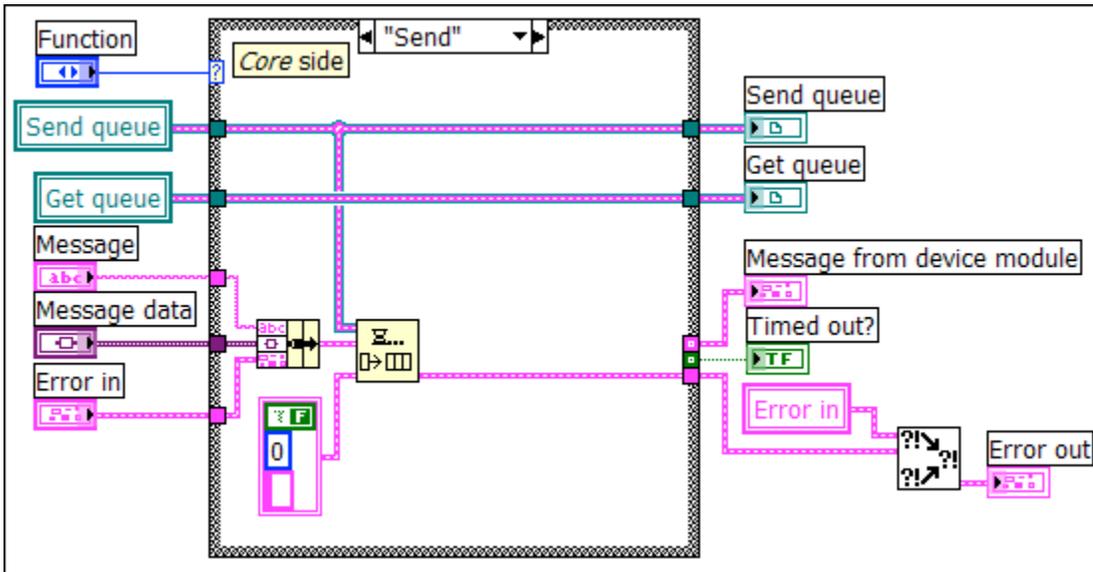
The Core side VI has three functions: “Initialize”, “Send”, and “Get”. The “Initialize” is the same as for the Device module side except that it does not create the queues; instead, it merely obtains references to them. Of course, what the Device module calls the Send queue, the Core calls the Get queue, and vice versa.



The “Get” function is simply an attempt to read the Get queue with a short timeout. If there is no message it returns. Timing out is not an error for this function.



The “Send” function is equally simple. The Message, Message data, and Error in are bundled into a cluster and put onto the Send queue.



Using the queue communication VI's

On the *Device module* side, the “Send + get” function is the equivalent of a call by reference. On the *Core* side, the “Get” function needs to be repeatedly called to see if there is anything on the queue. If there is, the *Core* should fulfill the request and send a message back as soon as possible

Both of the queue communication VI's described above are included with this installation guide as “Queue example (device module side).vi” and “Queue example (core side).vi”.

2.2) Low-level core architectures

There are virtually no restrictions as to what the *Low-level core* code can be. The three most likely possibilities are 1) a simple wrapper VI for a collection of low-level calls or API's provided with the device, 2) a fully functional application provided with the device or found through an Internet search, and 3) a separate system such as the Housekeeper which is not merely a device to be used in a data run but has other responsibilities and typically runs all the time. I will discuss these three in general but clearly cannot provide detailed coding examples because the final form of the *Low-level core* is so specific to each particular device.

Simple wrapper VI

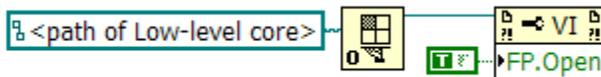
The simple wrapper VI is a callable VI and the communication method is, of course, call by reference. It should be named something like "<device> core.vi". The desired functions will have been determined at the time of coding the *Device module*. Besides the device-specific requests, it is likely that the *Low-level core* will have to handle "Load configuration" and "Get configuration" requests. A framework for a simple wrapper VI can be found in "Example low-level core.vi", provided with this guide.

Supplied application

This is an interesting case because it may be possible to take advantage of a large amount of functionality with a fairly small effort. The difficult choice is whether to make it into a callable VI or leave it as an independently running application. Again, the tradeoff is that the callable VI has better performance but is harder to code, whereas the independent application has worse performance but is easier to code. Strategies for both of these two approaches are suggested below.

Supplied application made callable:

In general, a fully functional application will have an initialization phase after which it will enter a while loop, repeatedly exercising various functions of the device based on the current configuration parameters. To convert this to a callable VI involves extracting (a) the initialization code and (b) the core code within the while loop into two or more separate VI's. These VI's will then be used within a framework similar to the simple wrapper VI mentioned above. Since a complete user interface likely comes with the original application, it is worth implementing the "Configure" request within the *Low-level core*, as opposed to in the *Device module* where it normally would be. This requires making the front panel of the *Low-level core* visible for the duration of the "Configure" request. To do this, first change the *Core* to be a dialog: open the *Core* VI, right-click the icon in the upper right and select VI properties... >> Window Appearance >> Dialog. Then, in the *Device module* show the *Core* before the "Configure" request and hide it again after. This can be done by connecting the path of the *Low-level core* to an Open VI Reference VI, then connecting the VI reference to a Property Node (see below). The property to set is Front Panel Window >> Open.



Supplied application made free-standing:

Taking the route of the independent free-standing application implies queue communication. Probably the best strategy is to add a separate while loop within the *Core* in which the *Core* side "Get" function (see Queue communication VI (*Core* side) above) is called at the beginning of each iteration, followed by a case statement which switches on the request received. Inside the case statement various controls/indicators would be set or read depending on the request. After processing of the request is complete, a response must be returned using

the “Send” function. The overall effect is similar to being able to operate the front panel of the application by sending requests over a queue. If care is taken to minimize the amount of CPU time used by the application, then the performance penalty can be limited. Also, if there are no other modules running on the same PC, then performance may not be a critical issue.

Separate system

The Housekeeper is a perfect example of a system separate from the data acquisition system which could be integrated into it using the *Remote module dock*. Queues are the clear choice for communication and the implementation strategy is the same as for the supplied application (above).

Since the Housekeeper is a large system which may very well grow in time, it would be worth assigning multiple *Device modules* to it. That is, there should probably be one *Device module* for each complete subsystem within the Housekeeper. Remember that once a *Device module* has been integrated into the data acquisition system it is difficult to change the configuration associated with it. Adding functionality, on the other hand, is not a problem.

3. Final integration

Once the *Device module* and *Low-level core* are functioning properly, it is time to do the final integration. These are the steps that make the new *Device module* known to the *ACQ II* system.

3.1) Install *Remote module dock* package

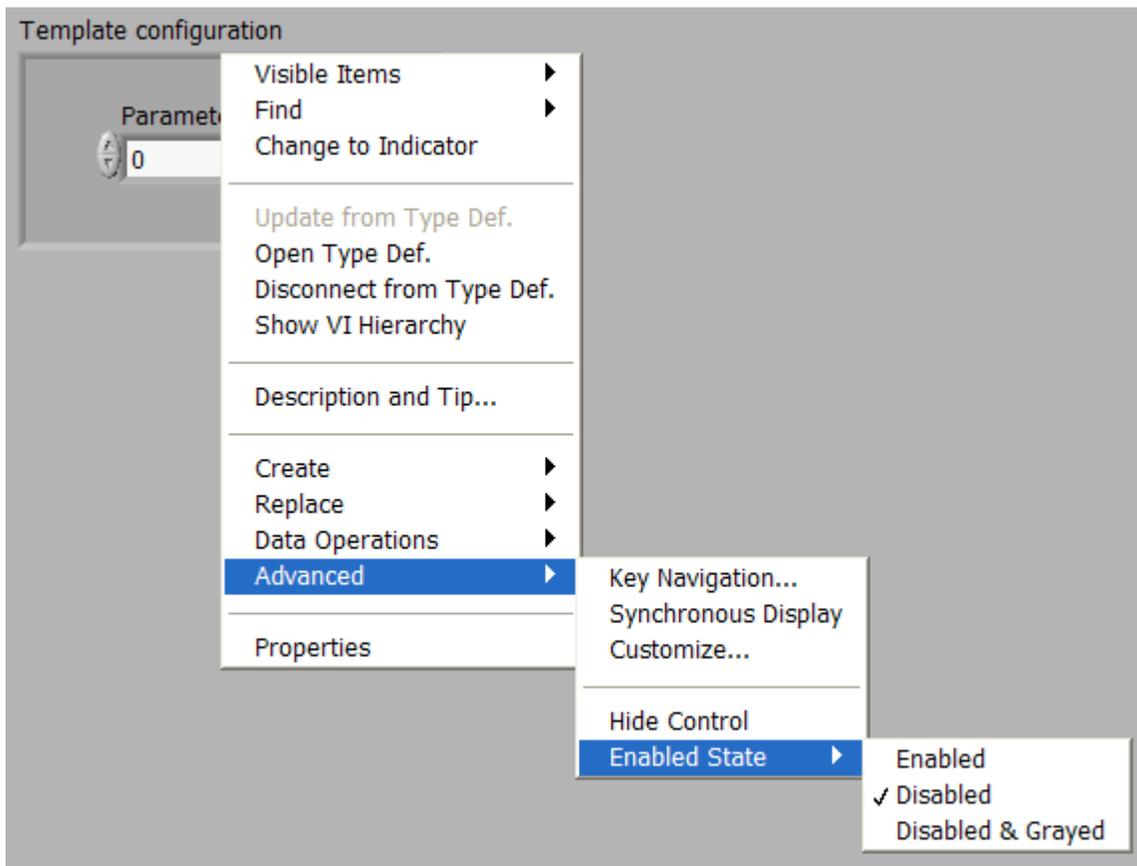
If the *Remote module dock* is not already installed, copy the 2005-10-21_remote_module_dock.zip release package into the ACQ II home\releases folder. Unzip it and follow the instructions in the release document contained within the unzipped folder. This should produce two folders: ACQ II home\Remote module dock and ACQ II home\SeamlessDB.

3.2) Create Dock – configuration (<device>).vi

Enter ACQ II home\Remote module dock and make a copy of Dock – configuration (Template).vi. Rename the copy “Dock – configuration (<device>).vi, *i.e.* insert the device name between the parentheses. Note that the device name string is an important identifier in the *Remote module dock* and must be identical in all places in which it appears.

Replace configuration cluster

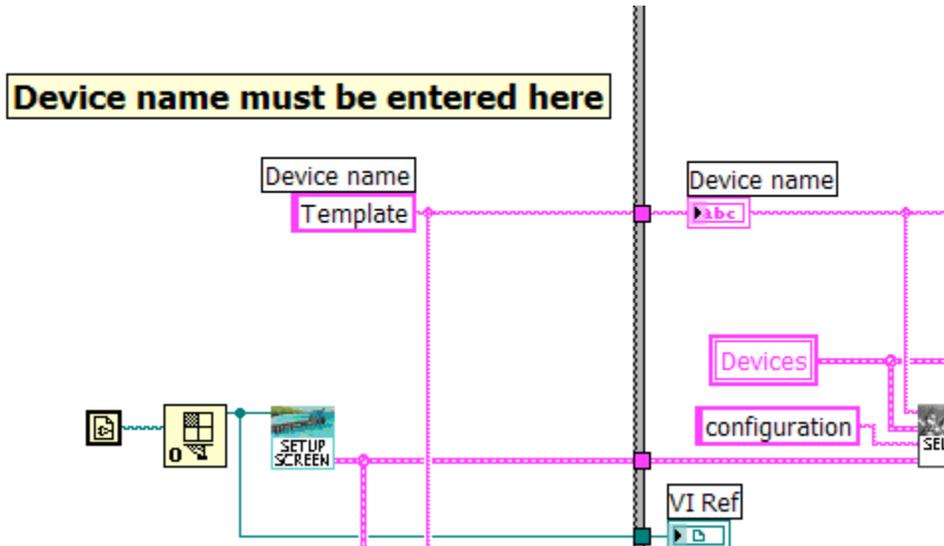
Now open Dock – configuration (<device>).vi and replace the Template configuration cluster with the



configuration cluster specific to the new device. Ensure that the newly placed cluster is a control: right-click and make sure third line is “Change to Indicator” (see previous page). Also, ensure that the cluster is disabled.

Replace device name constant

Switch to the block diagram and scroll to the far left to find the Device name constant. Enter the actual device name here.



3.3) Modify Dock – run-time.vi

Open Dock – run-time.vi. Replace the device name constant exactly as in the *Dock – configuration* VI.

3.4) Run Dock – configuration (<device>).vi

The *Dock – configuration* VI must be run once before proceeding further with the new device. As part of its initialization, the *Dock – configuration* VI checks to see if the device database installation has been done and, if not, does so automatically. This requires some input from the end user:

- Device type (selected from a list)
- Device module IP address
- Device module VI path (selected using a browser)
- Device description

Once this last step is completed, the device is fully integrated and ready to use.

4. Checklist

1. Device module

- 1.1) Configure the connector pane
- 1.2) Design the configuration cluster
- 1.3) Identify the device-specific run-time state parameters
- 1.4) Setup block diagram
- 1.5) Implement “*Get configuration*” request
- 1.6) Implement “*Load configuration*” request
- 1.7) Implement “*Configure*” request
- 1.8) Implement “*Get run-time state*” request
- 1.9) Implement “*Get device-specific requests*” request
- 1.10) Implement “*Get integer names*” request
- 1.11) Implement “*Get integer*” request
- 1.12) Implement “*<Device-specific requests>*”
- 1.13) (Optional) Edit VI icon

2. Low-level core

3. Final integration

- 3.1) Install *Remote module dock* package
- 3.2) Create Dock – configuration (<device>).vi
- 3.3) Modify Dock – run-time.vi
- 3.4) Run Dock – configuration (<device>).vi