Arcus Technology DMX Series Steppers and LabVIEW

An Introduction to DMX Series Stepper Motor Automation With LabVIEW
Application Note 001

Kod Integrations, LLC
http://www.kodintegrations.com/solutions/labview/stepper

1. Introduction

Arcus Technology, in conjunction with Kod Integrations, LLC recently added to its suite of stepper motor software tools a fully functional LabVIEW driver. This application note is designed to facilitate even the novice LabVIEW developer in swiftly creating their own functional LabVIEW applications utilizing this driver and a set of (3) DMX-J-SA-17 stepper motors.

Please note that this document is not intended to act as a lesson in proper LabVIEW development techniques, nor does it focus on one design pattern over another. Its intent is to demonstrate the usage of the Arcus supplied LabVIEW driver for controlling DMX-J-SA-17 stepper motor(s) via simple, clean and functional examples.

2. The Basics

Before one can begin developing their own custom application, an understanding of the fundamental LabVIEW driver components is necessary. Assuming the driver has been downloaded and properly installed to the LabVIEW development environment, the top-level of the Arcus custom palette of VI's looks as seen in *Figure 1.0, Top Level Custom Palette.*

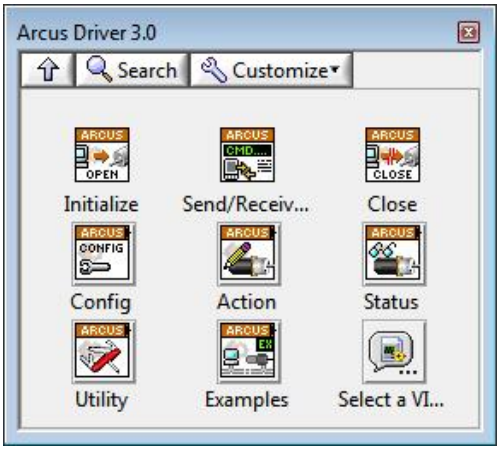

Figure 1, Top Level Custom Palette

The three primary LabVIEW VI's to be discussed here are "Initialize", "Send/Receive Command" and "Close".

2.1 Creating a Connection - Initialize

Before any standard communications can be established with an USB-connected DMX device, a connection reference must first be established. This is where the initialize.vi comes in:

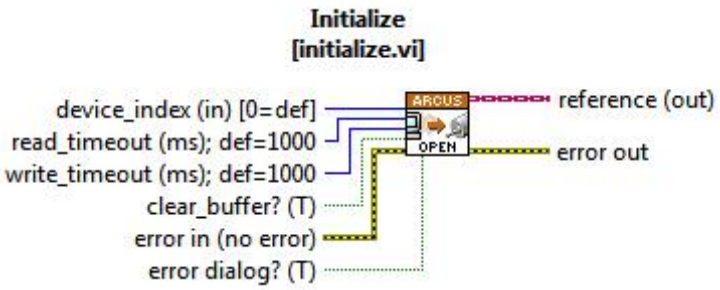

Figure 2, Create Connection

device_index (in) — An attached device is initially identified by its index. For example, if a single motor is physically connected to the controlling computer, its index is by default '0'. If more than one device is connected, the index of each additional device is incremented by 1; device 1 is at index 0, device 2 is at index 1, device *n* is at index *n-1* and so forth.

read_timeout (ms) — The option to set the time out (in milliseconds) on USB reads exists at initialization. The default is 1000 ms (1 sec).

write_timeout (ms) — The option to set the time out (in milliseconds) on USB writes exists at initialization. The default is 1000 ms (1 sec).

clear_buffer? (T) — In the event a motor session was improperly terminated (i.e. PC power loss), the 'clear_buffer' option is designed to reset that connection, allowing a newly refreshed connection to be established. This, by default, is set to TRUE.

error_dialog? (T) — In the event an exception occurs during runtime, a dialog box will appear at the time of the exception IF this default value is left as TRUE. Set to FALSE if dialog messages are to be suppressed.

NOTE: errors are always propagated through, regardless of the error dialog state.

reference — Once a motor has been properly initialized (a connection has been properly established), a reference is generated that not only identifies the connected motor but also contains several pieces of information that help define the connected motor.

2.2 Send/Receive Command

Perhaps the heart of the driver is the Send/Receive Command VI designed to communicate to the device the commands defined in the Arcus Technology **ASCII Language Specification** for the DMX Series devices:

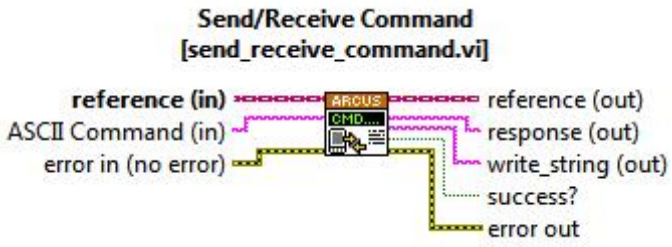

Figure 3, Send/Receive Command

reference (in)	A required input, this is the motor reference generated via the Create Connection described above.
ASCII Command (in)	The raw ASCII command as defined in the Arcus Technology ASCII Language Specification for the DMX Series devices.
reference (out)	This is the duplicated input reference passed back out
response (out)	This is the motor response after sending the command. The response could be a simple OK, acknowledging successful receipt of a command or it could provide a response to a motor query such as the programmed acceleration value.
success?	Returns TRUE if the command/query was successful. Returns FALSE if an error was encountered. **NOTE:** an exception will also be transmitted out the "error out" cluster in the event "success?" happens to be FALSE.
write_string (out)	The ASCII Command (in) passed through

2.3 Close the Connection

Once communication with the attached motor is complete, the reference must be properly closed.

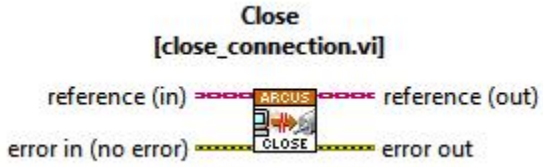

Figure 4, Close Connection

reference (in) A required input, this is the motor reference generated via the Create Connection (and required for further communications) as described above.

3. Single DMX Series Communication

With a firm understanding of the primary driver components required to establish communication with the DMX series stepper, the following steps to building a simple LabVIEW application will be much more intuitive.

Have a look at the following front panel and block diagram for the VI "*01_issue_raw_command_string.vi*":

Front Panel:

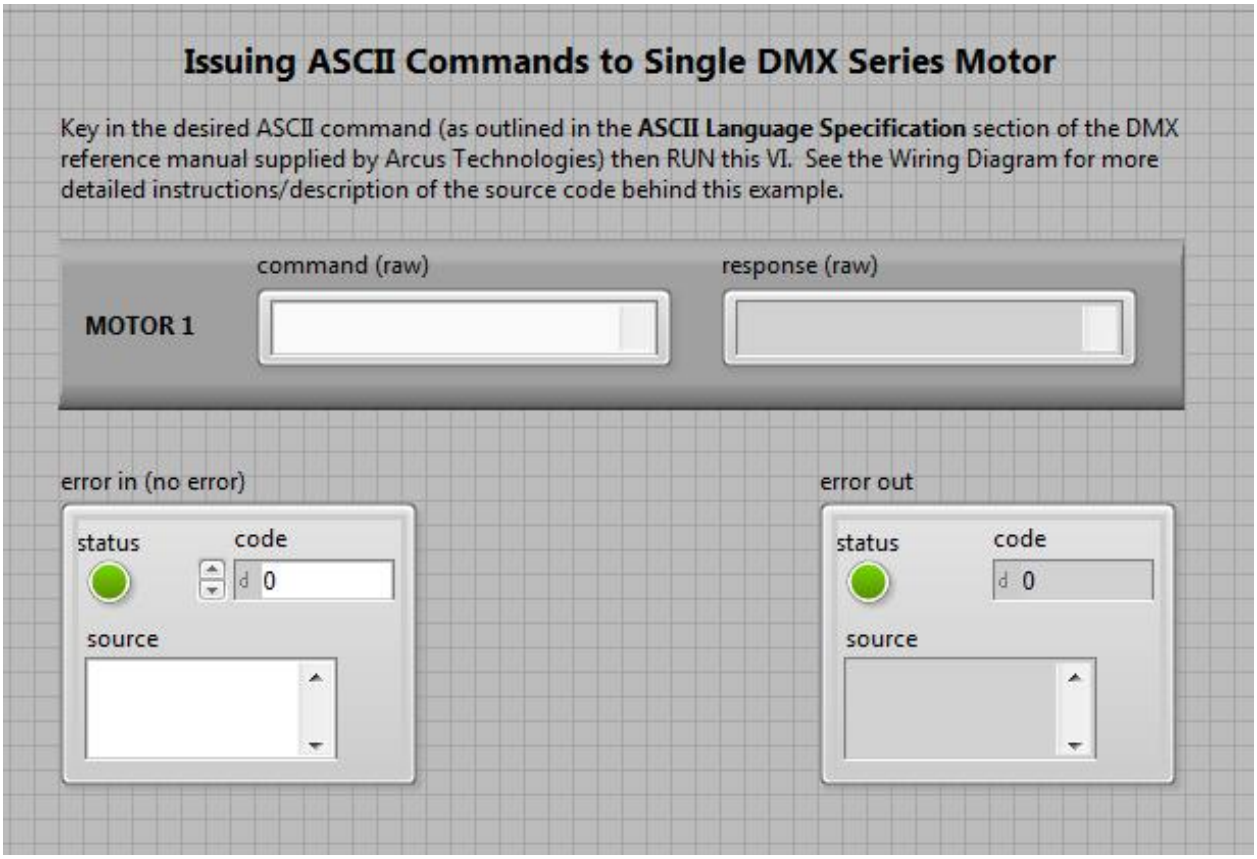

Figure 5, 01_issue_raw_command_string.vi front panel

Block Diagram:

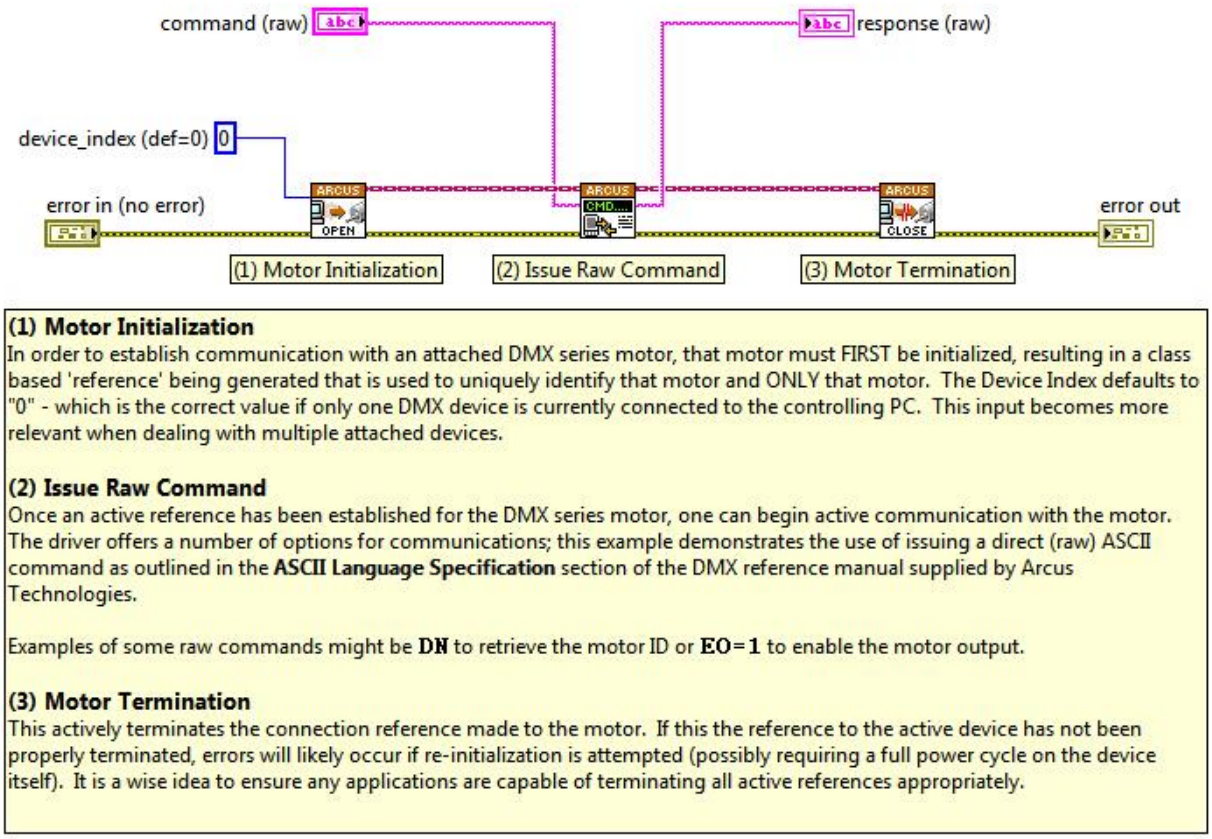

Figure 6, 01_issue_raw_command_string.vi block diagram

As demonstrated here, all three main components individually described in Section 2 above are wired to form a single, simple application for communicating with a single DMX series stepper motor.

Note that because this is a single-motor application, the “device_index” input will always be ‘0’. The value of this input becomes more important when dealing with multiple connected devices (described in subsequent sections).

To see how this VI works, simply go to the “dmx_applications.lvproj” project and launch the VI “01_issue_raw_command_string.vi” beneath the “single_unit” folder:

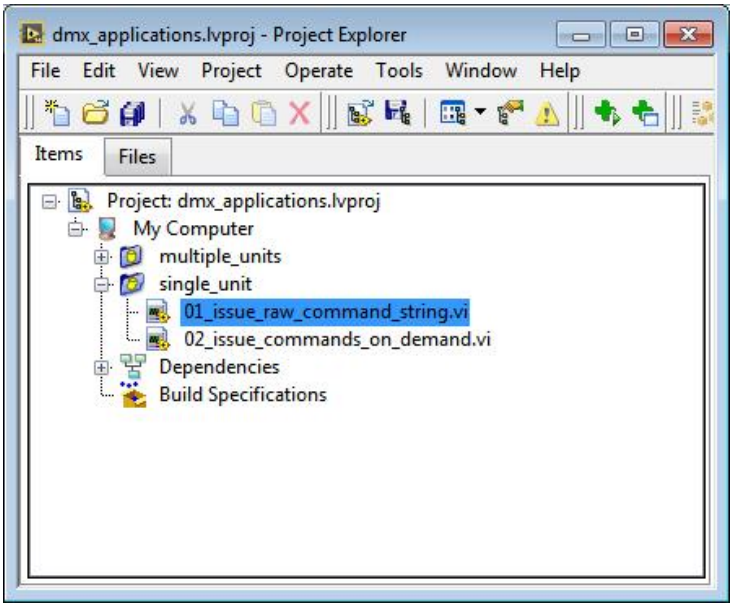

Figure 7, Project View

Enter a command in the "command (raw)" field and RUN the VI. For example, enter the command "DN" (without the quotes) and RUN the VI. The motor ID is returned.

Although this VI demonstrates the full functionality of initializing the motor, issuing a command and closing that reference, doing this each time to send multiple commands is inefficient and unrealistic. Have a look at the following front panel and diagram:

Front Panel:

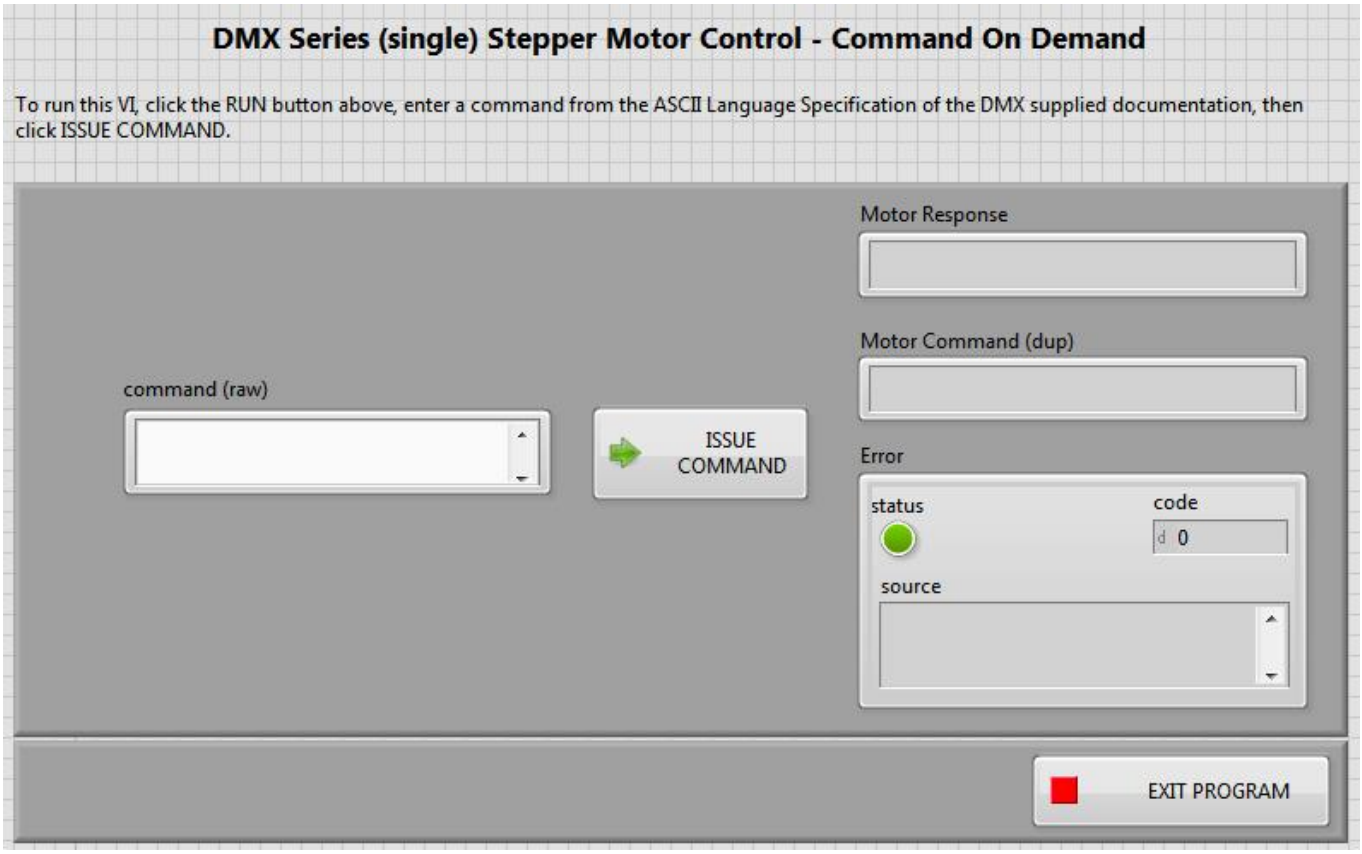

Figure 8, 02_issue_commands_on_demand.vi front panel

Block Diagram:

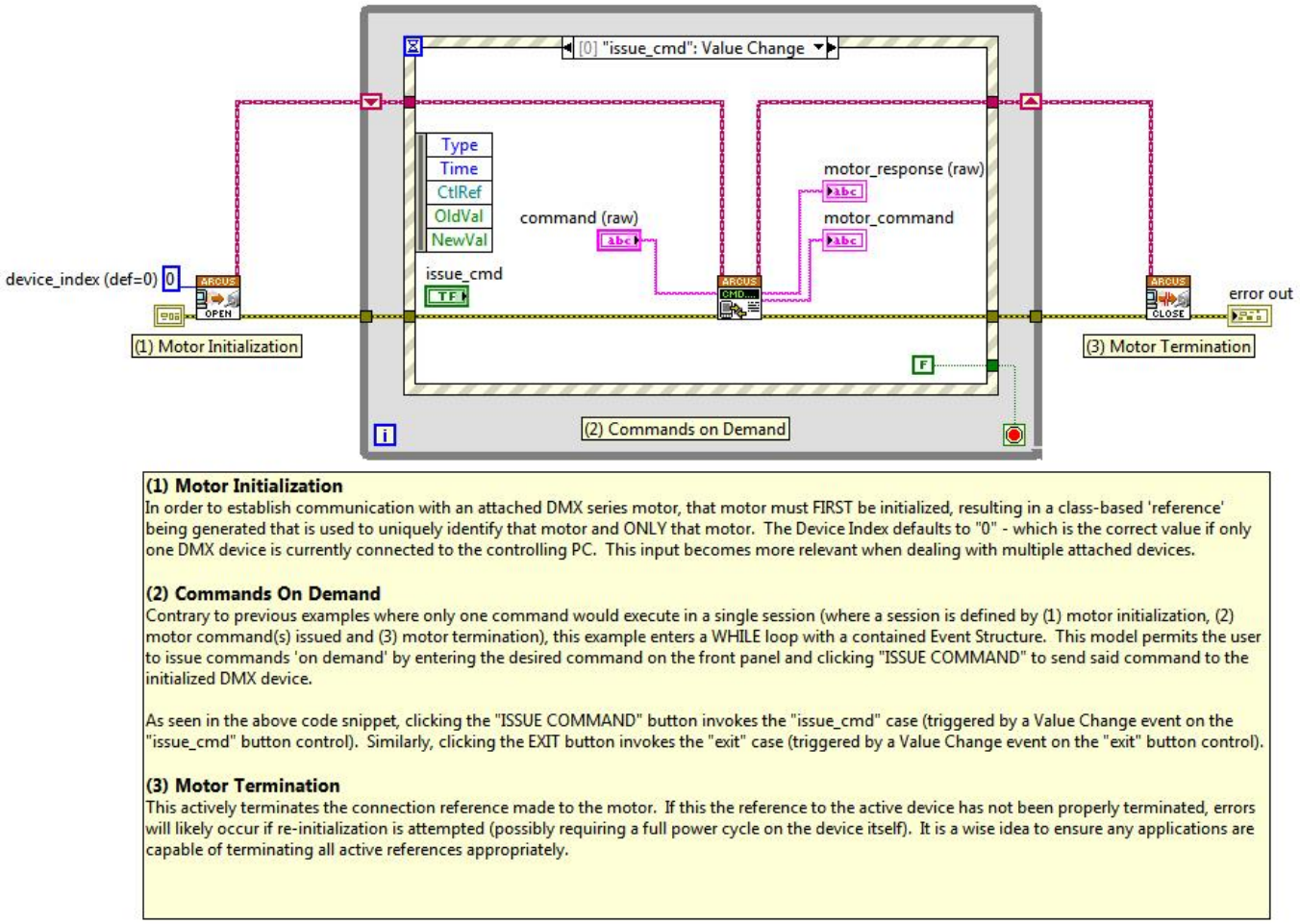

Figure 9, 02_issue_commands_on_demand.vi block diagram

In this particular example, when the VI is executed, the attached motor at index 0 (remember, by default) is initialized and the VI sits and waits for the User to (1) enter a command to be issued and (2) click the “ISSUE COMMAND” button to send that command. This allows the User to send multiple commands in a single session without the need to initialize the motor (create a reference) and terminate that motor session (closing the reference) each time. Ultimately, when the User is finished with communications, he/she clicks the “EXIT PROGRAM” button to exit the “commands on demand” while loop above and closing the motor reference.

To see how this VI works:

1. go to the "dmx_applications.lvproj" project and launch the VI "02_issue_commands_on_demand.vi" beneath the "single_unit" folder.
2. RUN the VI
3. Enter a command in the "command (raw)"
4. Click the "ISSUE COMMAND" button

Note the VI continues to run, waiting for the User to issue another command (or until the User clicks the "EXIT PROGRAM" button). So to best demonstrate the advantage to this design approach, one might want to issue a number of commands in a single session: get the device ID, enable the motor output, jog the motor in the positive direction, stop motor movement and disable motor output:

1. **Get ID**: Enter `DN` in "command (raw)" field, click ISSUE COMMAND
    a. Note the output "Motor Response" might read JSA00
2. **Enable Output**: Enter `EO=1` in "command (raw)" field, click ISSUE COMMAND
    a. Note the output reads "OK"
3. **Jog**: Enter `J+` in the "command (raw)" field, click ISSUE COMMAND
    a. Note the motor begins rotating
    b. Note the output reads "OK"
4. **Stop**: Enter `STOP` in the "command (raw)" field, click ISSUE COMMAND
    a. Note the motor stops rotating
    b. Note the output reads "OK"
5. **Disable Output**: Enter `EO=0` in the "command (raw)" field, click ISSUE COMMAND
    a. Note the output reads "OK"

4. Multiple DMX Series Communication

Many applications may require more than one DMX series motor to be controlled at one time. This section, as with the previous section, covers two example applications demonstrating the use of the Arcus LabVIEW driver but to control (3) DMX series stepper motors as opposed to (1).

Having a look at the front panel and block diagram of another VI (in the same project) named "*01_issue_raw_command_string (multi).vi*":

Front Panel

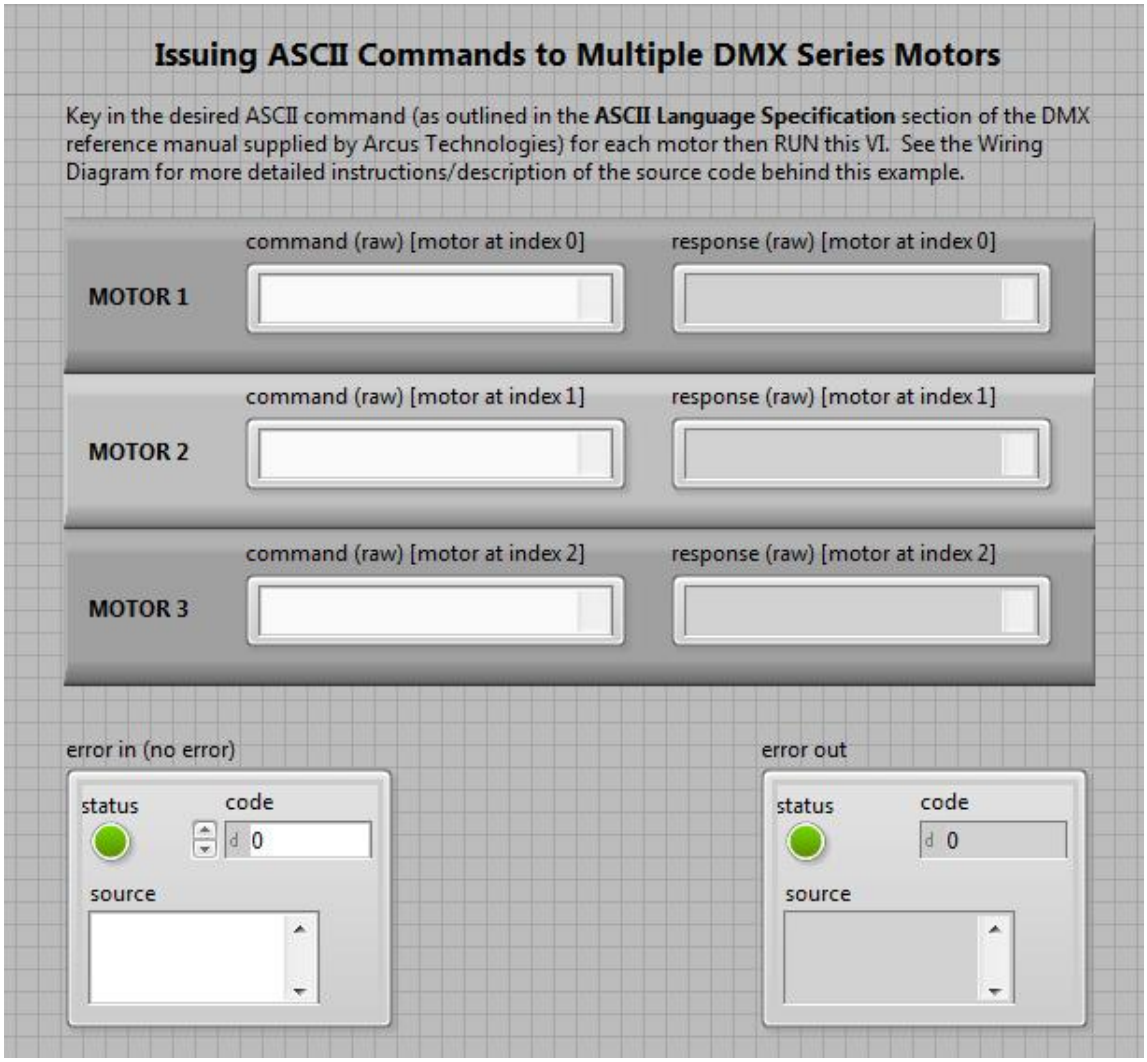

Figure 10, 01_issue_raw_command_string (multi).vi front panel

Block Diagram

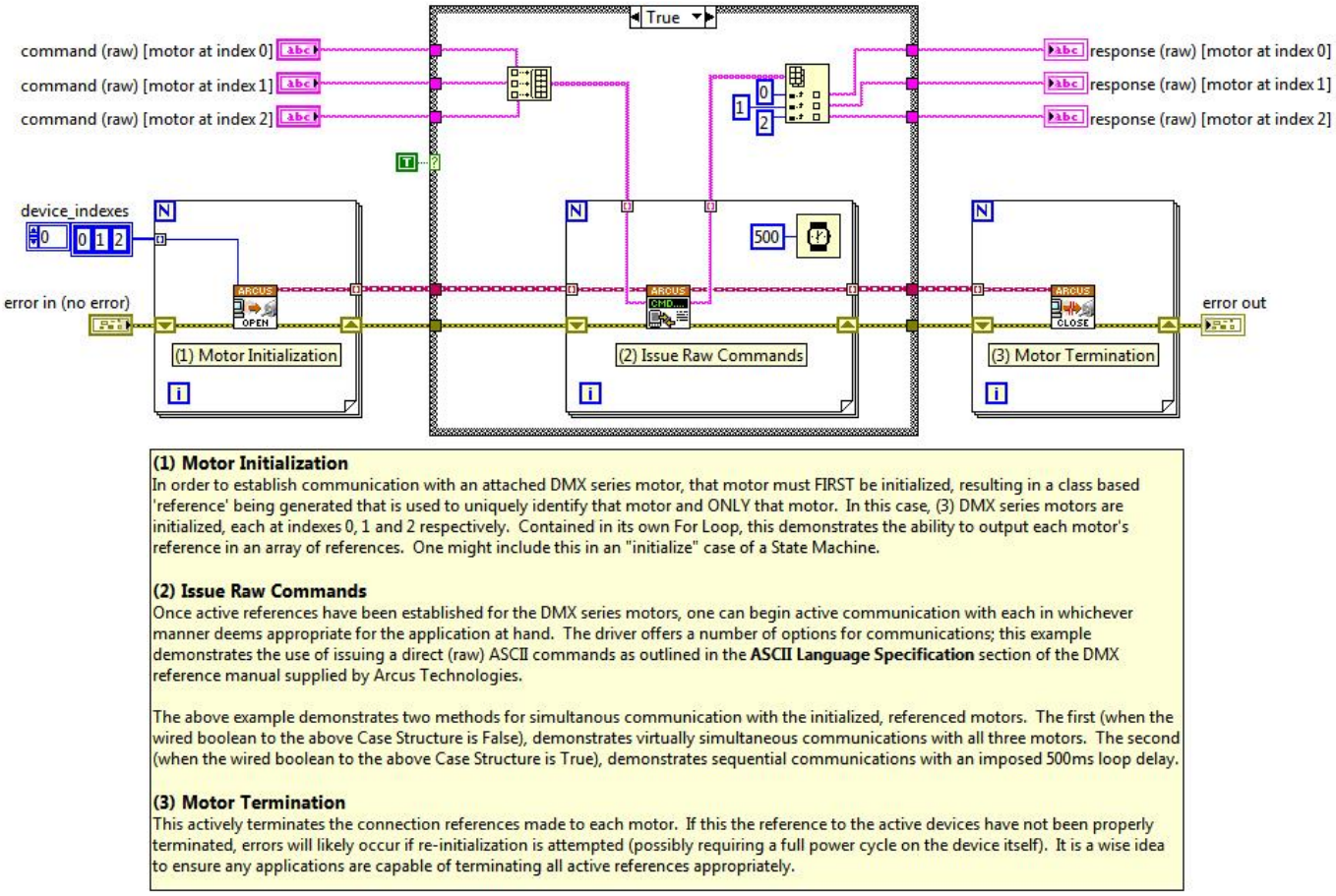

Figure 11, 01_issue_raw_command_string (multi).vi block diagram (sequential commands)

Analogous in behavior to the first example discussed in Section 3, this VI demonstrates usage of the three primary driver components applied to (3) stepper motors.

The attached device indexes are 0-indexed – meaning the first connected device is index 0, the second is index 1, the third is index 2 and so forth. Here the initialize step is encased in a For Loop, indexed by an array of device indexes 0 through 2. Therefore the first step creates (3) separate references, 1 for each initialized motor.

The second step permits the User to send (3) independent commands to each of the referenced motors, with a 500ms delay imposed. All three responses are retrieved and sent to (3) independent indicators on the front panel.

Lastly, all three references are closed.

To see how this VI works:

1. go to the "dmx_applications.lvproj" project and launch the VI "01_issue_raw_command_string (multi).vi" beneath the "multiple_units" folder.
2. Enter 3 separate ASCII commands in the available "command (raw)" fields for each of the motors
3. RUN the VI
4. Note the outputs for each motor

Note in this particular example, the second step (issue raw commands) is wrapped in a case structure, displaying the TRUE case. As mentioned above, the three motors receive their commands sequentially. Have a look the FALSE case, illustrated below:

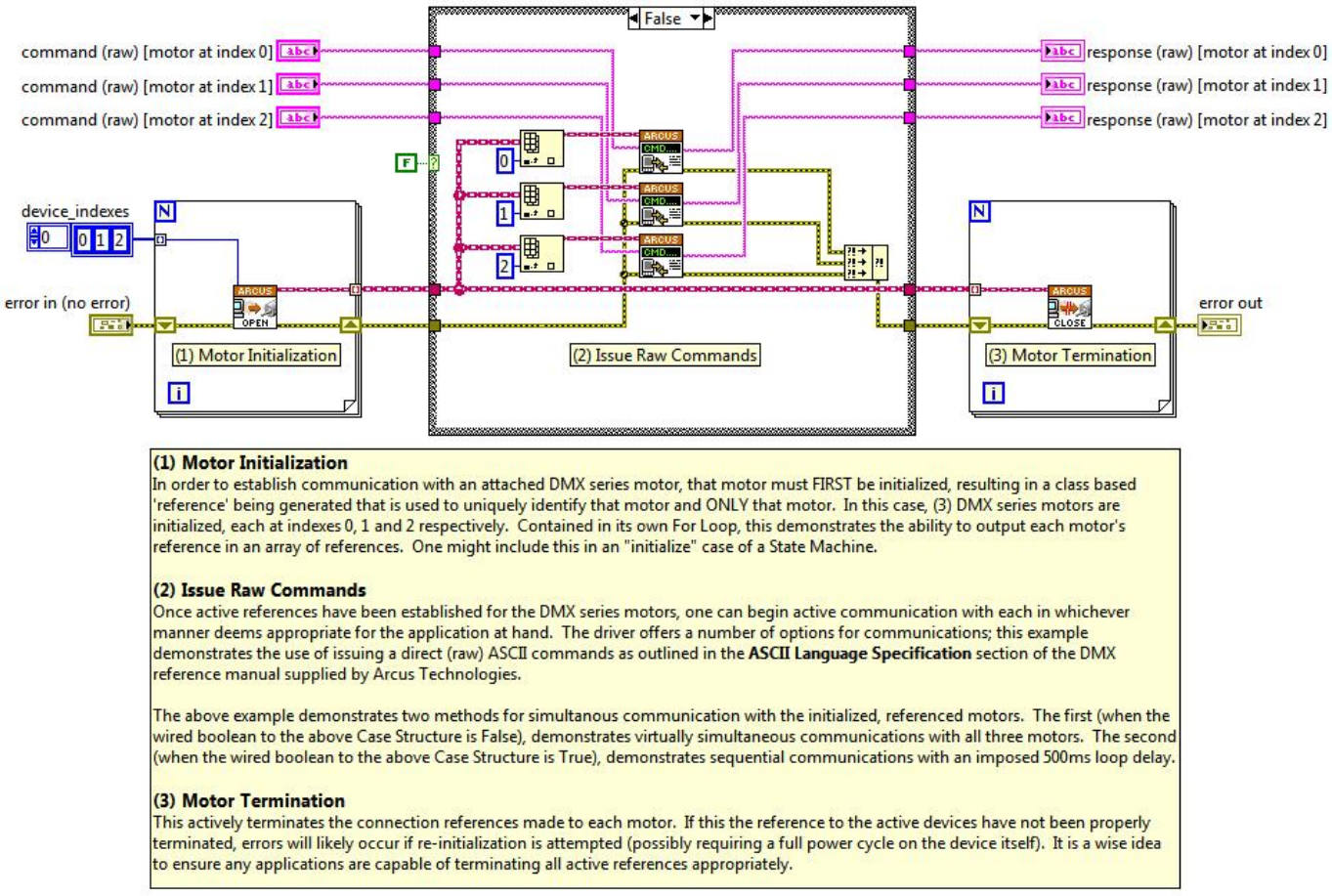

Figure 12, 01_issue_raw_command_string (multi).vi block diagram (simultaneous commands)

This case demonstrates an approach to simultaneously send independent commands to all three motors.

Now for a more realistic design approach. Take a look at the VI "*02_issue_commands_on_demand (multi).vi*":

Front Panel:

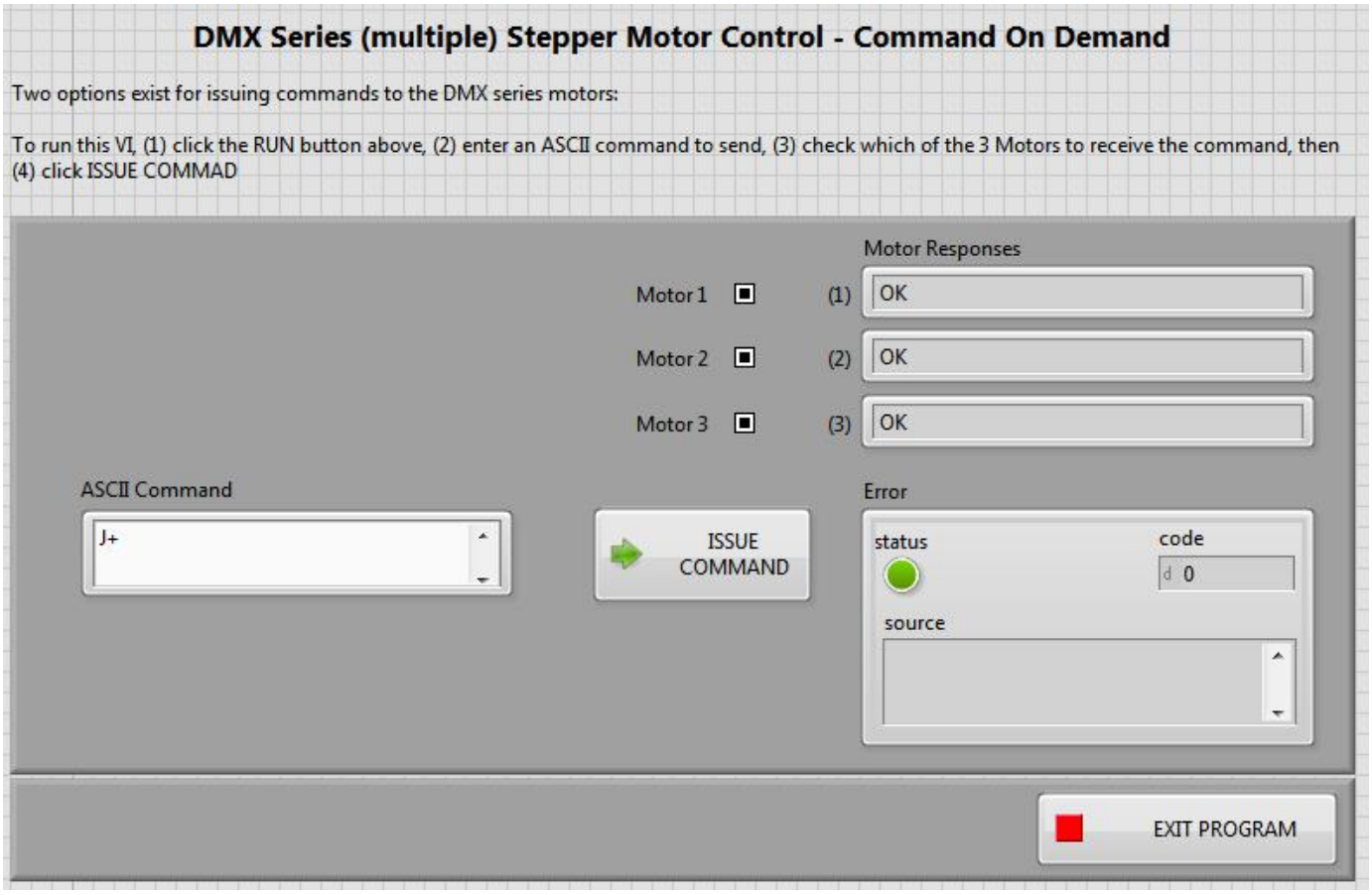

Figure 13, 02_issue_commands_on_demand (multi).vi front panel

Block Diagram:

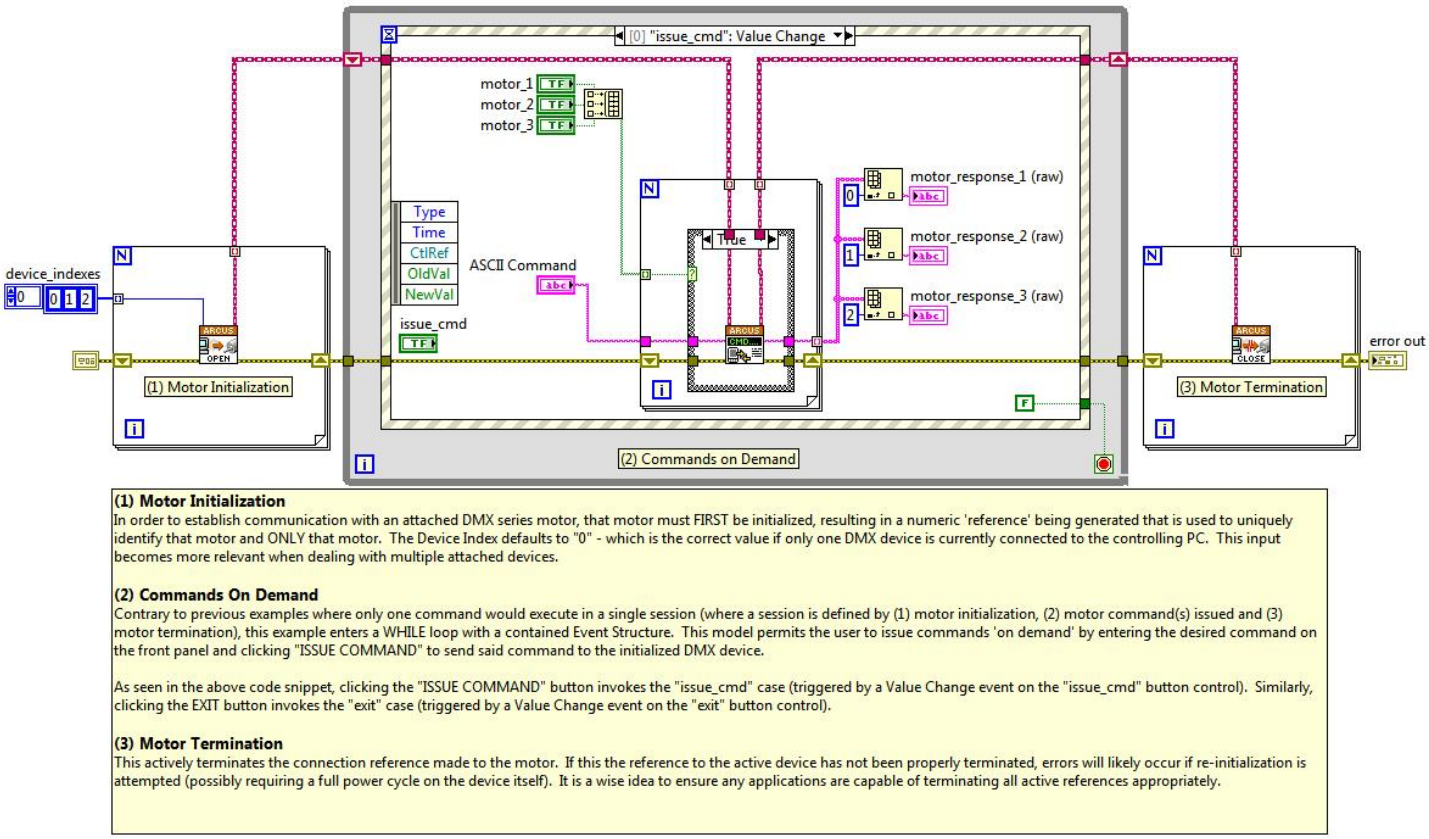

Figure 14, 02_issue_commands_on_demand (multi).vi block diagram

Analogous in behavior to the second example in section 3 above, the User first RUNs this VI and proceeds to send commands on demand. However, in this case, just a single command is delivered to motors of the Users choosing. For example (as illustrated in the front panel screenshot above), the use can issue a Jog command (J+) to all three motors by checking the checkboxes for each motor and clicking the ISSUE COMMAND button.

To see how this VI works:

1. go to the “dmx_applications.lvproj” project and launch the VI “02_issue_commands_on_demand (multi).vi” beneath the “multiple_units” folder.
2. RUN the VI
3. Enter an ASCII commands in the available “command (raw)” field
4. Choose the motors to that are to receive the command
5. Click ISSUE COMMAND
6. Note the outputs for each motor

4.1 Motor Identification

When dealing with multiple motors, it is not intuitive as to which of these motors will be assigned to which index – making it difficult when programmatically identifying and targeting specific motors. That said, the generated "reference" upon initialization is an instantiated base "_Controller" class that contains parameters that identify the initialized motor

Within the Controller class object are data members such as "Device ID", "Device Name", "Device Model (or type)" and "Device Index". This information is inherently transmitted via the reference throughout the control application and is accessible at any time prior to closing said reference. The example (03_initialization_and_identification.vi) demonstrates the initialization process and how one might want to access these parameters via a series of "member accessor" VIs (all of which are available in the Arcus USB Comm driver).

NOTE: All motors received from Arcus technologies will likely be shipped with the same name (JSA00 or JSA01). It is highly suggested to connect each motor independently and change their names to be JSA00, JSA01, JSA02, etc.

Front Panel:

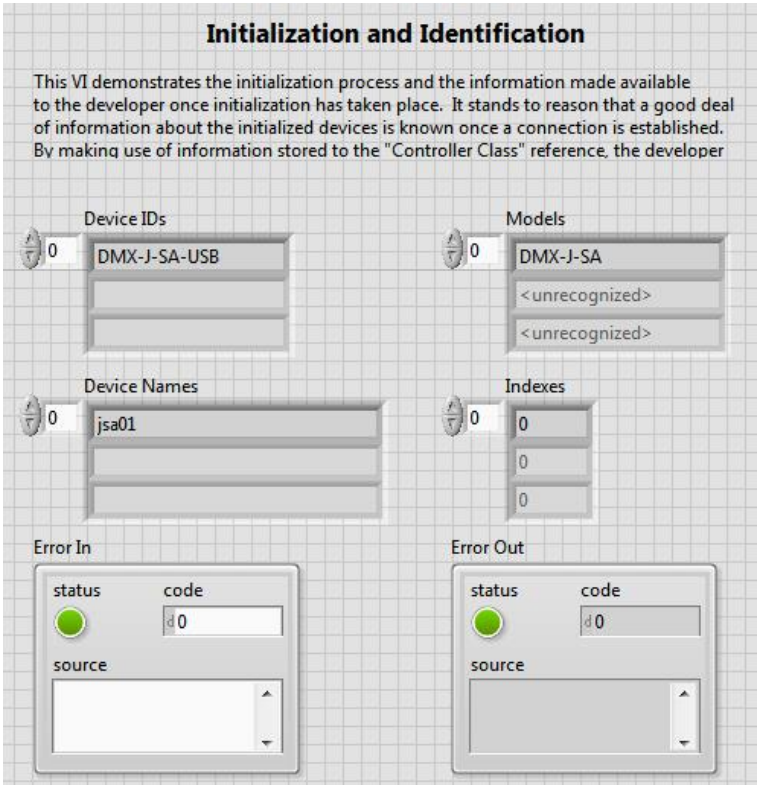

Figure 15, 03_initialization_and_identification.vi front panel

Block Diagram:

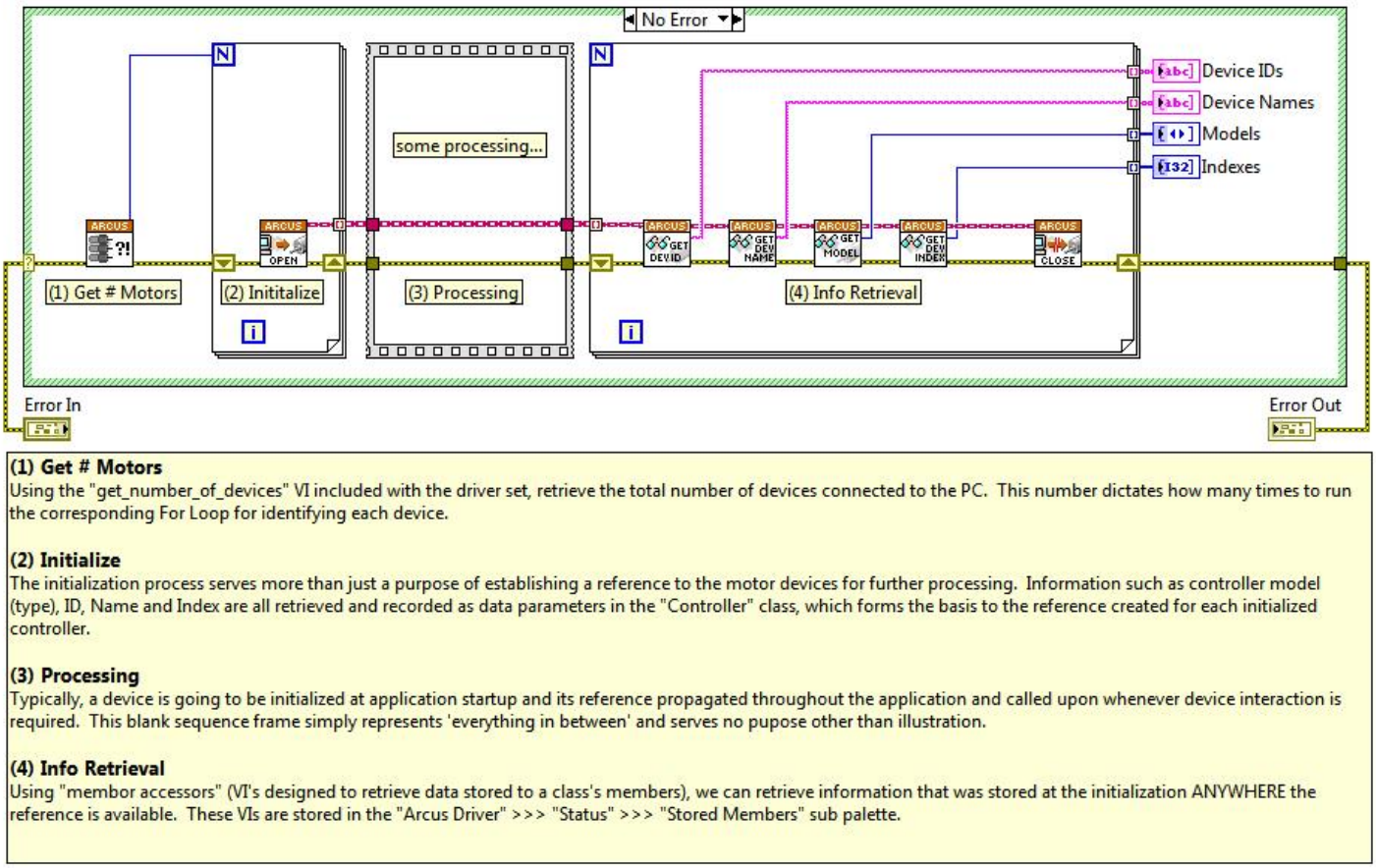

Figure 16, 03_initialization_and_identification.vi block diagram