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.*

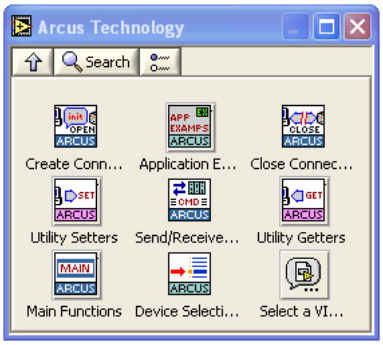


Figure 1, Top Level Custom Palette

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

2.1 Creating a Connection

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

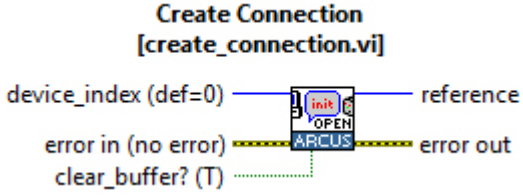


Figure 2, Create Connection

device_index (def=0) — An attached device is 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.

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.

reference — Once a motor has been properly initialized (a connection has been properly established), a reference is generated identifying this specific motor's connection.

2.2 Send/Receive Command

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

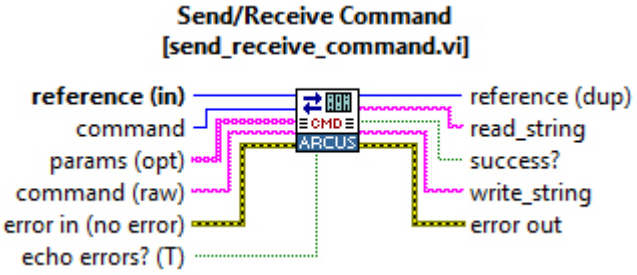


Figure 3, Send/Receive Command

reference (in) — A required input, this is the motor reference generated via the Create Connection described above.

command (raw) — The raw ASCII command as defined in the Arcus Technology **ASCII Language Specification** for the DMX Series devices. If this input is non-null (not empty), it becomes the overriding 'command input' to this VI. Meaning, any values wired to inputs *command* and *params (opt)* are disregarded. See below for a more detailed description of these two inputs.

command — This is an alternative approach to sending the desired command to the referenced motor. It is nothing more than a pre-defined, strict-type def enumerated constant providing the developer with a more human-readable set of commands. Behind the scenes, each value in the enumerated constant simply gets mapped to the corresponding ASCII command.

params (opt) — Some ASCII commands take as input required parameter(s) for commanding the motor(s). Those parameters are broken out into an optional array of type String and is *only* used in conjunction with the *command* input and not the *command (raw)* input as

the *command (raw)* input will already have these parameters factored into the command string.

NOTE: It is recommended the developer first become acquainted with the ASCII command set in the DMX motor documentation and then, once comfortable, begin using the *command/params(opt)* inputs for ease of development.

echo errors? (T) By default, the "echo errors?" input is TRUE. That said, if an error is encountered while attempting to send the desired command, a dialog box will pop up stating what the error is. Regardless of whether this is input is TRUE or not, the error is always transmitted out the "error out" cluster.

reference (dup) This is the duplicated input reference passed back out

read string 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.

write string this is nothing more than the ASCII equivalent to the input command. Regardless of which command approach is taken (using *command (raw)* or *command w/params*), the ASCII equivalent can be retrieved here.

2.3 Close the Connection

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

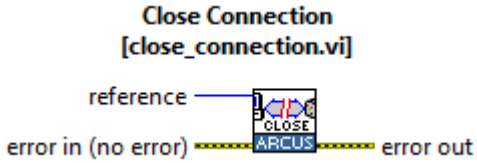


Figure 4, Close Connection

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:

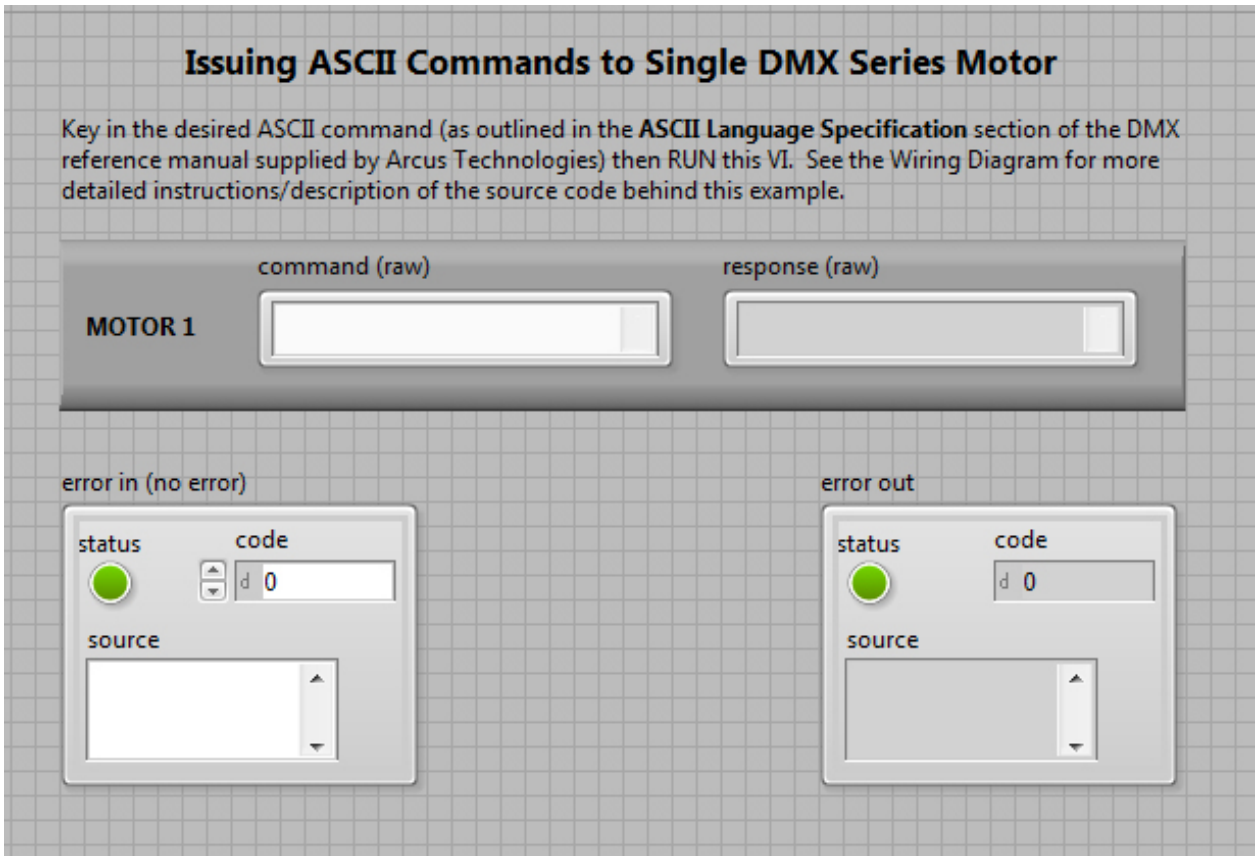


Figure 5, 01_issue_raw_command_string.vi front panel

Block Diagram:

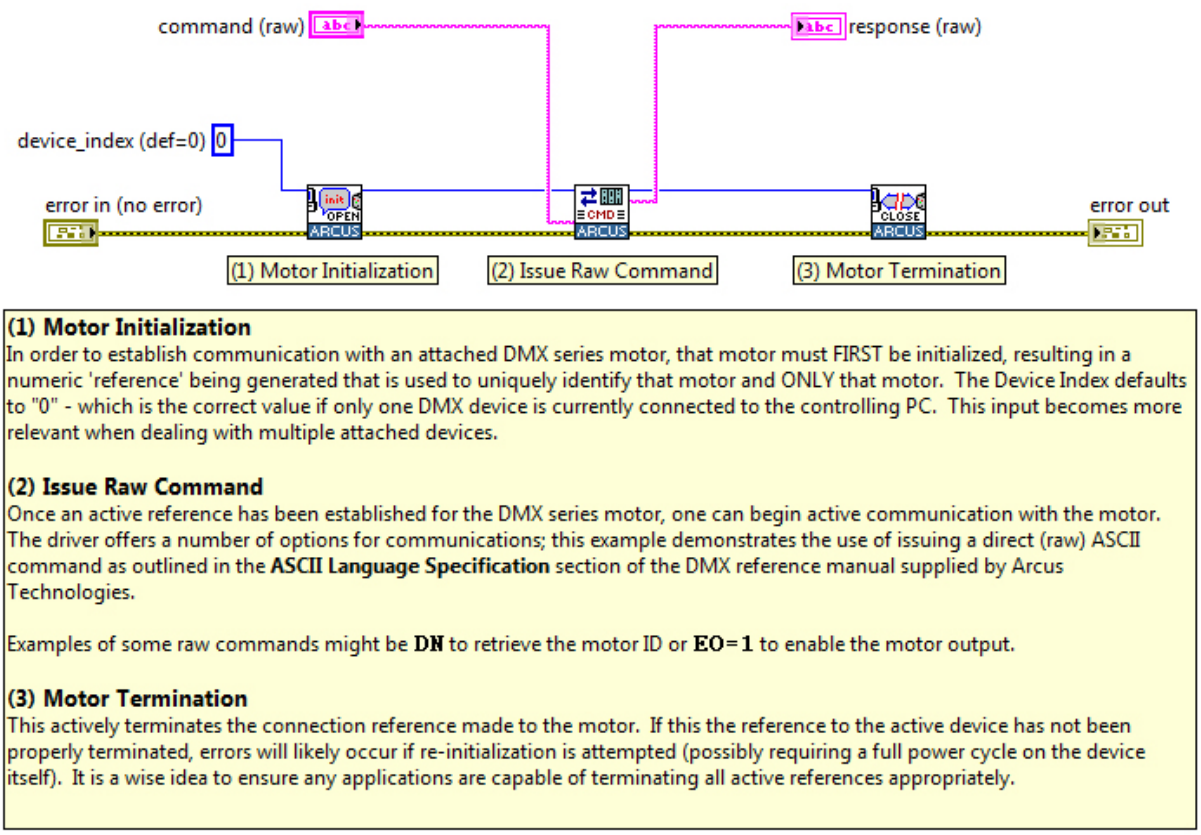


Figure 6, 01_issue_raw_command_string.vi block diagram

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:

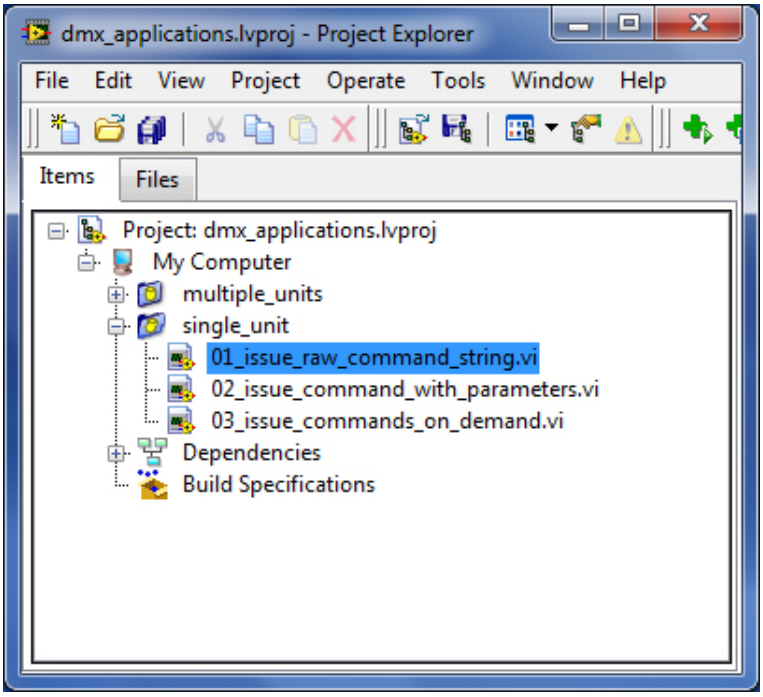


Figure 7, Project View

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:

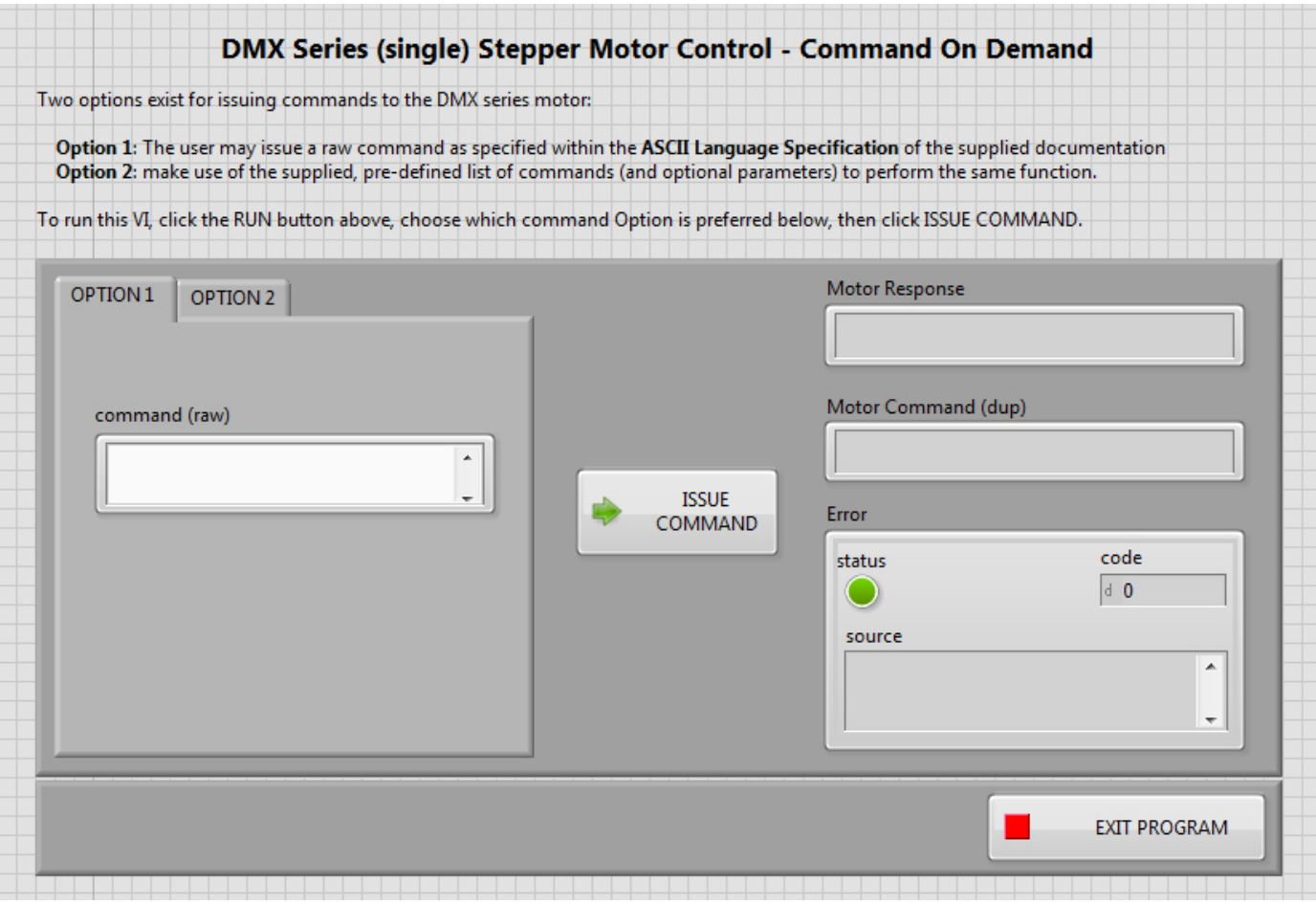
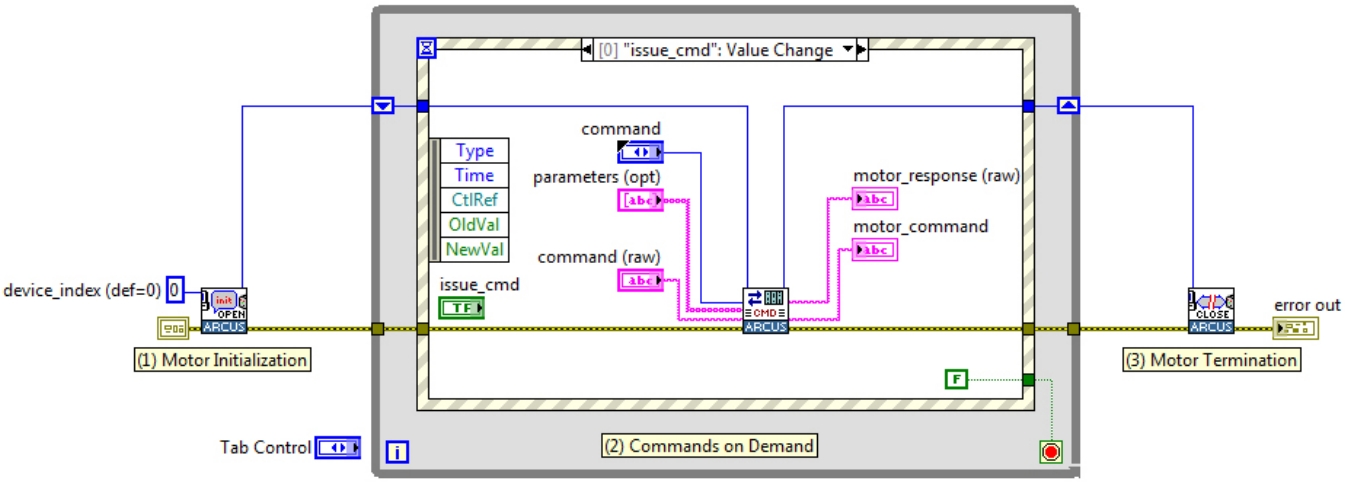


Figure 8, 03_issue_commands_on_demand.vi front panel

Block Diagram:



(1) Motor Initialization
In order to establish communication with an attached DMX series motor, that motor must FIRST be initialized, resulting in a numeric 'reference' being generated that is used to uniquely identify that motor and ONLY that motor. The Device Index defaults to "0" - which is the correct value if only one DMX device is currently connected to the controlling PC. This input becomes more relevant when dealing with multiple attached devices.

(2) Commands On Demand
Contrary to previous examples where only one command would execute in a single session (where a session is defined by (1) motor initialization, (2) motor command(s) issued and (3) motor termination), this example enters a WHILE loop with a contained Event Structure. This model permits the user to issue commands 'on demand' by entering the desired command on the front panel and clicking "ISSUE COMMAND" to send said command to the initialized DMX device.

As seen in the above code snippet, clicking the "ISSUE COMMAND" button invokes the "issue_cmd" case (triggered by a Value Change event on the "issue_cmd" button control). Similarly, clicking the EXIT button invokes the "exit" case (triggered by a Value Change event on the "exit" button control).

Please refer to the instructions on the Front Panel that detail the usage of all three input controls wired to the send_receive_cmd.vi.

(3) Motor Termination
This actively terminates the connection reference made to the motor. If this the reference to the active device has not been properly terminated, errors will likely occur if re-initialization is attempted (possibly requiring a full power cycle on the device itself). It is a wise idea to ensure any applications are capable of terminating all active references appropriately.

Figure 9, 03_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 a command is issued. 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 "03_issue_commands_on_demand.vi" beneath the "single_unit" folder.
2. RUN the VI
3. Enter a command in the "command (raw)" (beneath the Option 1 tab)
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"

Note this VI presents an alternative approach to issuing these same commands. On the front panel, click the "OPTION 2" tab to reveal two fields "Pre-defined Command" and "Parameters". Rather than recalling, for example, what the ASCII command may be for retrieving the motor ID one has the option to choose the command from a human-readable drop down list. Revisiting the same sequence of commands:

1. **Get ID**: select "get device id" from the "Pre-defined Command" enumerated control list, click ISSUE COMMAND
2. **Enable Output**: select "power up" from the "Pre-defined Command" enumerated control list, click ISSUE COMMAND
3. **Jog**: select "constant jog(+)" from the "Pre-defined Command" enumerated control list, click ISSUE COMMAND
4. **Stop**: select "stop slow" from the "Pre-defined Command" enumerated control list, click ISSUE COMMAND
5. **Disable Output**: select "power down" from the "Pre-defined Command" enumerated control list, click ISSUE COMMAND

The “Parameters (where applicable)” field isn’t required for these commands. An example of where one might make use of the parameters array control is if a command requires some additional parameters for proper execution. Setting the device ID is an example.

The ASCII command for setting device ID is DN=<dev_id> where <dev_id> is any ID with names JSA00 through JSA99 (pulled directly from the **ASCII Language Specification**). The parameter (just one) in this case would be the desired device ID (i.e. JSA99). So to set the ID using the OPTION 2 command method, the “Pre-defined Command” would be “set device id” and the one parameter would be JSA99.

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

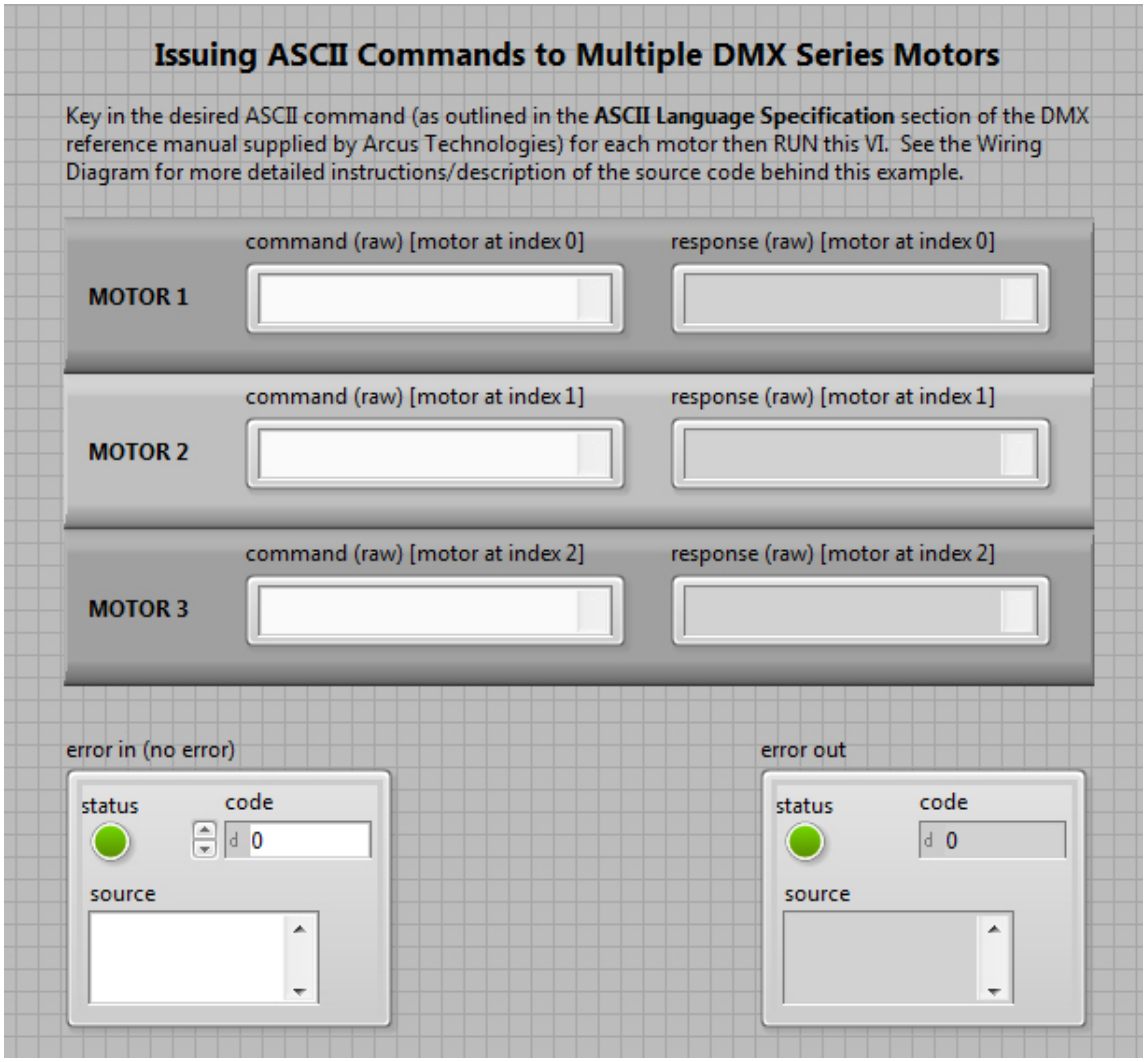


Figure 10, 01_issue_raw_command_string (multi).vi front panel

Block Diagram

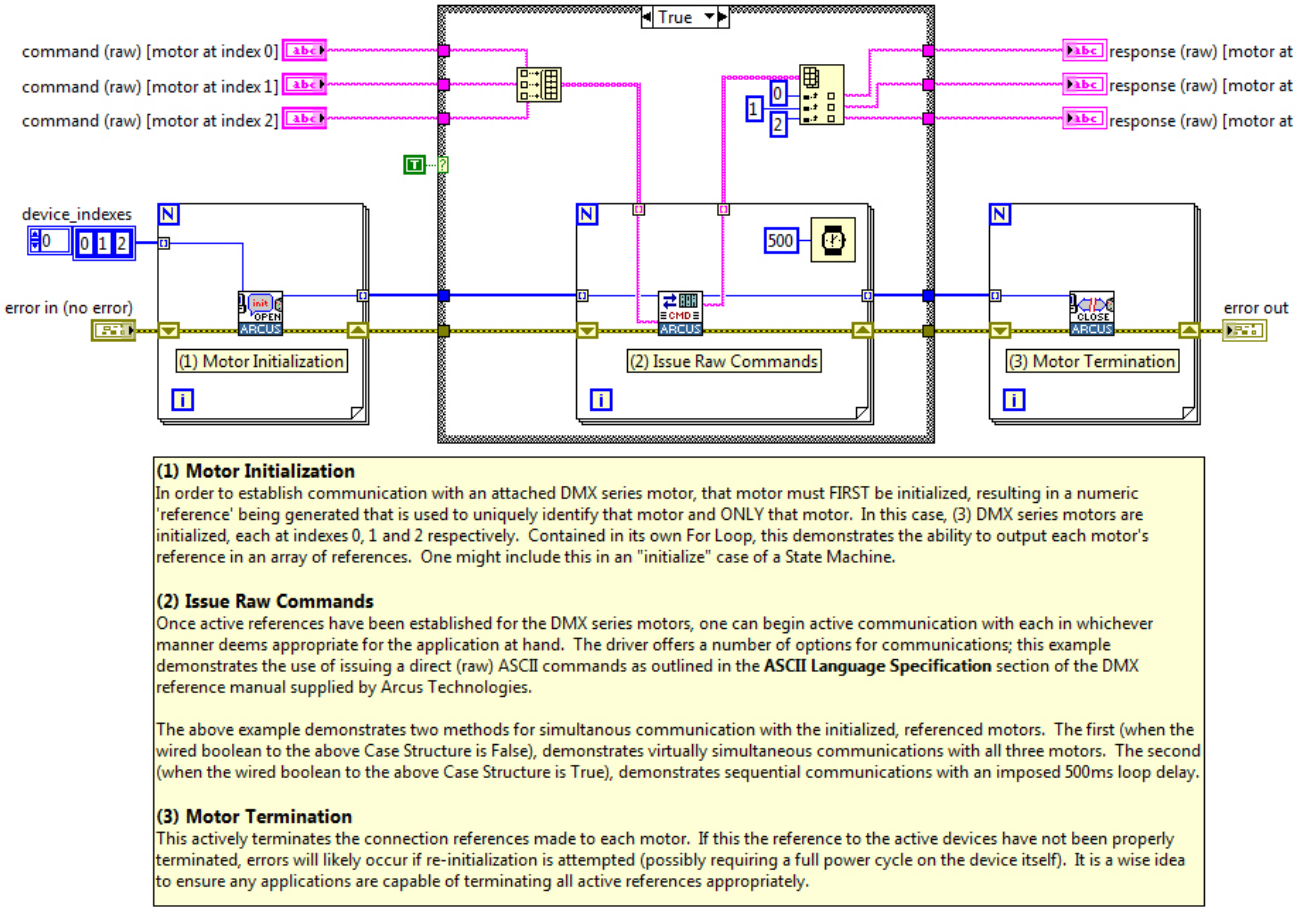


Figure 11, 01_issue_raw_command_string (multi).vi block diagram (sequential commands)

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:

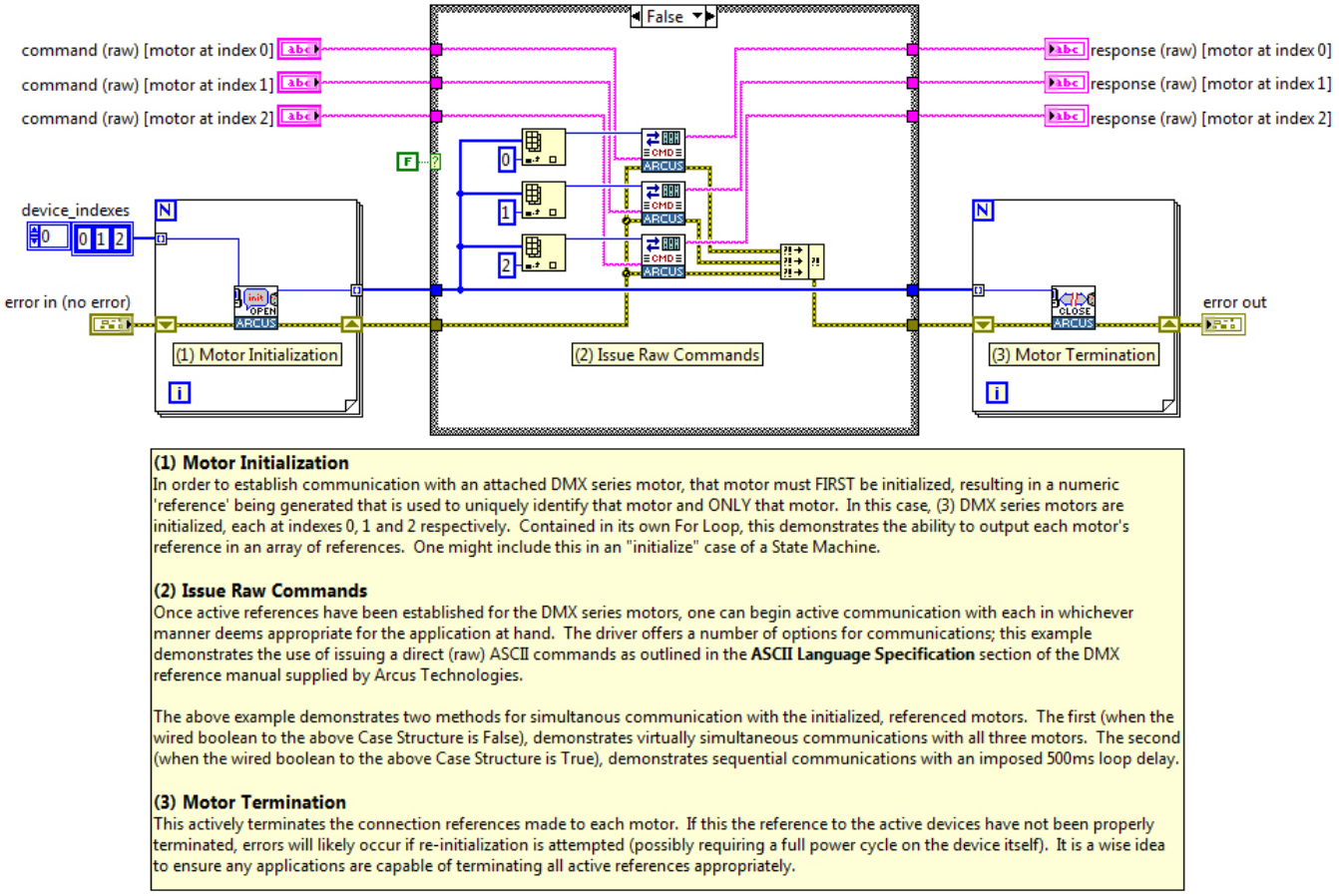


Figure 12, 01_issue_raw_command_string (multi).vi block diagram (simultaneous commands)

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:

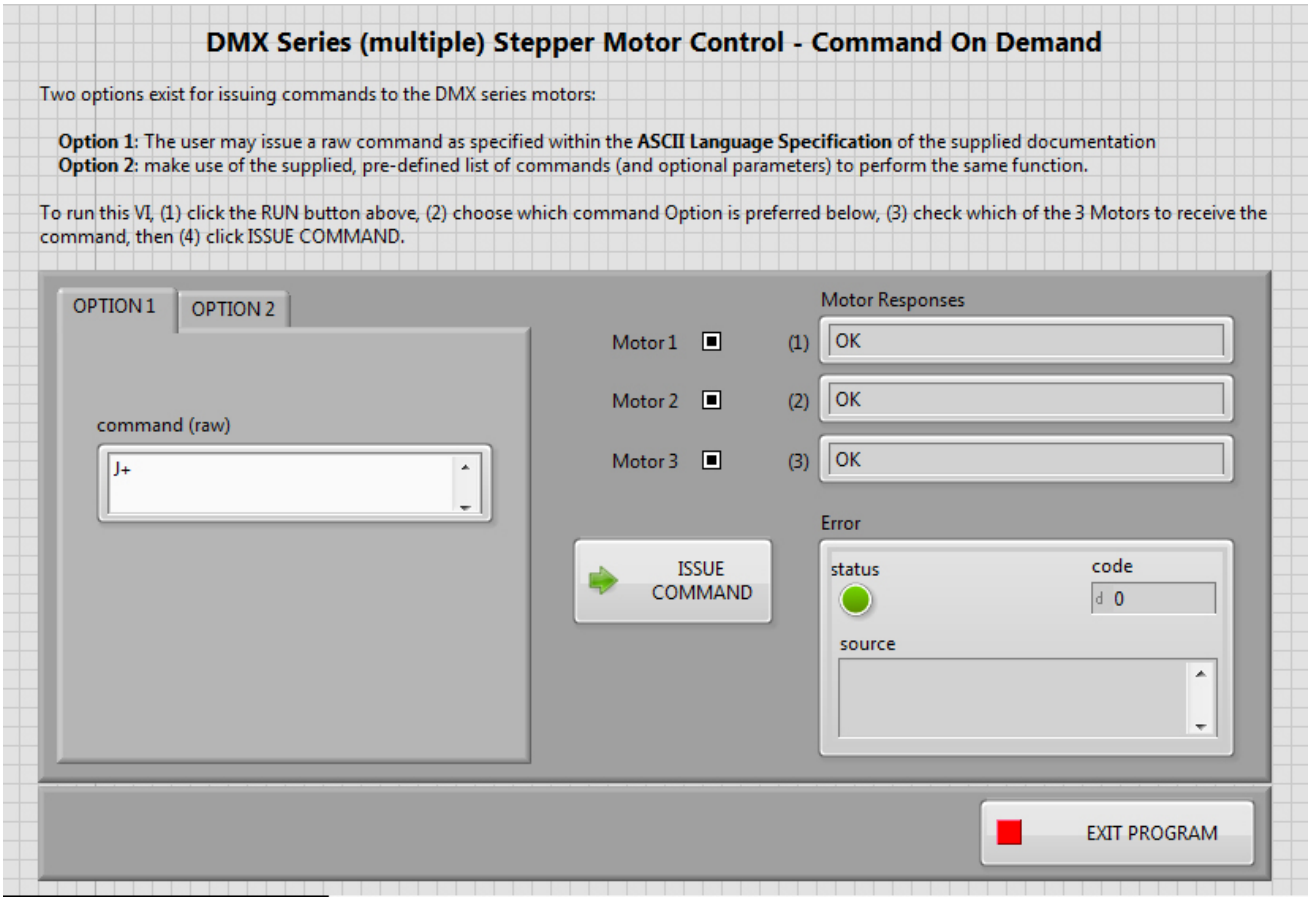


Figure 13, 02_issue_commands_on_demand (multi).vi front panel

Block Diagram:

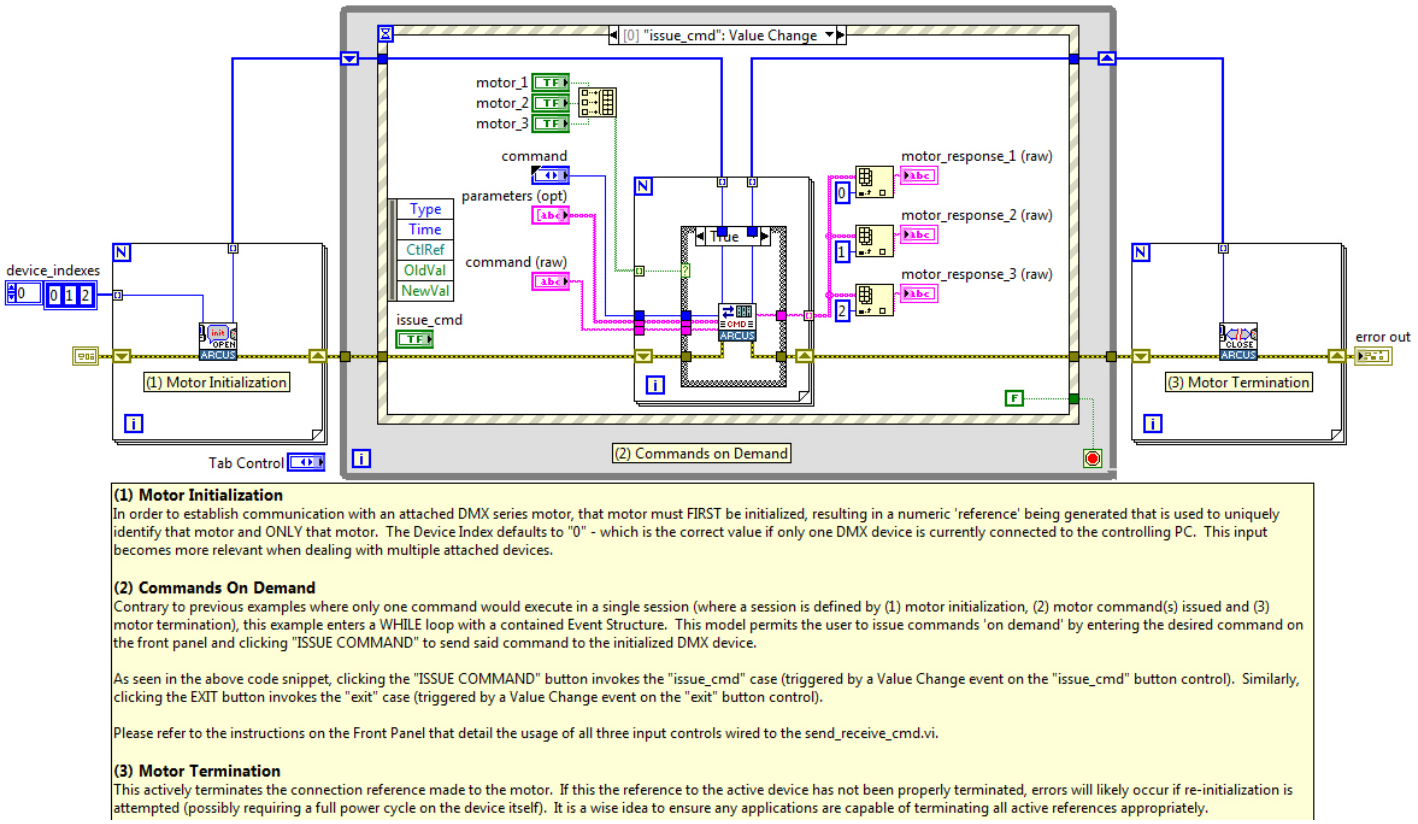


Figure 14, 02_issue_commands_on_demand (multi).vi block diagram

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 beneath the OPTION 1 tab
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, for more sophisticated applications, a mechanism must be in place to identify a motor by both its index and its name (referred to as device ID in this document). Once set, a motors name/ID will never change. However, its index can and likely will depending on the configuration of the application.

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. Now you can imagine an initialization routine that is capable of (1) identifying how many devices are attached to the controlling PC, (2) initializing all motors and (3) retrieving their ID's (names) and associating them to their references. Now, knowing the names of each motor (and the function of each), they can be properly identified in the program, *regardless* of their assigned index. Consider this alternative method for initialization *and* identification, having a look at VI *"03_initialization_and_identification.vi":*

Front Panel:

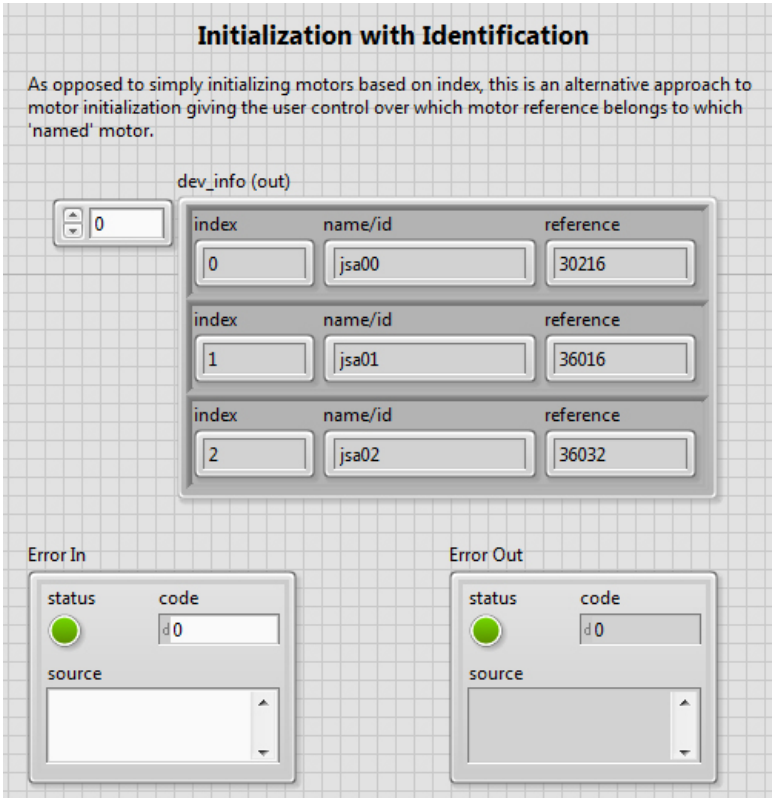
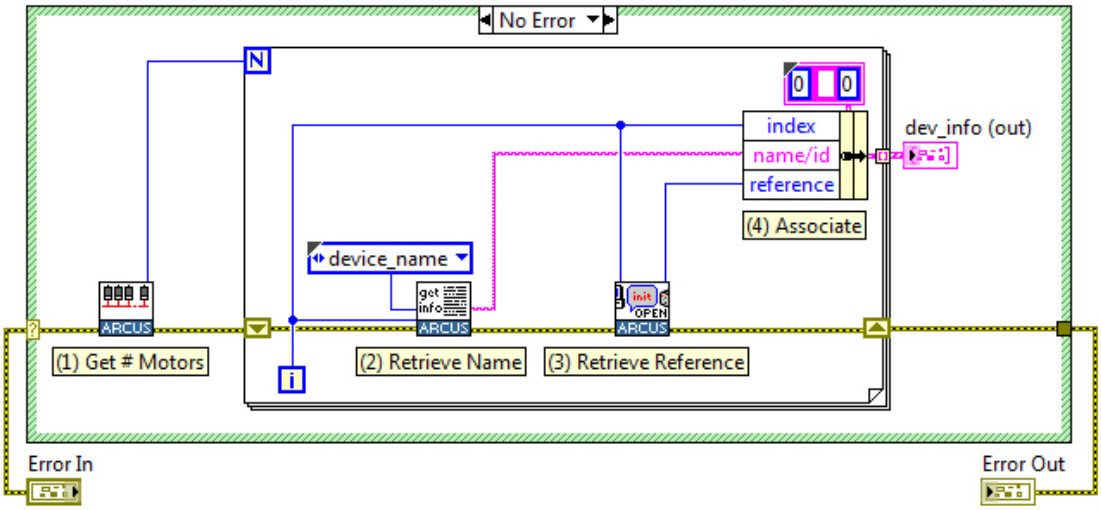


Figure 15, 03_initialization_and_identification.vi front panel

Block Diagram:



(1) Get # Motors
Using the "get_number_of_devices" VI included with the driver set, retrieve the total number of devices connected to the PC. This number dictates how many times to run the corresponding For Loop for identifying each device.

(2) Retrieve Name
Retrieve the name of each attached device.

(3) Retrieve Reference
Initialize each motor, given its index and output a reference used for further communications

(4) Associate
Using a strict type def cluster, combine (for each motor) the motor's index, name and reference. Now, rather than simply referring to a reference for further communications, one may 'identify' a reference from this list of identification information by its corresponding motor's name.

Figure 16, 03_initialization_and_identification.vi block diagram

As an alternative, more thorough approach to initializing the attached motors, one may want to consider using this utility VI supplied in the "dmx_applications.lvproj" project library. Not only does this utility VI initialize all attached motors, but it outputs a complete list of associations where each motor is identified by not just its reference (as in all past examples) but by its reference, its index *and* its name. Now, this permits the developer that ability to make logical business decisions in downstream code based on an unchanging, assigned device name and not just an index or reference.