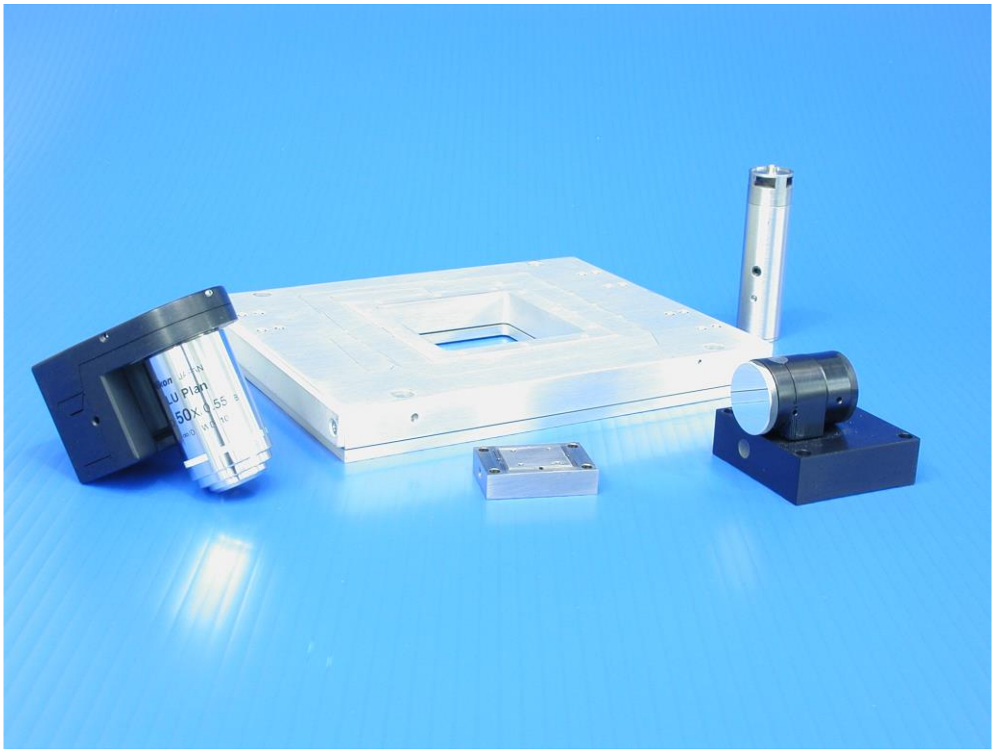
Controlling Mad City Labs' USB enabled nanopositioning devices via National Instruments LabView programming



2524 Todd Drive
Madison WI 53713 USA
Technical Support: sales@madcitylabs.com
www.madcitylabs.com

Copyright 2013-2017

TABLE OF CONTENTS

Chapter 1: Introduction to the Mad City Labs LabVIEW Tutorial

This tutorial is designed to illustrate communication with a Mad City Labs nanopositioning system. The nanopositioning system is equipped with a USB digital interface, and communication is done using LabVIEW on a computer with a Windows operating system. This tutorial will describe, in detail, how to create the Mad City Labs example LabVIEW programs (VIs) from scratch. The purpose is to help Mad City Labs' customers understand how to communicate with the Nano-Drive®controller via USB, how to program certain useful motions, and how the Mad City Labs example VI's are constructed. Completion of this tutorial will require the Nano-Drive®USB software has been installed. If this has been done, a folder exists on your hard drive containing Nano-Drive™ software, manuals, and example VI's (the default directory for this folder is C:\Program Files\NanoDrive\Manuals.) If the USB software has not yet been installed, refer to the "Nano-Drive®USB Quickstart.pdf" file. Your Nano-Drive®is ready for use upon completion of the Nano-Drive®USB Quickstart.pdf.

This tutorial is written for users with a very basic understanding of LabVIEW, but it is not intended to be a comprehensive tutorial for LabVIEW. National Instruments provides several excellent tutorials that should be used to learn about the LabVIEW programming environment. In addition, LabVIEW has excellent help documentation, that the user should become familiar. Each chapter is written as a stand-alone chapter so that readers with more than a very basic understanding of LabVIEW can skip ahead if desired. Each chapter incorporates and builds on ideas from the previous chapters. Note that the tutorial does not demonstrate how to produce every VI on the CD, but if you complete the tutorial, you should be able to build and incorporate any of the VI's provided by Mad City Labs.

LabVIEW can be thought of as a way to make a custom digital controller. This tutorial will illustrate how to set up LabVIEW VI's and how to use them to command your Mad City Labs nanopositioner. It is our hope that after completing this tutorial, you will understand how our example LabVIEW VIs were constructed, how commands are passed from the computer VI to the nanopositioner, and some of the common ways to move a nanopositioner. We also hope that after completing this tutorial you will have some ideas of how to modify these VIs to suit your own purposes.

Before discussing LabVIEW, there are several important concepts that will make your LabVIEW programming tasks easier to understand.

Equipment List

To follow the steps in this tutorial, you will need the following:

1. Mad City Labs nanopositioning stage
2. Mad City Labs Nano-Drive®controller with USB interface
3. Windows computer with Nano-Drive®USB interface driver installed
4. National Instruments LabVIEW installed on your computer

Using a computer to communicate with the Nano-Drive®

Mad City Labs produces a wide range of nanopositioning systems for a variety of applications. All of our nanopositioners may be controlled through the 0 – 10 Volt front panel analog input, but many customers choose to purchase our USB digital interface which allows the user to communicate with nanopositioner controller from a Windows based PC. To make this communication possible, we provide a dynamically linked library (DLL). From our point of view a DLL is a compilation of functions that can be called from a Windows user mode program. A user mode program is usually an executable program that is written in a high level programming language such as C/C++, JAVA, Basic etc., but LabVIEW is also a user mode program. LabVIEW owes much of it's versatility to the fact that it can be used to make function calls to any Windows DLL, and it has a specific procedure for doing this. These function calls are referred to in LabVIEW as a call library function, and the user should consider reading the excellent LabVIEW documentation on the subject. In this tutorial we will demonstrate explicitly how to use call library functions to interact with the Nano-Drive®.

Another important concept of the Windows operating system is the Handle. If you want to pick up your suitcase, you would grab it by the handle. In Windows, if you want to take hold of a device, you also want to grab it by the Handle. In Windows, the device Handle is really an address. In this tutorial, we will show you how to grab a device Handle for the Nano-Drive®. In addition, we will show you how to release the Handle when you are done communicating with the Nano-Drive®

The final Windows operating system concept that is important for our customers is the issue of Windows latency. In user mode programs such as LabVIEW, the Windows operating system controls all interaction between your program and all other parts of the operating system, DLL's, hardware and other programs. User mode programs are not allowed to interact directly with the PC hardware. That includes all USB, RS232, GPIB or PCI bus devices. The operating system only allows your program to talk to hardware through a driver, and the driver can only access the hardware about every 1 millisecond. This means that you can only send or receive information from your hardware device, like the Nano-Drive®, every millisecond or so, and this is Windows latency. In most cases, Windows latency is not an issue, but sometimes it is a problem. In this tutorial, we will show you how to overcome some aspects of Windows latency by using waveform data acquisition. With waveform acquisition, you can achieve more accurate timing than is possible in LabVIEW.

How do you want to use your nanopositioner?

We are always amazed by the diversity of applications for our nanopositioning products, and the creativity or our customers in finding new applications for their use. Unfortunately, this means that we don't know how you wish to use our product, what experiments you wish to perform, and what kind of data should be the result. Our reaction to this quandary has been to produce general LabVIEW examples that meet the basic requirements of many of our customers, but none of the LabVIEW examples have been designed to do your experiment. We provide the examples so that you can build more complex programs that are tailored to your exact requirements. We provide several examples for building a scanning program, but you will have to decide whether to use a saw tooth scan or a triangle scan for your application. In addition, you will have to make decisions about scan speed, scan accuracy and settling time. Should you decide that you are not up to the task of writing a LabVIEW program for your system, Mad City Labs, Inc. may be able to provide a custom solution that meets your needs.

What are the specifications of my nanopositioner?

When your system was purchased, the issues of resolution were no doubt discussed with one of our sales representatives, but you the system user may not have been involved in that discussion, and you may not know or have access to the specifications of your system. The easiest solution to this problem is to call Mad City Labs, Inc. and tell us the serial number of your stage or the serial number of your controller, and we can tell you the specifications for your stage. In addition, most of this information is written into the EEPROM of the USB interface, and can be accessed via a LabVIEW VI.

When you move the stage, you are really changing a command voltage with a DAC (digital to analog converter). This DAC has either 16 bit or 20 bit resolution. If your system has a full range of motion of 100 microns and a 16 bit DAC, then the smallest step, S, you can perform is;

$$S = 100\ \mu m / 65536 = 1.6\ nm.$$

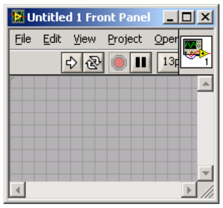
When you measure the position of the stage, you are actually measuring the position sensor output voltage. Similar to the DAC if you wish to measure your position with a 16 bit ADC (analog to digital converter), then the smallest movement you can resolve is also 1.6 nm. Now these conversions are made by MadLib.dll for you, so you don’t need to program the conversion directly, but it is important to understand how position measurements are made.

Where is the documentation for MadLib.dll?

The documentation for MadLib.dll is included on your distribution disk under the file name Madlib_*_*.doc, but it is also included in Appendix A of this document for quick reference.

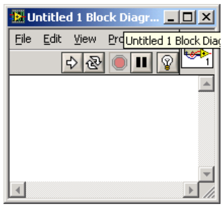
Chapter 2: Introduction to the LabVIEW Programming Environment

We introduce the structure of LabVIEW programs and some basic and concepts to begin creating your own programs. This chapter is a brief introduction to LabVIEW programming. Subsequent chapters relate to specific programming examples for Mad City Labs nanopositioners. Users already familiar with LabVIEW may wish to skip ahead to Chapter 3. LabVIEW programs are often created to imitate the function and appearance of a physical instrument. A LabVIEW program, called a VI (Virtual Instrument,) consists of two separate windows, the Front Panel and Block Diagram.



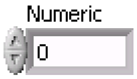
The Front Panel

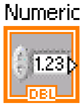
Every new VI is automatically created with a Front Panel interface. The Front Panel contains the controls and indicators which allow the user to manipulate the operation of the VI. Any Control or Indicator created here is also automatically placed on your Block Diagram.



Block Diagram

Every new VI is automatically created with a Block Diagram. The Block Diagram contains the code to activate the VI. The values of Front Panel components can be read and assessed within the Block Diagram. The Block Diagram can also pass data to the Front Panel to be displayed within graphical indicators. Think of the Block Diagram as the guts of your virtual instrument. After all the Block Diagram is where you wire things together.



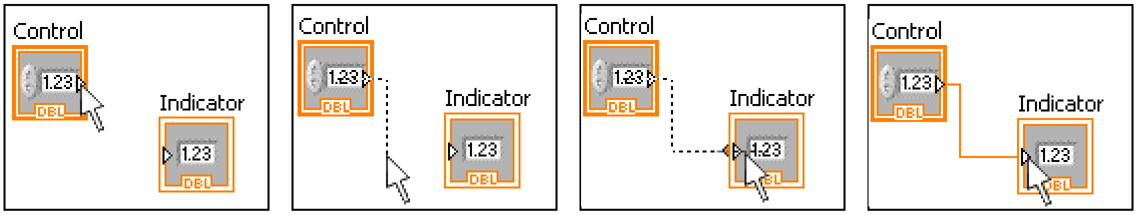


Controls and Indicators

Any parameters of the LabVIEW program that must change according to the user's specifications should be editable via the Front Panel. This is accomplished by first creating a user-friendly control on the Front Panel, then using the control within operations on the Block Diagram. Seen at left are two images of the same Numeric Control. The top image was taken from the Front Panel, the bottom from the Block Diagram (the Value '1.23' seen inside the Block Diagram representation of the control is not real; the true value can only be seen from the Front Panel.) The essential difference between a control and indicator is that a control inputs values to the block diagram while the indicator outputs values from the block diagram to the front panel.

Wire Connections

After a component of the Block Diagram performs an operation, any data it has generated can be passed to other Block Diagram components via wire connections. Therefore, the LabVIEW Block Diagram will contain a mixture of components interconnected with wires. The sequence of images below demonstrates the method of creating wire connections in LabVIEW.



Adding Components to the Front Panel/Block Diagram

Controls and Indicators are examples of components that can be added to either the Block Diagram or Front Panel of the LabVIEW VI. However, many components exist which can be added to the Block Diagram but not the Front Panel, and vice versa. LabVIEW allows the programmer to add *Controls* to the Front Panel, and *Functions* to the Block Diagram. Therefore, anything to be added to the Front Panel is found within the Controls Palette (including indicators), and anything to be added to the Block Diagram can be found within the Functions Palette.

The Controls Palette

The Controls palette is available only on the front panel and contains the controls and indicators you use to create the front panel. This makes sense because you are trying to create a virtual instrument, and the knobs, displays and switches always go on the front panel. Don't you hate it when a designer puts a switch on the back panel of an instrument, and it you have to reach through cables and power cords to turn the switch on and off. Fortunately there are no back panels in LabVIEW.

The Functions Palette

The Functions palette is available only on the block diagram, and the Functions Palette contains the VIs and functions you use to build the block diagram. The basic palette (depending on the level of your LabVIEW license) contains a wealth of preprogrammed functions which can be used to construct your virtual instrument.

You should familiarize yourself with the many functions and controls available in the basic version of LabVIEW, and remember that if LabVIEW does not provide the functionality you require, you can create itself.

Chapter 3: Creating a LabVIEW VI to Control a Nano-Drive®Axis

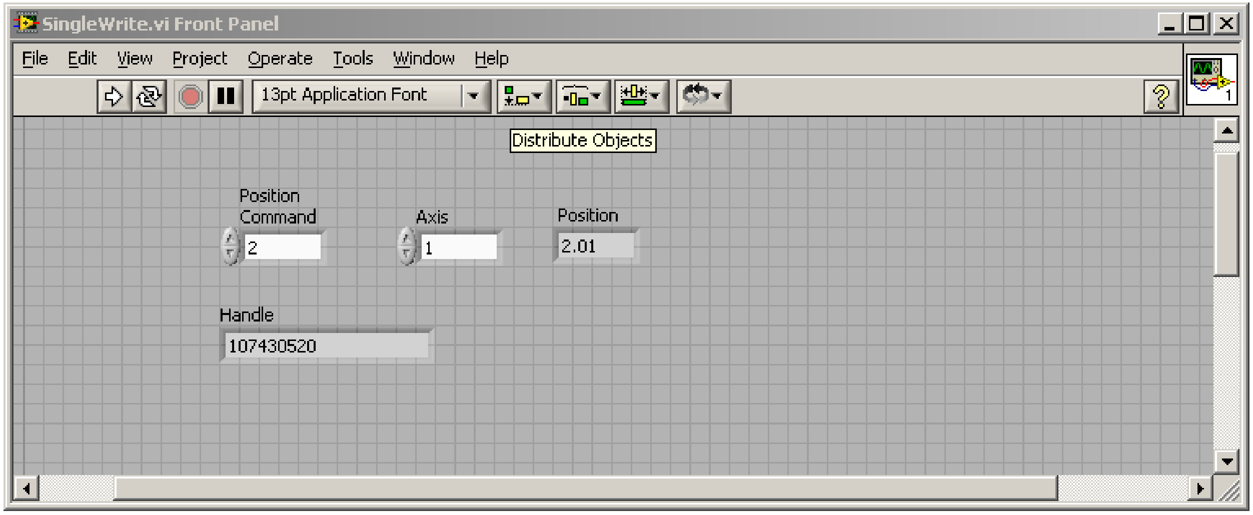
You will be creating a LabVIEW VI that takes a user entered value and writes this value to the Nano-Drive®. The Nano-Drive®USB interface interprets a write command as a position in microns, however it is important to remember that you are really changing the value of a DAC (Digital to Analog Converter). The resolution of the position command is therefore determined by the underlying resolution of the DAC (either 16 bits or 20 bits). In addition, the smallest step size of your stage will be determined by the DAC resolution and the range of motion of your stage (in microns). After changing the position, the VI then waits 100 milliseconds before reading the Nano-Drive® position. The read command is also interpreted as a position in microns, but you are really reading the sensor position with an ADC (Analog to Digital Converter). Like the DAC, the ADC has a finite resolution (either 16 bits or 20 bits). There is nothing magical about the 100 millisecond delay it simply allows the system to settle completely. Because nanopositioning stages are slower than the control electronics, it is always important to consider whether the stage has "settled" to the desired position before reading the position. Finally, the position data are printed to the monitor giving the position of the stage.

Topics Include:

1. Basic editing of the Front Panel of a LabVIEW VI.
2. Basic editing of the Block Diagram of a LabVIEW VI.
3. Interfacing with the Nano-Drive®using the MADlib.dll.
4. Referencing the Madlib_*_*.doc for function information (see Appendix A as well).
5. Using Structures, Timers, and other LabVIEW components to control dataflow.

Accessing functions within the MADlib.dll via LabVIEW's Call Library Function Nodes

The VI shown below will be created in Chapter 3.



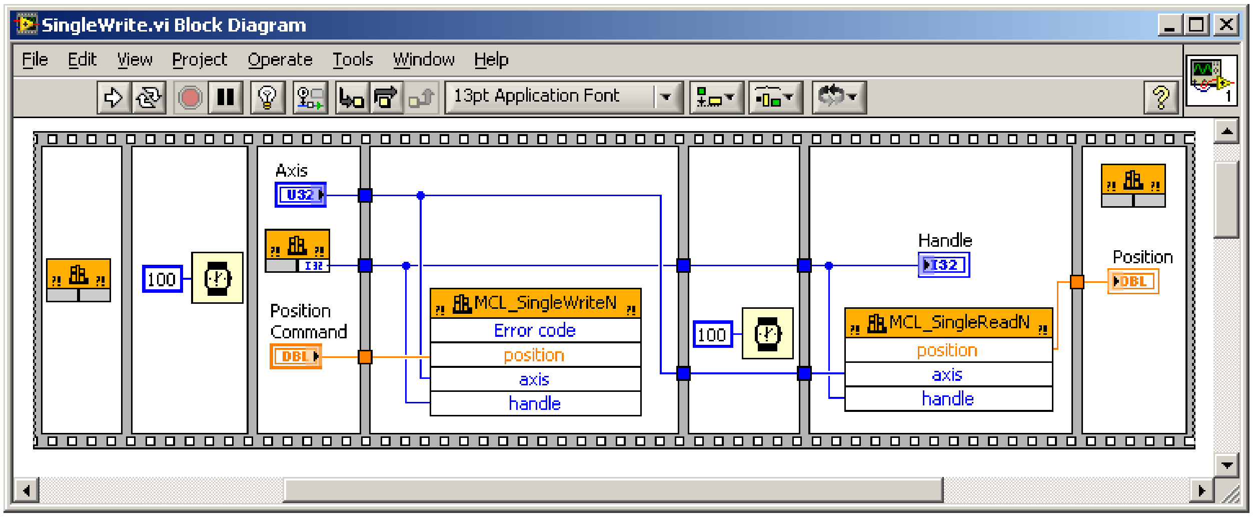
Position Command: 2

Axis: 1

Position: 2.01

Handle: 107430520

(Image of the finished Front Panel by the end of Chapter 3)



(Image of the finished Block Diagram by the end of Chapter 3)

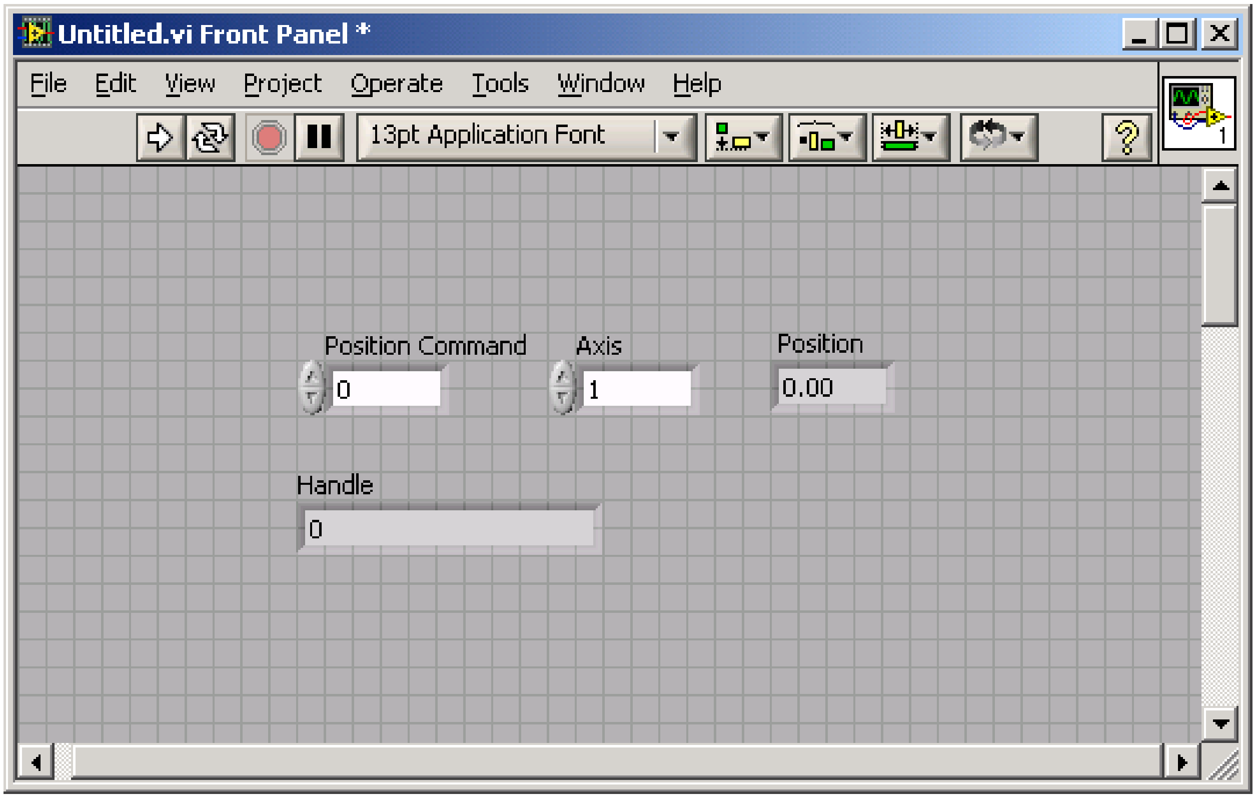
Section 3.1 - Preparing the Front Panel interface

Project Goal

In this section you will create a Front Panel containing all the necessary elements to make some simple Nano-Drive® read and write operations. The Block Diagram will not be edited within this section. Completion of all steps within this section will result in a Front Panel equal to the one depicted below.

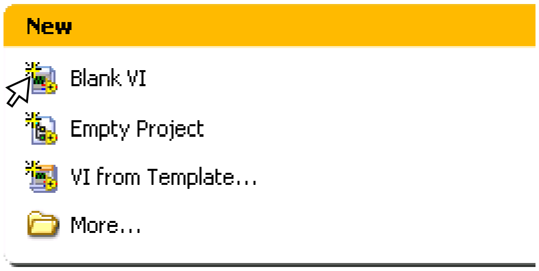
Note

Although the properties of the Numeric Controls/Indicators are modified within this section, the modifications have been limited to data type and label. Experienced LabVIEW users are encouraged to skip the intermediate steps and build the application according to the picture and description presented at the beginning of this section.


Untitled.vi Front Panel *
File Edit View Project Operate Tools Window Help
13pt Application Font
Position Command
0
Axis
1
Position
0.00
Handle
0

(photo of the finished Front Panel following completion of Section 3.1)

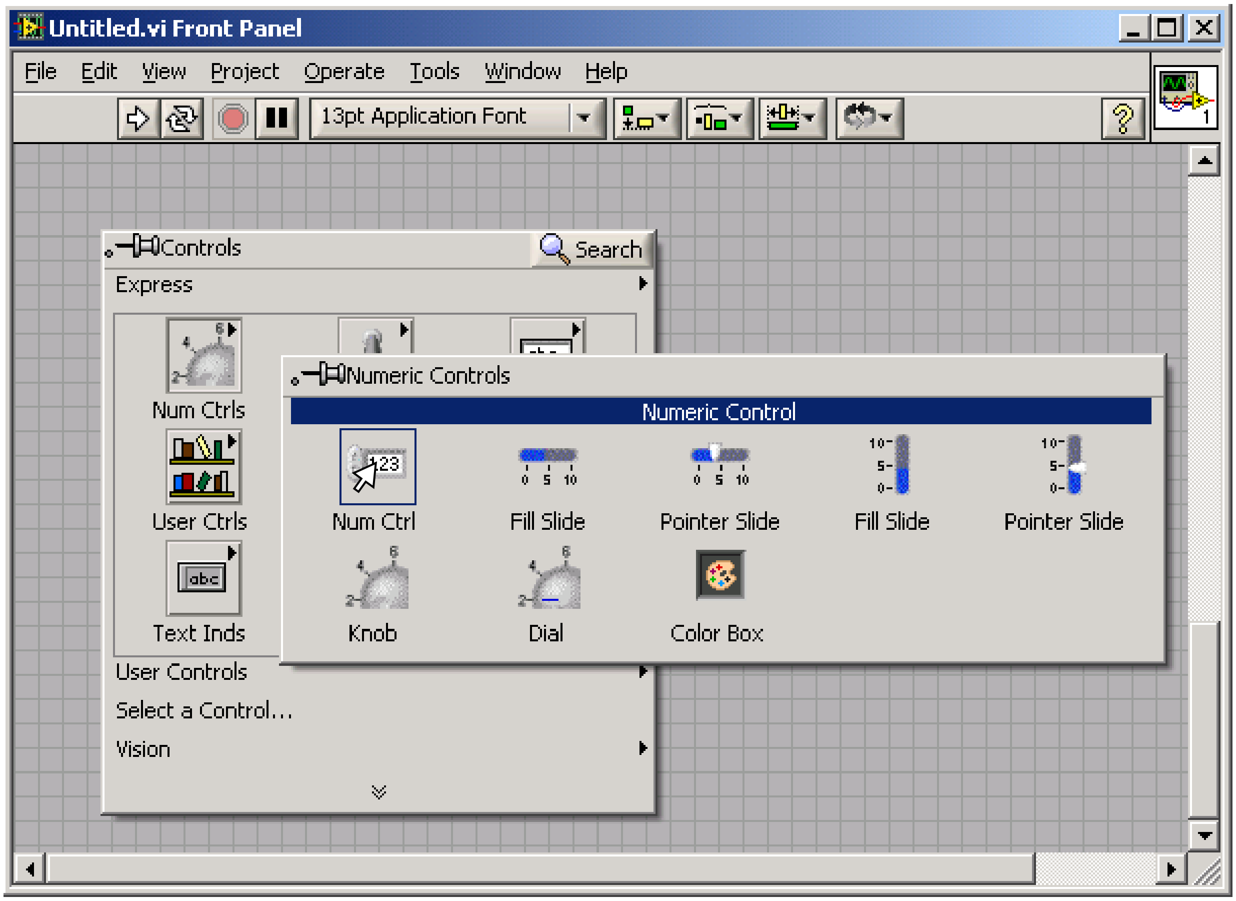
3.1.1 Start a new VI, and create the two Numeric Controls



- Begin LabVIEW and start a new VI by selecting *Blank VI* from the *Getting Started* Window. The Front panel and Block Diagram of a new VI appear.

- Right Click anywhere within the Front Panel area (seen below) to access the *Controls* menu.

- Hover your cursor over the *Num Ctrl* button. The *Numeric Controls* window appears on top of the Controls window.

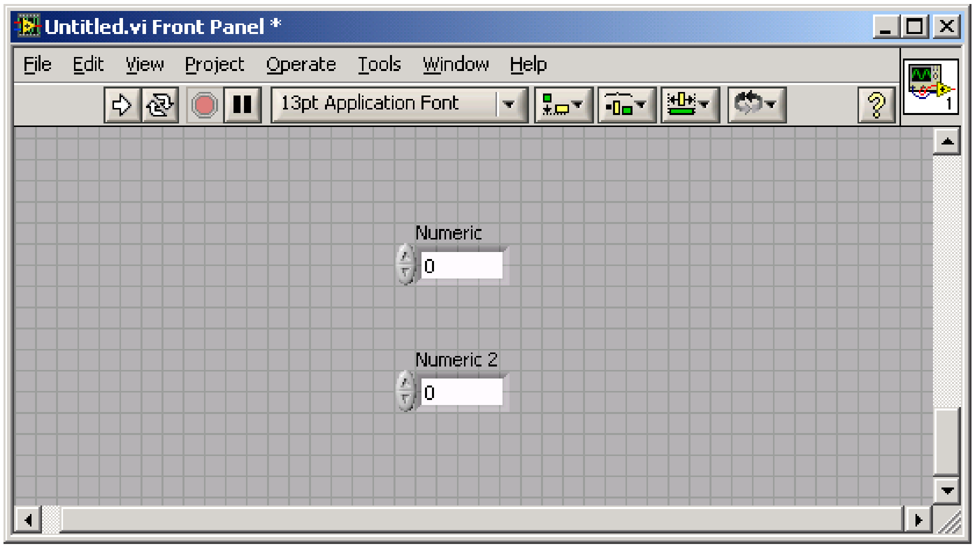


Note
If you’re having trouble locating a specific control, left click the *Search* button to search by name via an automated search.

- Left click the *Num Ctrl* button that appears on the Numeric Controls window. The windows vanish and your cursor takes the shape of a hand. You are now holding the footprint of a Numeric Control which will later be used as your position control.

- Left click anywhere within the front panel to place the Numeric Control.

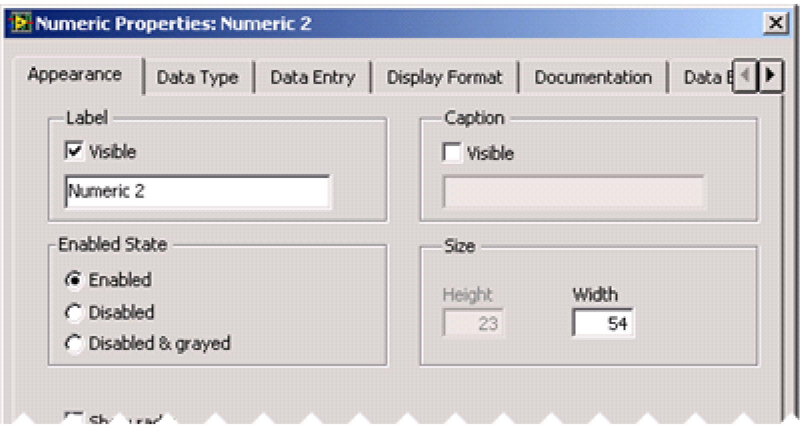
- Follow the same process to place another control next to this one. The VI should now look similar to the one below.


Untitled.vi Front Panel *
File Edit View Project Operate Tools Window Help
13pt Application Font
Numeric
0
Numeric 2
0

Note
The Numeric Controls we've added to the Front Panel can be incremented/ decremented regardless of the state of the program (running/not running) using the arrows at left of the text field. However, on the Block Diagram, controls cannot be the recipient of data. Their use is limited to *holding* data from the user (Front Panel) for use within the Block Diagram.

3.1.2 Editing Data Type

- Right click one of the Numeric controls and select *Properties* from the pop-up menu. The *Numeric Properties* window (seen below) opens into the *Appearance* tab.



Label
☑ Visible
Numeric 2

Caption
□ Visible

Enabled State
● Enabled
○ Disabled
○ Disabled & grayed

Size
Height 23
Width 54

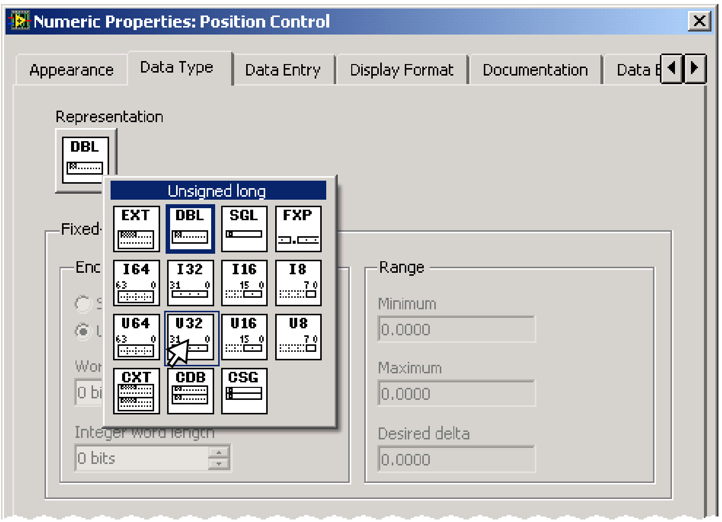
- This Control will be used to select the axis to move, so change the text within the *Label* text field from "Numeric" to "Axis".

- Next, click the *Data Type* tab.

- Left click the data type representation button, and select *U32* from the menu that appears.

Note
It's important to match the data type representation of the control to the data type expected by any functions using the control. An axis of the Nano-Drive® is accessed via a function within the MADlib.dll, and is always identified as a 32-bit unsigned integer. The format of the function to be used must always be followed as it appears within the MADlib.dll. For a list of all functions, and their formats, refer to the Appendix A or Madlib_*_*.doc.

- Click the *OK* button near the bottom of the Numeric Properties menu to accept the changes and exit the menu.
- Right click the other Control and select *Properties* from the menu that appears.
- Change its label to "Position command."
- Change the data type to *DBL*. The MADlib.dll expects position commands to be of data type Double.



- Click *OK* to accept the changes and exit the menu.

Position command and Axis selection using LabVIEW Numeric Controls
The simplest MADlib.dll functions require only three inputs: the Handle of the Nano-Drive®, an axis to move, and a position to move to. The Handle will be acquired by code on the Block Diagram. Therefore, the Position Command and Axis will be the only controls you will need to make.

The Handle
The Nano-Drive®manual provided with your system explains in detail the necessary role of the Handle; however, at this point it would suffice to understand that to communicate with a Nano-Drive® we must first identify it by its Handle, and before we can identify a Nano-Drive® by its Handle we must first command Windows to specify a Handle for the Nano-Drive®.

3.1.3 Create the two Numeric Indicators

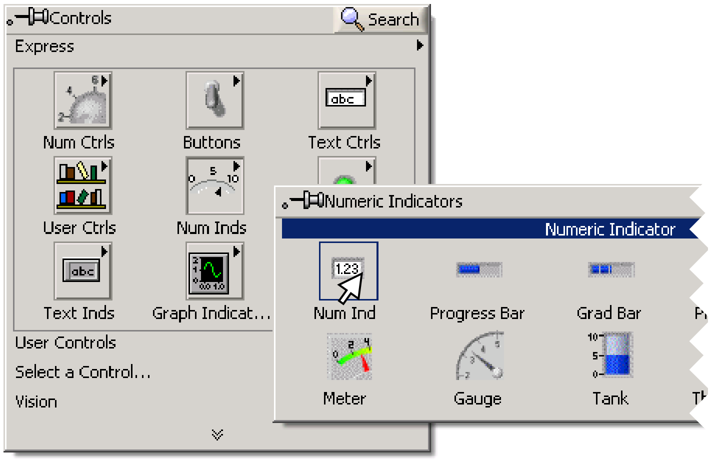
You will now create the *Numeric Indicator* used to display position data onto the Front Panel. Also, you'll be instructed to create a Numeric Indicator to display the *Handle* the VI acquires to communicate with the Nano-Drive®.

Retrieve Position Data from the Nano-Drive™
For your VI to fully interact with the Nano-Drive®controller, it must not only send position command data to the Nano-Drive® DAC, but should also read and display data extracted from the Nano-Drive® ADC. The data extracted from the ADC will be displayed on your Front Panel using a *Numeric Indicator*.

Introduction to the Nano-Drive®ADC
Each axis of the Nano-Drive® has an ADC to read the position of the axis. The ADC constantly reads the position and stores the data within internal memory. When the Nano-Drive® is commanded to a position, it is often useful to read the data coming from the ADC to verify the Nano-Drive® has arrived successfully.

Introduction to reading the ADC using the MADlib.dll
The MADlib.dll contains several functions we can use to read the ADC data from the internal memory. The ADC data can then be displayed on your Front Panel via a numerical or graphical indicator, giving you the ability to monitor the position of your Nano-Drive®.

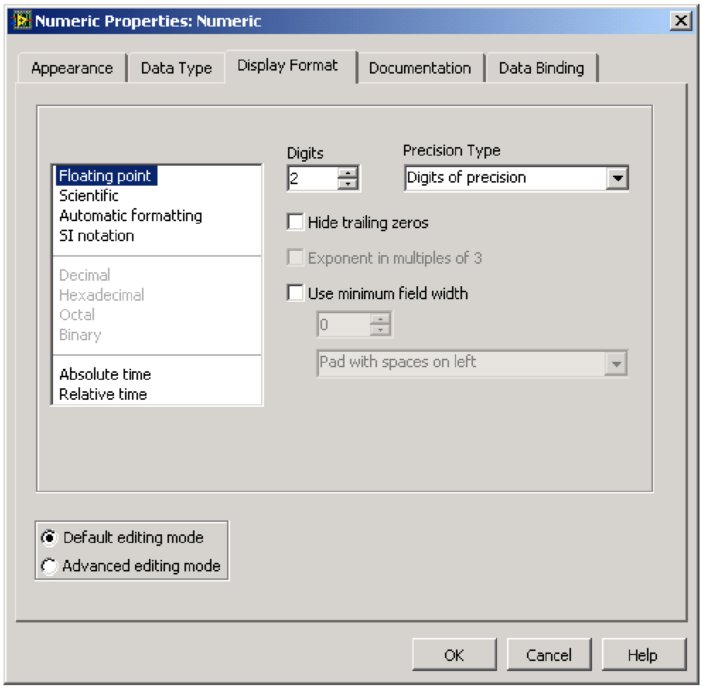
- Right click over any empty area of the Front Panel and hover your mouse over the *Numeric Indicators* button. Two menus should be open now, the newest being the Numeric Indicators menu.
- Left click the upper-left most icon; the *Numeric Indicator*.
- Left click again to place the Numeric Indicator.
- Follow the same process to add another indicator next to this one.



- Edit the properties of the first indicator. Make its data type *DBL*, and change its label to "Position". This indicator will show the position of the axis.
- Edit the properties of the second indicator. Make its data type *I32*, and change its label to "Handle."

3.1.4 Editing Display Format

You will notice your Position Indicator still doesn't exactly match the Position Indicator depicted in the image of the finished Front Panel (seen at the beginning of this section.) You will achieve this appearance by editing the *Display Format* of the Indicator.



Floating point
Scientific
Automatic formatting
SI notation
Decimal
Hexadecimal
Octal
Binary
Absolute time
Relative time

Digits: 2

Precision Type: Digits of precision

□ Hide trailing zeros

□ Exponent in multiples of 3

□ Use minimum field width

0

Pad with spaces on left

● Default editing mode

○ Advanced editing mode

- Re-enter the properties menu of the Position Indicator. Click the *Display Format* tab.

- Select *Floating point* from the list of display types on the left side.

- Set the number of digits to display to 2.

- Set *Precision Type* to "Digits of precision".

- Uncheck the *Hide trailing zeros* option.

- All other options should be most appropriate at their default settings. However, verify that your Position Indicator's properties are identical to those displayed in the image to the right before continuing.

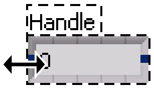
- Click *OK* to accept the changes. The indicator should now appear identical to the Position indicator which appears in the image at the beginning of this section. It should display three digits with a decimal point.

Section 3.1 Completed

We have completed the goal of Section 1. Our Front Panel now contains all the elements required to proceed into editing the Block Diagram. Also, the properties of all our controls and indicators have been set appropriately for their future applications.

Note
Before proceeding, you could arrange the icons on the Front Panel to match the image presented at the beginning of this section, or into some other arrangement that you like.

Resizing Indicators/Controls



The handle is typically a very large number and may not fit within the default display size of the Handle Numeric Indicator. Drag a selection box over the indicator to select the entire indicator (versus entering its text field, or label editor). Hover your cursor over the indicator, a blue box appears on the left and right side of the indicator. You can now expand the indicator horizontally by grabbing and dragging the blue box.

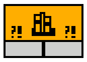
Section 3.2 - Preparing the Block Diagram

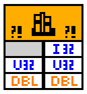
By the end of this section, you will have a Block Diagram equal to the one depicted below. Once again, experienced LabVIEW users are encouraged to assemble the Block Diagram according to the picture. The dll function calls appearing in this image are: (from left to right):
1. *MCL_ReleaseAllHandles* **2.** *MCL_InitHandle* **3.** *MCL_SingleWriteN* **4.** *MCL_SingleReadN*
5. *MCL_ReleaseAllHandles*.

Introduction to the Block Diagram

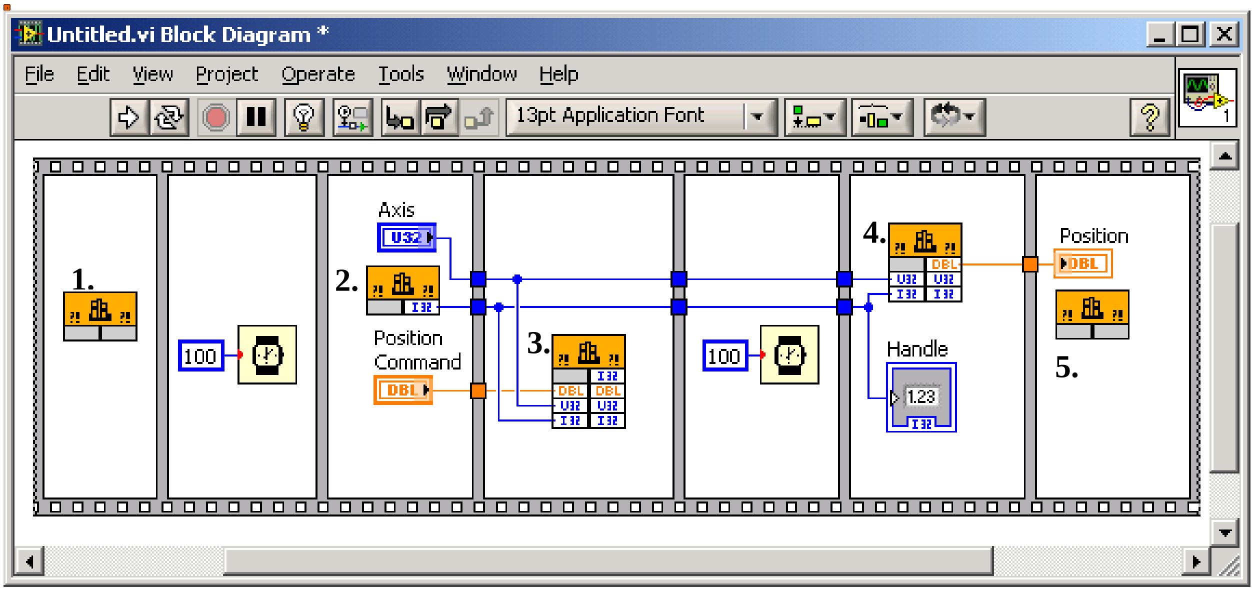
The Block Diagram of a LabVIEW VI contains the code to activate the Front Panel. The components and connections of the Block Diagram are essentially the inner-workings of the virtual machine. Code execution generally flows from left to right. The first component to activate in the image below would be the dll function call to MCL_ReleaseAllHandles. The next component to activate would be the millisecond time delay, which accepts the constant, 100, as the amount of milliseconds code execution will be delayed.)

DLL (Dynamically-Linked Library) Function Calls in LabVIEW





The Call Library Function Nodes appearing in the image below, and at left, call a unique instance of a function existing within the Madlib.dll. Simply put, the Madlib.dll can be thought of as a container of tools used to communicate with the Nano-Drive®. In LabVIEW these tools are accessed via Call Library Function Nodes. At left are images of two separate Call Library Function Nodes. The lower one calls a function with input/return values. The user is encouraged to read the LabVIEW documentation for a more detailed explanation of call library functions.



(photo of the finished Block Diagram following completion of Section 3.2)

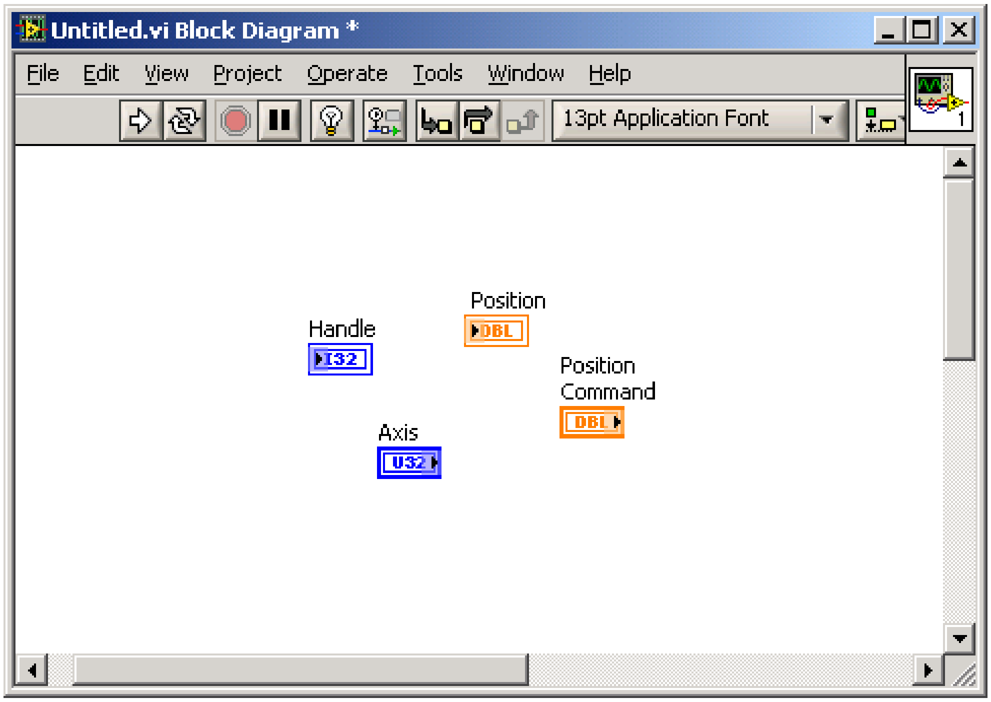
Note
Many Block Diagram elements are meant to imitate C programming elements; such as while loops, for loops, and data arrays. However, there exist distinct differences in the LabVIEW adaptation of these elements. Read the LabVIEW help file on an element before implementing it to avoid misuse due to a counterintuitive property.

3.2.1 Accessing and Understanding the Block Diagram

- Hold *Ctrl* and press *E* to access the Block Diagram from the Front Panel. Alternatively, you can toggle between windows using the menu bar item: *Window>>Show Front Panel*.

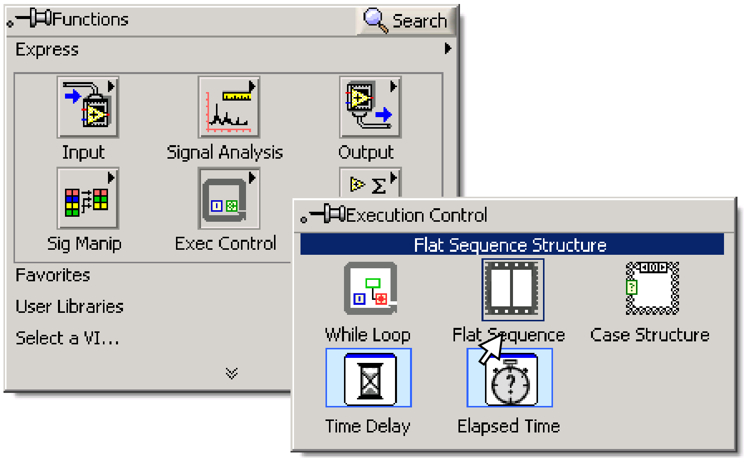
Block Diagram: Controls and Indicators
Each time a control or indicator is created on the Front Panel its Block Diagram representation is created here. The four elements you see correspond to a Front Panel Control or Indicator of the same name which we created in the previous section (your icons may appear larger and more box-like; the difference is purely visual.)

It's important that your Block Diagram very closely matches the one depicted below. The arrangement, and icon appearance, of the components is irrelevant at this point. Pay particular attention to the data types (U32, I32, DBL,) and whether the arrow appears on the left or right side of the component (showing the difference between a Control and an Indicator.)


Untitled.vi Block Diagram *
File Edit View Project Operate Tools Window Help
13pt Application Font
Handle
I32
Position
DBL
Position
Command
DBL
Axis
U32

Note
If at any time you wish to switch back to the Front Panel, hold *Ctrl* then press *E*. Alternatively, from the Front Panel menu bar, select: *Window>>Show Block Diagram*.

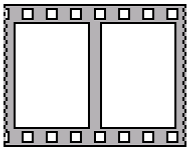
3.2.2 Create the Flat Sequence Structure



- Right click any empty area of the Block Diagram. The *Functions* menu appears.
- Hover your mouse over the *Exec Control* icon. A second window appears on top of the first.
- Left click the *Flat Sequence* icon. The windows vanish and your cursor is replaced with a box traced with a dashed line.

- Left click any empty space within your Block Diagram, move the mouse about four inches diagonally and left click again to finish creating a Flat Sequence Structure of a size equal to the box you just dragged (alternatively, you could have left clicked and held to create the structure. Upon releasing the left button the Flat Sequence Structure appears.)

The Flat Sequence Structure

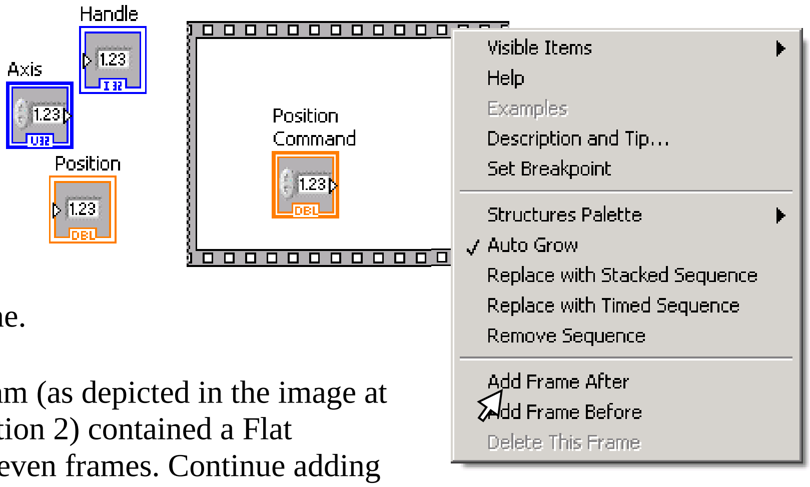


Creating the Flat Sequence Structure has fundamentally changed the relationships between your components. Inside the Flat Sequence Structure execution flow happens in stages; outside the Flat Sequence Structure execution cannot be so clearly defined.

Adding Components to the Flat Sequence Structure

Some components may have been automatically inserted inside your Flat Sequence Structure. This is because they were overlapped by the creation box while you were picking an initial size for the Flat Sequence Structure. If you wish, you can drag them back out, or you could put others in. Try putting some in, and then drag the Flat Sequence Structure around. You will see that the components inside move with the box. However, if you drop the Flat Sequence Structure on top of outsiders; either the component, or the Flat Sequence Structure, will appear hovering above the other.

3.2.3 Add Frames to the Flat Sequence Structure



- Right click the top, or bottom, grey border of the Flat Sequence Structure.

- Select *Add Frame After* from the list that appears. A second frame appears after this one.

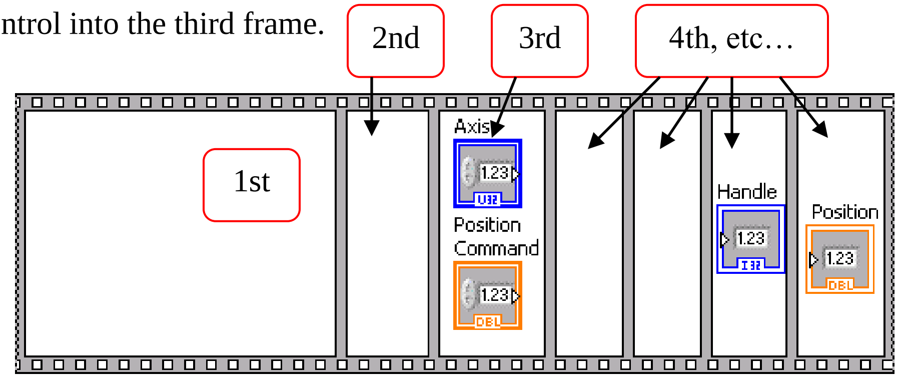
- The finished Block Diagram (as depicted in the image at the start of Chapter 3, Section 2) contained a Flat Sequence Structure with seven frames. Continue adding frames to your Flat Sequence Structure until you reach seven.

- Let’s continue to match the image of the finished Block Diagram. Drag the ‘Position Command’ Control into the third frame (the Flat Sequence Structure will auto-expand to fit the control if need be.)

- Drag the “Axis” Control into the third frame.

- Drag the “Handle” Indicator to the sixth frame.

- Drag the “Position” Indicator to the seventh frame.



- Verify your diagram appears very similar to the image seen above. The numbered boxes distinguish what is meant by “frames” of the Flat Sequence Structure.

3.2.4 Add a Timer

The diagram we are building makes use of Timers to synchronize data flow. Without Timers, we would have no way of guaranteeing stages of our program won’t mistakenly overlap each other, producing corrupted data. We will use a Millisecond Timer.

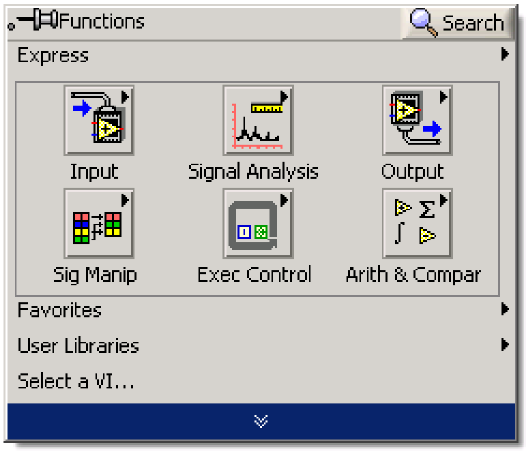
The *Wait (ms)* timer

The Millisecond Timer halts execution of the Block Diagram for a number of milliseconds equal to the numerical value passed to its input. This type of Timer requires an input to function. The input could come from a Numeric Control, allowing the user to alter the wait time from the Front Panel. However, to keep the example simple, this tutorial uses a Numerical Constant.

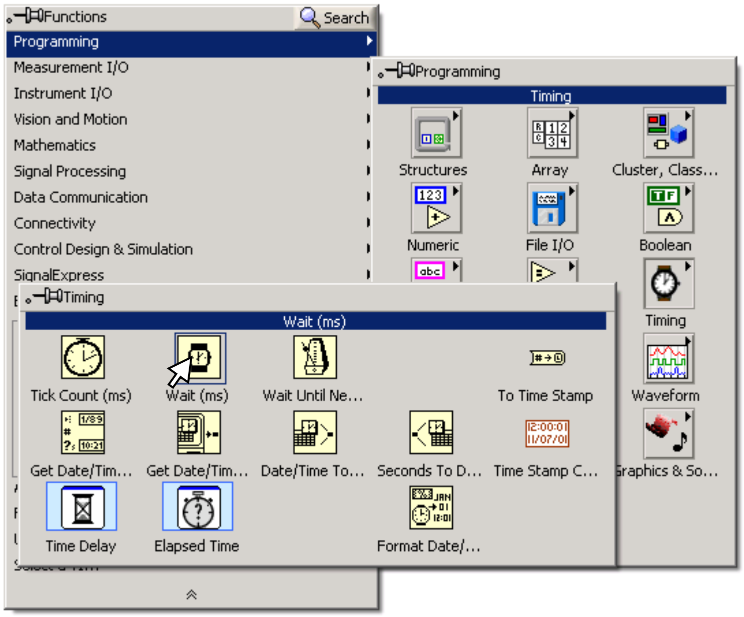
- Right click any empty area within the Block Diagram to open the *Functions* menu.

- Left click the bottom item of the menu (a double down arrow) to expand the menu. We can now see the full list of LabVIEW function categories.

- Hover your cursor over the *Programming* category. A new menu appears containing all sub-categories associated with programming.



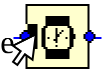
- Hover your cursor over the *Timing* icon. A third window appears containing an assortment of timing functions.

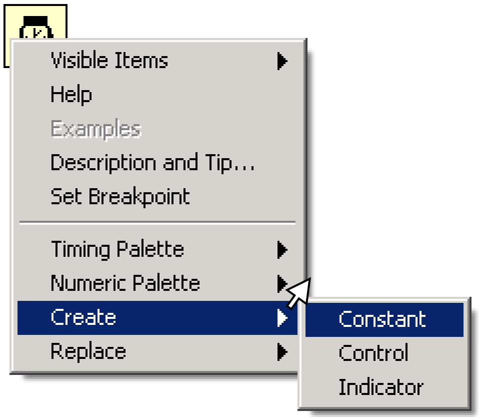


- Left click the *Wait (ms)* icon. The three windows vanish, and you should now be holding the icon for a timer.

- Bring the icon inside the second frame of the Flat Sequence Structure and left click to place the timer there.

- Expand the second frame of the Flat Sequence Structure to add some working room. To expand the frame; bring your mouse cursor within the frame so that eight small resize boxes appear along the border. Left click and horizontally drag a left or right side box to your desired size.

- Hover your cursor over the Timer. The input (a blue dot) should appear on the le  side of the Timer.



- Hover your cursor over the input (left terminal) of the Timer and right click. A menu opens (as seen to the right).

- Hover your cursor over the *Create* category. This gives us a list of items to create pre-wired to the input of our Timer.

- Left click *Constant*. The windows vanish and a blue box containing the number zero will be created to the left of the Timer. This is a Numeric Constant, and it should be wired to the input of your Timer (if it isn't, wire it now. Refer to *Chapter 1: Wire Connections* for wiring assistance.)

- Left click once on the number '0" and replace it with the number "100". The second frame of your Flat Sequence Structure should now match the second frame as depicted in the image presented at the start of Chapter 3, Section 2.

Note
Don't bother creating the second Timer at this point. Near the end of section 3.2 you will be asked to create the Timer based on the skills you've acquired..

3.2.5 Creating a Call Library Function Node

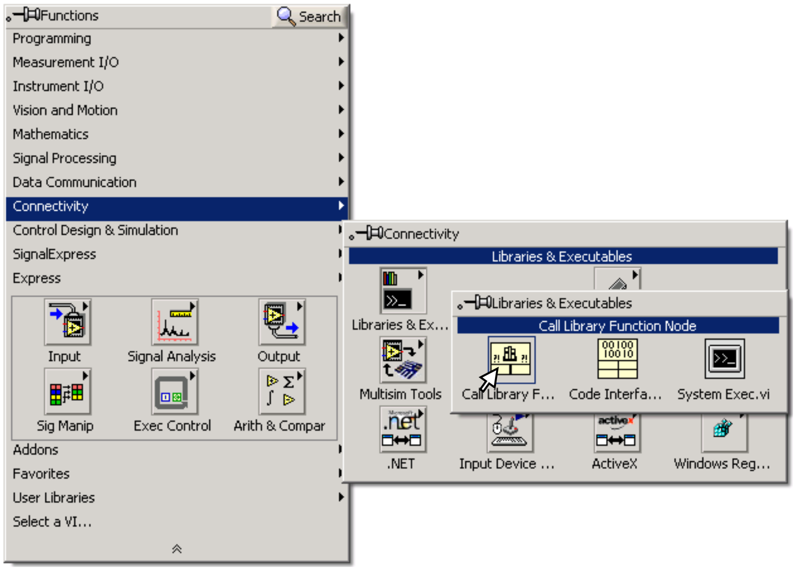
- Right click an empty area within your Block Diagram to access the Functions menu.

- Once again, left click the bottom item (a double down arrow) to expand the list of function categories.

- Hover your cursor over the *Connectivity* category. A new window opens.

- On the *Connectivity* menu, hover your mouse over the *Libraries and Executables* icon.

- Left click the *Call Library Function Node*. The windows vanish and you are now holding a footprint.



- Place the Call Library Function Node within the First frame of the Flat Sequence Structure.

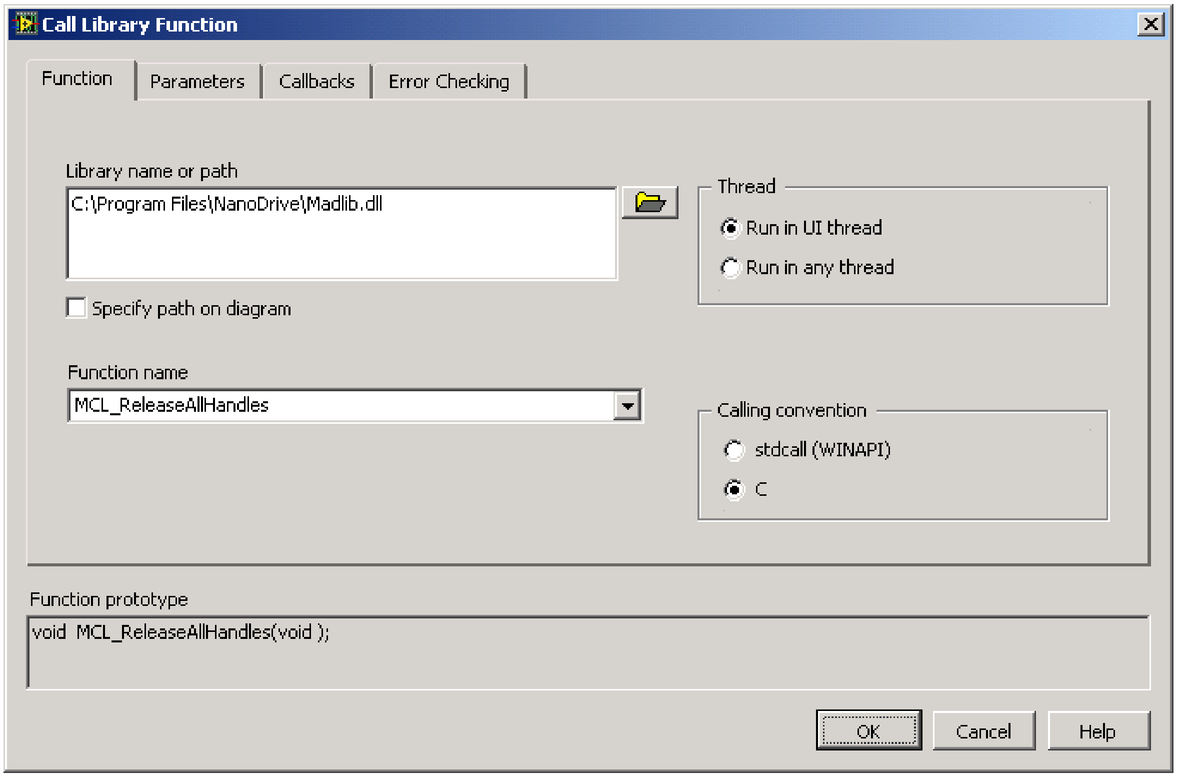
Call Library Function Node: Configuring to release all Handles.
Recall that to communicate with a Nano-Drive® we must first identify the Nano-Drive®by its Handle, and before we can identify the Nano-Drive® by its Handle we must first tell Windows to acquire a Handle for the device. You are now configuring a MADlib.dll function which will complete the cycle by telling windows to release the Handle, allowing the VI to clean up after itself before ending. Failure to release a Handle may cause problems with later VI's that attempt to communicate with the same device.

- Right click the newly created component, and select *Configure* from the menu that appears (or double-left click the component). Within the next few steps, we will set your Call Library Function Node to match the one seen below.

- Left click the *browse* icon.

- Locate the file "Madlib.dll". The file was automatically installed onto your hard drive when you first installed the Nano-Drive® software. The default location for this file is *C:\Program Files\NanoDrive\Madlib.dll*

- With the Madlib.dll selected as the Library to access, the *Function name* list has been automatically populated with all the supplied MCL functions. Left click the arrow at the right of the list to display all available functions.



Call Library Function

Function | Parameters | Callbacks | Error Checking

Library name or path: C:\Program Files\NanoDrive\Madlib.dll

☐ Specify path on diagram

Function name: MCL_ReleaseAllHandles

Thread
● Run in UI thread
○ Run in any thread

Calling convention
○ stdcall (WINAPI)
● C

Function prototype: void MCL_ReleaseAllHandles(void);

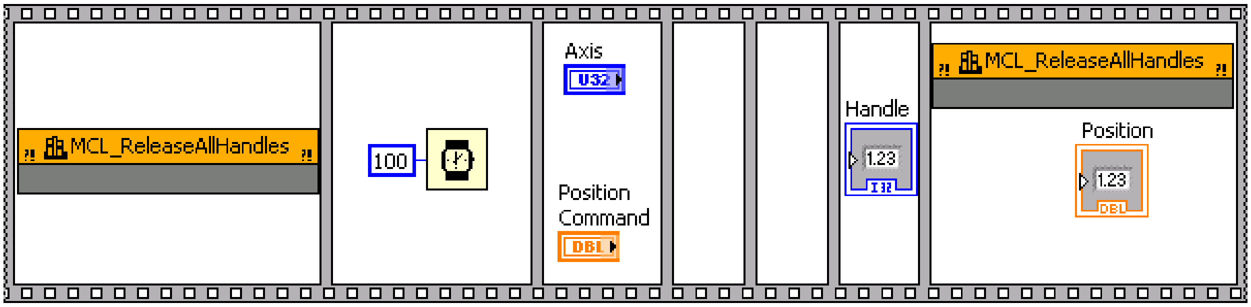
- Select *MCL_ReleaseAllHandles* from the list. The *Function* tab of your Call Library Function Node should now exactly match the image above.

- The function, *MCL*_ReleaseAllHandles, doesn't require that we add or edit any parameters. All other tabs should be left at their default settings. Left click *OK* to accept the changes we've made.

- A dialog box will appear to inform you that the parameters will be automatically set to void. Click *OK*. This Call Library Function Node is complete.

- Either repeat the steps of Section 2.3, or copy and paste this Call Library Function Node, to create a second call for the function MCL_ReleaseAllHandles in the last (7th) frame of the Flat Sequence Structure.

- Verify that your Block Diagram closely resembles the one depicted below.

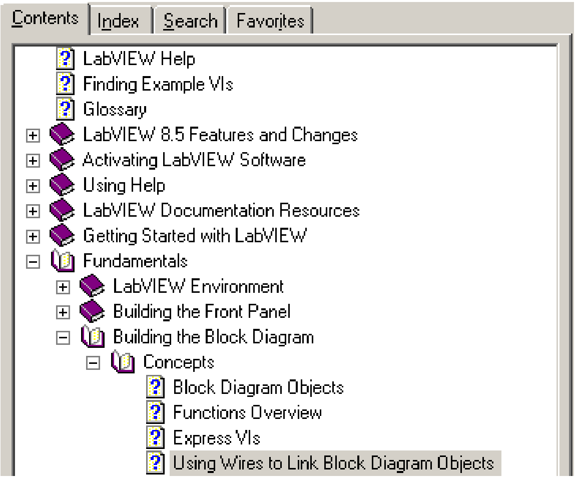


Note
Your Call Library Function Nodes may be set to a more compact display type than seen above. To display the name of the function being called, right click the CallLibrary Function Node and hover you mouse over the *Name Format* category. Here you   can toggle between *Names*, or *No names*.

3.2.6 Acquiring a Handle

You will now create the Call Library Function Node which calls an MCL function that will acquire a *Handle* for your Nano-Drive®. By the end of this section you will have a functioning VI that will, among other things, grab a Handle for your Nano-Drive® and display it on the Front Panel via the "Handle" Numeric Indicator.

Accessing the LabVIEW Help Document on Wire Connections
You will soon be creating wires to connect the components you've created. This would be an excellent time to review the LabVIEW Help documentation regarding wire connections. LabVIEW Help documents can be accessed via the *Help>>Search the LabVIEW Help...* menu item. The image below demonstrates locating a document by navigating the *Contents* tab of the LabVIEW Help viewer.



- LabVIEW Help
- Finding Example VIs
- Glossary
- LabVIEW 8.5 Features and Changes
- Activating LabVIEW Software
- Using Help
- LabVIEW Documentation Resources
- Getting Started with LabVIEW
- Fundamentals
  - LabVIEW Environment
  - Building the Front Panel
  - Building the Block Diagram
    - Concepts
      - Block Diagram Objects
      - Functions Overview
      - Express VIs
      - Using Wires to Link Block Diagram Objects

- Create another Call Library Function Node either by repeating the steps described above, or by copying and pasting.

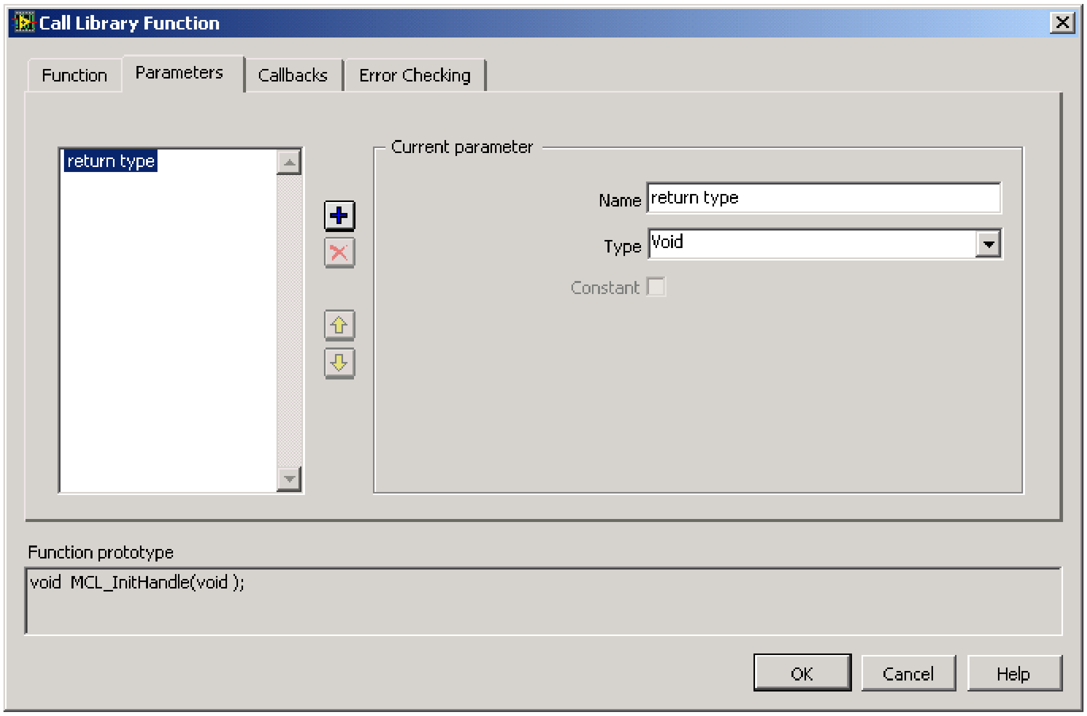
- Place it into the third frame of the Flat Sequence Structure.

- Enter its *Configure* menu.

- Direct the *Library name or path* to the Madlib.dll using the browse button, as done previously.

- Under *Function name*, select *MCL_InitHandle*.

- Left click the *Parameters* tab at the top of the Configure menu.



return type

Name: return type

Type: Void

Constant ☐

Function prototype

void MCL_InitHandle(void);

- In the *Current parameter* area, replace the name “return type” with the name “handle”.

- Change *Type* from “void” to “Numeric”. Two new settings should appear; *Constant*, and *Data Type*.

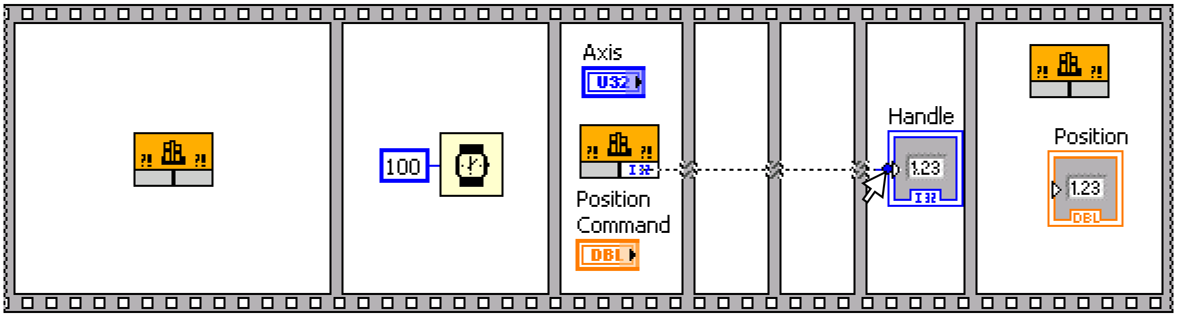
- Change Data Type to “Signed 32-bit Integer”.

- Left click *OK* to accept the changes.

Note
The appearance of this Call Library Function Node should have slightly changed. If the *Names* format is checked, the *handle* parameter now protrudes from the bottom of the Call Library Function Node. With the *No Names* format checked, an I32 has been

added to the output (right) side of the Call Library Function Node (hover your cursor over the blue "I32" output and LabVIEW will show you the name.)

- From the Block Diagram, left click the blue I32, or *Handle* output, on the MCL_InitHandle Call Library Function Node. You are now dragging a wire out of this output.



- Left click the input of the "Handle" Numeric Indicator. A wire has been automatically created from the output of the MCL_InitHandle function to the input of the Numeric Indicator.

Wire Connections and Tunneling
You'll notice LabVIEW has a special way of bridging Flat Sequence Structure frames; a blue box has been created at every junction. The blue box is called a *tunnel*; and it allows data to flow in, out, and between, frames. To learn more about this, read the LabVIEW Help file on Flat Sequence Structures, or read the enclosed glossary definition for *tunnels*.

3.2.7 Running the VI

- Turn on the Nano-Drive®™.

- Switch back to the Front Panel (*Ctrl* + *E*.)

- In the top left corner of the Front Panel, there should be an unbroken *Run* icon. If your *Run* icon appears broken , left click it to receive a list of errors in your program. LabVIEW will not allow the VI to run until all errors have been eliminated.

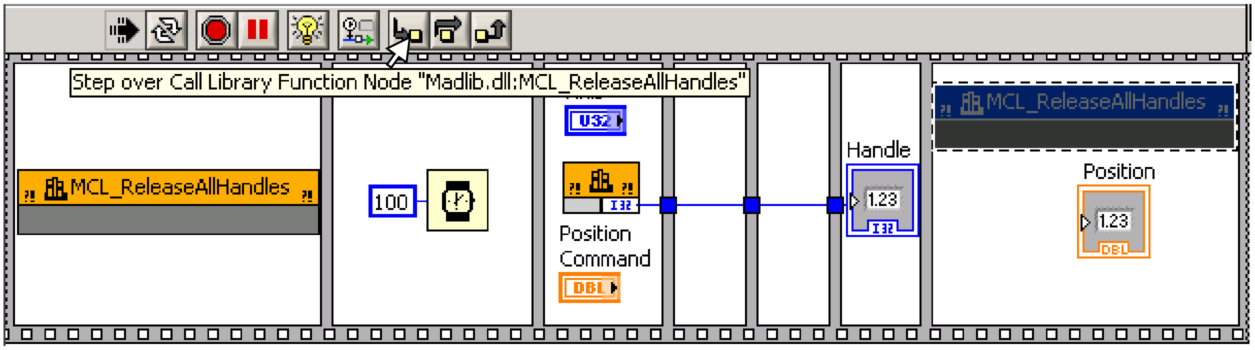
- Run the VI by left clicking the *Run* button. The VI runs for 100 milliseconds then. A number should now be present within the text field of the "Handle" Numeric Indicator.

- Switch to the Block Diagram of your VI. The Block Diagram offers a few additional ways to run the VI. These can be seen to the right of the *Run* icon.

- Left click the *Highlight Execution* icon . The bulb within the icon should light representing that your VI will now run in Highlight Execution mode.

- From the Block Diagram, and with Highlight Execution on, left click the Run icon to run the VI. LabVIEW will now slowly step through every phase of your program and present to you a visual representation of what is being done.

- Try *single-stepping* through the entire VI. Single-stepping puts you in complete control of the speed of execution, the VI only advances for each click of the single-step button. When you want to stop, left click the red stop button.



Step over Call Library Function Node "Madlib.dll:MCL_ReleaseAllHandles"

- Make sure your VI is stopped and Highlight Execution is turned off before continuing. If your VI is running the Run icon is a solid black arrow ➡ . Press the Stop icon to stop execution ● . If Highlight Execution is on, its toolbar icon will be a lit light bulb. Left click the icon to shut off Highlight Execution.

3.2.8 *Writing to the Nano-Drive® DAC*

- Create a Call Library Function Node.

- Place it within the fourth frame of the Flat Sequence Structure.

- Enter its Configure menu.

- Set the *Library name or path* to the location of the Madlib.dll.

- Set the *Function name* to “MCL_SingleWriteN.”

- Switch to the Parameters tab.

- Change the name of the “return type” parameter to “Error code.”

- Change the T*ype* of the Error Code parameter to “Numeric.”

- Leave the *Data type* of the Error Code parameter as “Signed 32-bit Integer.”

- Left click the *Add a parameter* ⊞ button three times.

- Check the *Function* p*rototype* of your Call Library Function Node. It will be located near the bottom of the Parameters tab of the Configure menu. It should read:

```
int32_t MCL_SingleWriteN(int32_t arg1, int32_t arg2, int32_t arg3);
```

Interpreting the Function Prototype
The function MCL_SingleWriteN is now configured to receive three arguments (arg1, arg2, and arg3), all of type “int32_t”, and will return an Error Code as a Signed 32-bit Integer. Compare this to the actual prototype for the function MCL_SingleWriteN as it appears within the Appendix A or Madlib_*_*.doc. You will see that we are not finished configuring the function parameters…

- Change the name of “arg1” to “position”.

- Leave the *Type* as Numeric, but change the *Data type* to 8-byte Double.

- Leave the *Pass* as Value.

- Change the name of “arg2” to “axis”.

- Leave the *Type* as Numeric, but change the *Data type* to “Unsigned 32-bit Integer”.

- Leave the *Pass* as Value.

- Change the name of “arg3” to “handle”.

- Leave the *Type* as Numeric, leave the *Data type* as Signed 32-bit Integer, and leave the *Pass* as Value.

- Once again, check the Function Prototype. It should reflect the changes we’ve made. Compare this to the actual prototype of the MCL_SingeWriteN function as it appears within the Madlib_*_*.doc.

```
int  MCL_SingleWriteN(double position, unsigned int axis, int handle)
```

(The MCL_SingleWriteN function as it appears within the Madlib_*_*.doc)

```
int32_t MCL_SingleWriteN(double position, uint32_t axis, int32_t handle)
```

(Your MCL_SingleWriteN function as it appears in the Function prototype area of the Configure menu)

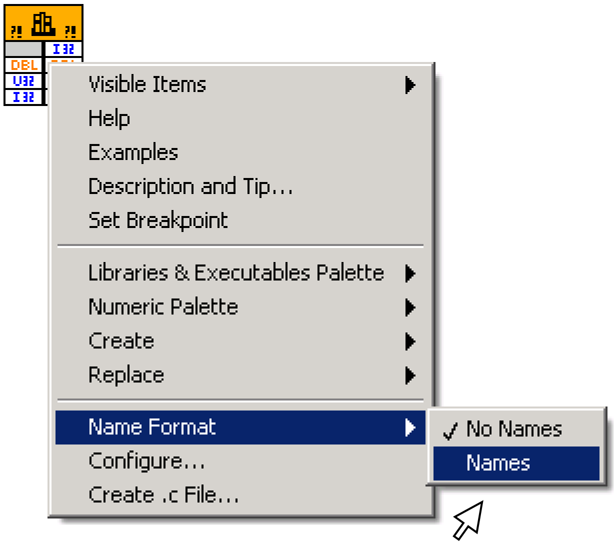
Interpreting the Function Prototype; Part 2
The declaration *uint32_t* is equivalent to declaring the variable as *unsigned int*. Similarly, the declaration *int32_t* is equivalent to declaring the variable simply as *int*. Your Call Library Function Node is now properly configured to call, and handle, the function MCL_SingleWriteN.

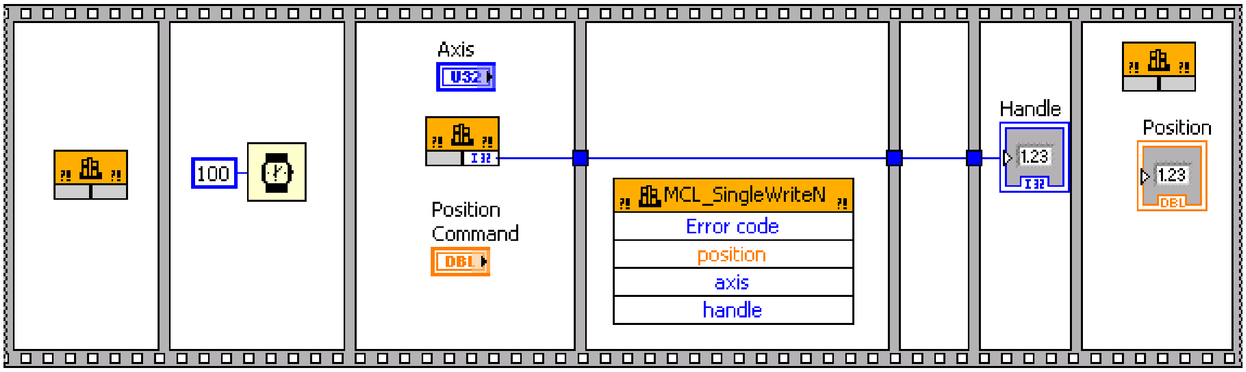
All parameters must appear in the exact order depicted above. If necessary, use the up ⇧ and down ⇩ arrow buttons to rearrange the order of the parameters.

- Left click *Ok* to accept the changes. The Call Library Function Node has become taller, compensating for the added parameters.

- Right click the Call Library Function Node, and hover your cursor over the *Name Format* category.

- Left click *Names*. It should now be clear to see the parameters of this Call Library Function Node. If you don't prefer this icon type, simply toggle back to *No Names*.





Note
Avoid congested Block Diagrams. Don't allow components to overlap wires, and make sure wire connections are clearly depicted as entering the proper input. Expand the frames of the Flat Sequence Structure if you need more room to work.

Interpreting the Function Prototype; Part 3

Unlike the Call Library Function Nodes for the MCL_ReleaseAllHandles() function, the MCL_SingleWriteN() Call Library Function Node requires you to input data. Also, the function "returns" (outputs) an error code (or returns "0" if no error occurs.) Did you see an Error Code parameter in the Function Prototype? The Data Type Representation of a return (output) parameter of a function always begins the function, i.e. the *int 32_t* in *int   32_t MCLSingleWriteN(...)*. The name we designated, "Error code", is a superficial LabVIEW conventionality and so does not appear in the actual function.

3.2.9 Reading the Nano-Drive® ADC

- Create a Call Library Function Node.
- Place it within the sixth frame of the Flat Sequence Structure.
- Enter its Configure menu.
- Set the *Library name or path* to Madlib.dll.
- Set the *Function name* to “MCL_SingleReadN.”
- Switch to the Parameters tab.
- Change the name of the “return type” parameter to “position”
- Change the T*ype* of the position parameter to “Numeric.”
- Change the *Data type* of the position parameter to “8-byte Double.”
- Left click the *Add a parameter* ⊞ button two times.
- Change the name of “arg1” to “axis.”
- Leave the *Type* as “Numeric”, but change the *Data type* to “Unsigned 32-bit Integer.”
- Leave the *Pass* as “Value.”
- Change the name of “arg2” to “handle.”
- Leave the *Type* as “Numeric”, and leave the *Data type* as “Signed 32-bit Integer.”
- Leave the *Pass* as “Value.”

  Your function prototype should now appear as:

  ```
  double MCL_SingleReadN(uint32_t axis, int32_t handle);
  ```

- Rearrange the order of the parameters if necessary to match the prototype displayed above. Remember to use the up ⇧ and down ⇩ arrow buttons to accomplish this.
- Left Click *Ok* to accept the changes.

3.2.10 Make the wire connections, and create the second Timer

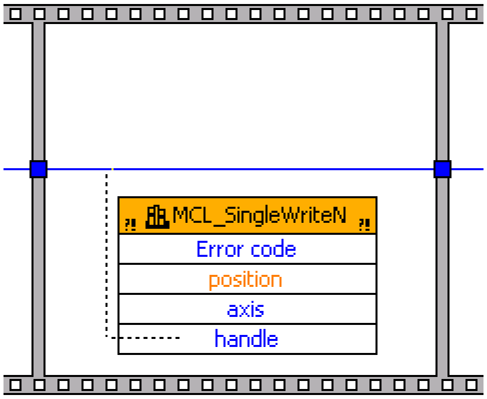
It is now time to create the remaining wire connections. The following instructions include every step necessary to accomplish this. However, if a wire is accidentally `placed, or overlaps other components, refer to *Chapter 1: Wire Connections* for supplementary assistance.

- Configure the MCL_SingleWriteN, and MCL_SingleReadN, Call Library Function Nodes to display Names (right click the component and select *Names* in the *Name Format* category.)

- Expand the frames of your Flat Sequence Structure so that no component is within an inch (25mm) of an edge. This will give you enough room to make the wire connections.

Note
Make Coherent Wire Connections. After connecting a wire, left click to select a section of it. Vertically traveling wires can be moved horizontally, horizontal wires can be moved vertically. Wires can be selected and moved one pixel at a time with the arrow keys. Avoid overlapping components with your wires whenever possible.

- Left click the left side of the *handle* parameter on the MCL_SingleWriteN Call Library Function Node. You are now drawing a wire, connected at one end to the *Handle* input of the Single Write function.

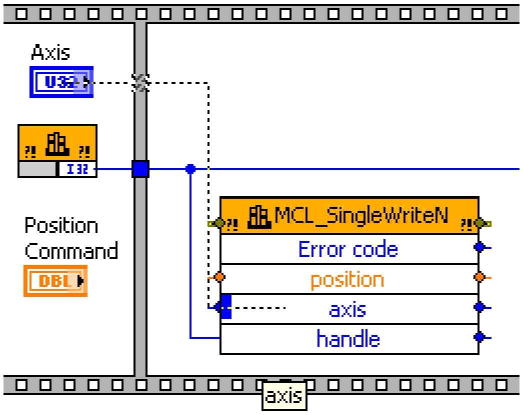


- Left click on the existing wire that passes through this frame. The wire passing through this frame comes from the MCL_InitHandle function, and is being passed to a Numeric Indicator. The MCL_SingleWriteN function now receives the same Handle as the Numeric Indicator.

- Do the same for the MCL_SingleReadN Call Library Function Node. Make your wire connections within the frame which contains this Call Library Function Node.

- Connect the output (right) side of the “Axis” Numeric Control to the *axis* input of the MCL_SingleWriteN Call Library Function Node.

- Do the same for MCL_SingleReadN. Once again, make your wire connections within the frame containing MCL_SingleReadN.



- Connect the output of the “Position Command” Numeric Control to the *position* input of MCL_SingleWriteN.

- Connect the *position* output of MCL_SingleReadN to the input (left) side of the “Position” Numeric Indicator (located within the seventh frame.)

- Create another Timer that will halt execution for 100 milliseconds. Rather than copying and pasting the Timer into existence, try recalling how to create it via the functions palette. Place this Timer within the fifth frame of the Flat Sequence Structure (don’t forget to set the number of milliseconds to wait.)

Section 3.3 – Perform a Write/Read Operation

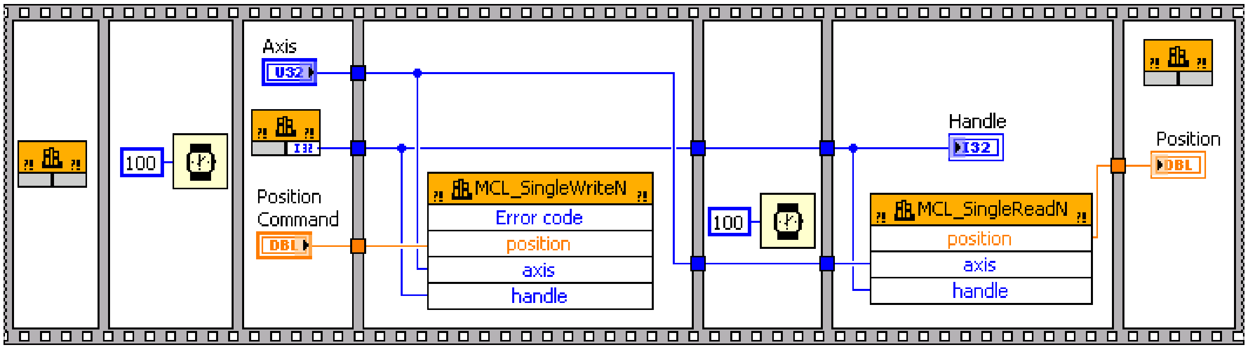
The Block Diagram is complete. Take a look at the *Run* arrow; it should be the normal image of an unbroken, white, arrow. However, if the Run arrow appears broken; the Block Diagram of this VI contains errors.

Troubleshooting a VI

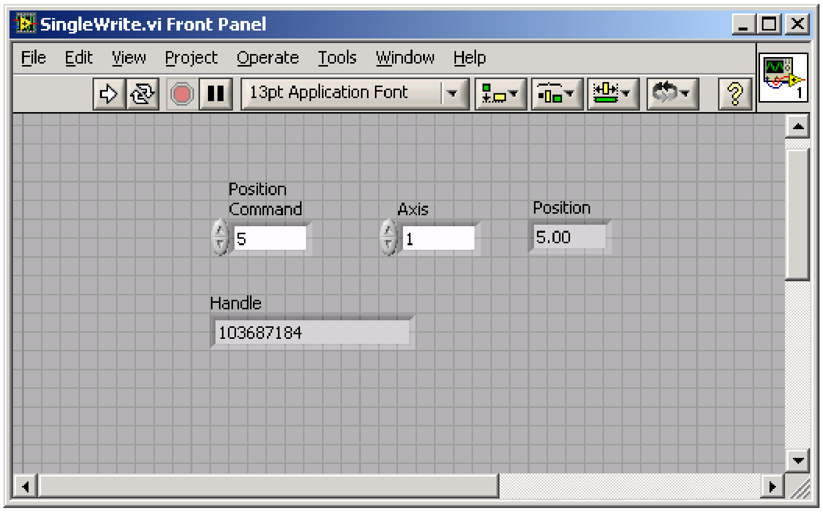
LabVIEW will not allow a VI to run while errors exist on the Block Diagram. Examples of such errors include an open/unconnected input or multiple outputs are wired together. Left click the broken Run arrow (seen at left) and LabVIEW will help you find the errors. If there is an open input, take a look at the image below to assist you in finding the correct connection. Also, this would be a good time to verify that all Indicators, Controls, and Function Parameters, are set to the correct Data Types.

Note

Connecting together components of different Data Type Representations will not generate an error. Instead, LabVIEW automatically converts the data which is passed. Therefore, it is entirely up to you to verify that your Data Type Representations are correct.



(Image of the Finished Block Diagram)



Position Command: 5

Axis: 1

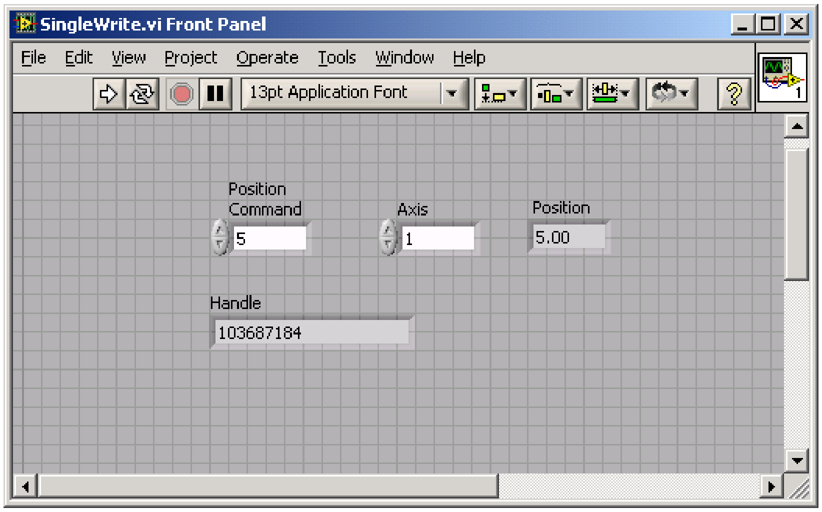
Position: 5.00

Handle: 103687184

(Image of the Completed VI's Front Panel after a single Run.)

3.3.1 Enter Valid Command Parameters

- Turn on the Nano-Drive®.
- Switch to the Front Panel of your VI.
- Select an axis to move (X = 1, Y = 2, Z = 3).
- Enter a position to move to into the “Move to (um)” control. Choose a value within the range 0-50.
- Run the VI. The “Handle” indicator fills with a large number, and the “Position” indicator displays a number from 0-50 according to the position read from the specified axis 100 milliseconds following the Write command.


SingleWrite.vi Front Panel
File Edit View Project Operate Tools Window Help
13pt Application Font

Position Command: 5
Axis: 1
Position: 5.00
Handle: 103687184

(Image of the Completed VI's Front Panel after a single Run.)

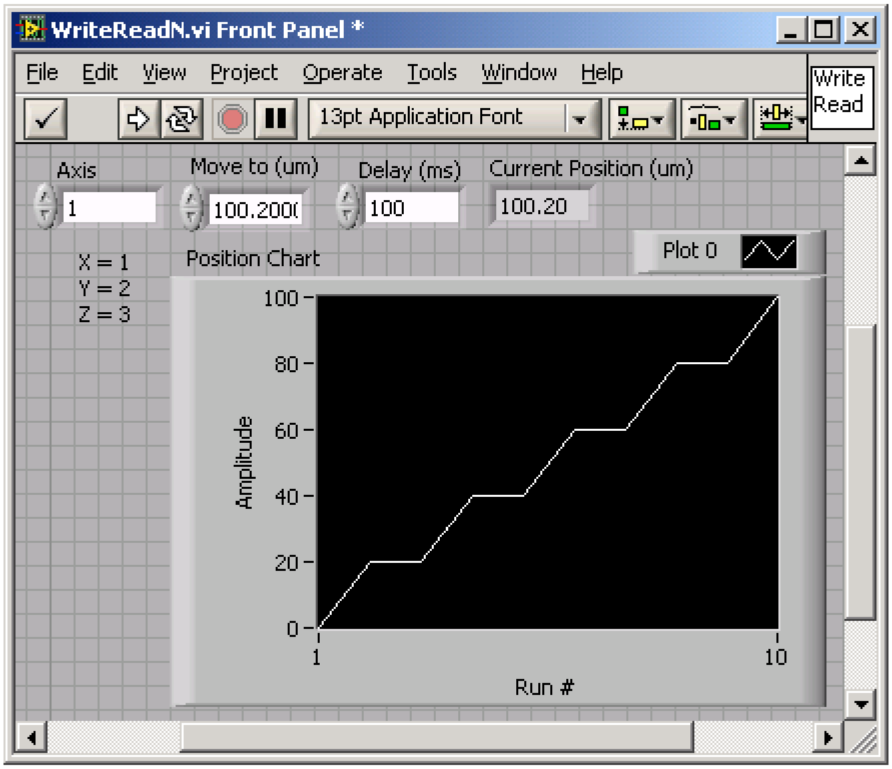
Project Completed

Completion of this Chapter has resulted in a VI that will write a position command to the Nano-Drive®and read the position of the Nano-Drive® 100 milliseconds after the command. A VI built to control one or more Nano-Drive® controllers will often include these fundamental elements. The following Chapters will build upon the skills acquired in this Chapter.

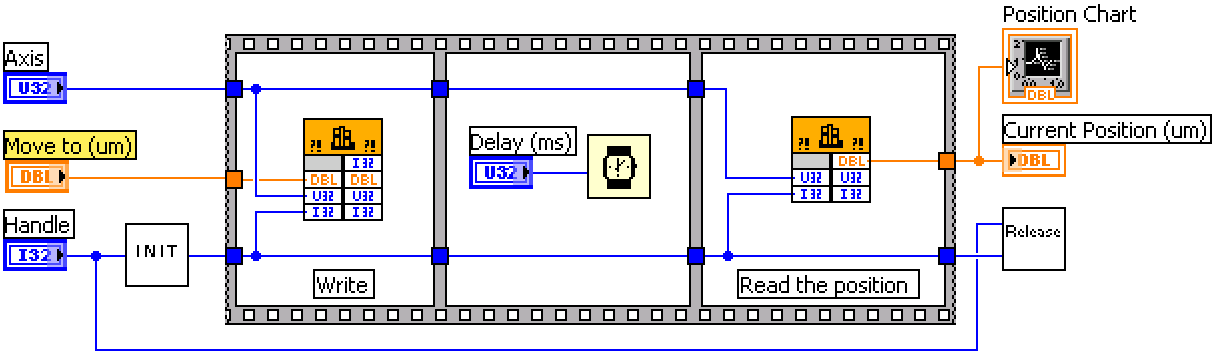
Note

From this point on each chapter will begin with a declaration of all MADlib.dll functions to be used within the chapter. An indication will also be present when the function is making its first appearance in the tutorial.

The VI shown below will be created in Chapter 4.



(Front Panel of the finished Chapter 4 VI after a series of executions)



(Block Diagram of the Finished Chapter 4 VI)

Goal of Chapter 4

During this chapter, you will be instructed through the process of creating three distinct VIs, each of which will handle a separate portion of the same program. A few new LabVIEW concepts will be introduced along with several new LabVIEW components. The finished VI will accomplish a similar task as the VI created in Chapter 3; however, the emphasis here will be the use of SubVIs as a programming component.

Topics Include:

1. The use of SubVIs to simplify Block Diagram programming.
2. Editing the Connector Pane of the VI.
3. Responding to data using Case Structures and Comparison Functions.

MADlib.dll Functions Used Within This Chapter Include:

1. void MCL_ReleaseHandle(int handle);
2. int MCL_InitHandle();
3. int MCL_SingleWriteN(double nposition, unsigned int axis, int handle);
4. double MCL_SingleReadN(unsigned int axis, int handle);

SubVIs Used Within This Chapters Project Include:

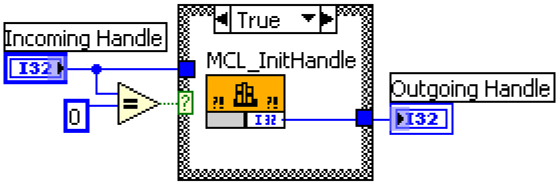
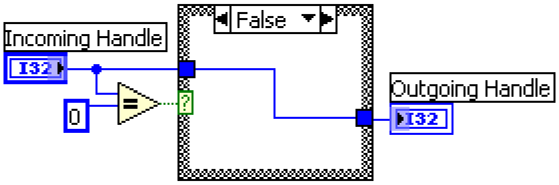
1. StandardInit.vi / StandardInitSubVI.vi *(new)*
2. StandardRelease.vi / StandardReleaseSubVI.vi *(new)*

Section 4.1 – Creating the Init SubVI

The two images below depict two different states of the same Block Diagram. This is the Block Diagram you will create in this section of the tutorial. It isn’t necessary to see an image of the finished Front Panel; the Front Panel simply contains the ‘Incoming Handle” Numeric Control and “Outgoing Handle” Numeric Indicator.

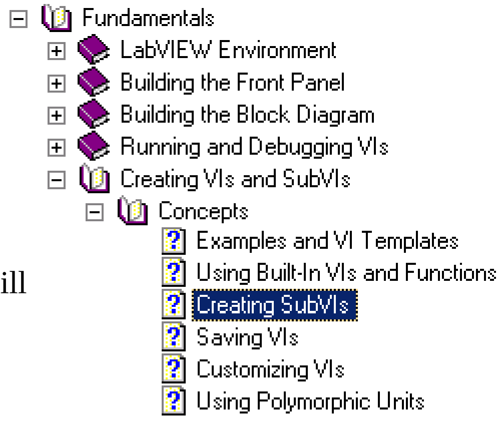
Creating a SubVI to initialize the handle

As you may have assumed from the label “Incoming Handle” of the Numeric Control (see below,) this Control will receive an input from outside the VI. Similarly, the “Outgoing Handle” Indicator will pass data outside of the VI for use in other VIs. With this concept in mind, try interpreting what will happen when the VI is run. The function of this VI will be explained in depth as you progress through this section.

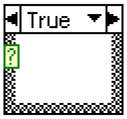
Incoming Handle **is equal** to zero:	Incoming Handle is **not equal** to zero:
 Incoming Handle I32 0 True MCL_InitHandle I32 Outgoing Handle I32	 Incoming Handle I32 0 False Outgoing Handle I32

Recommended Reading: SubVI
This VI will act as a SubVI; this means that you will design it to work as a component within other VIs which you create. The SubVI is an important concept; it is therefore recommended that you read the LabVIEW Help document on SubVIs.

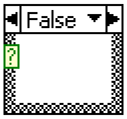
Accessing LabVIEW Help documents on SubVIs
To access the LabVIEW Help document on Creating SubVIs: Left click the *Help* Menu item at the top of your LabVIEW window; select *Search the LabVIEW help…* from the window that appears; (the LabVIEWHelp window should open) select the *Contents* tab from the left-most menu; expand the *Fundamentals* folder, expand the *Creating VIs and SubVIs* folder, expand the *Concepts* folder. Select *Creating SubVIs* from the list. The document will display in a window to the right.



The Case Structure (Block Diagram)



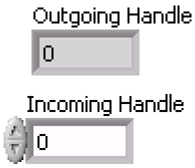
The *Case Structure* (seen at left) resembles a Flat Sequence Structure in both appearance and operation. However, the frames of the Case Structure are inseparably stacked one on top of another. Consequently, only one at a time is visible on the Block Diagram at a time.



The key to the Case Structure is that only one of its frames (referred to as *cases*) will execute, the rest remain inactive for the duration of the containing VI/loop. Which case (frame) runs depends entirely on the value passed to the *Selector Terminal* ? (which appears on the left side of the structure). The Case Structure therefore allows the programmer to answer a question asked on the Block Diagram

4.1.1 Create the Numeric Control and Indicator.

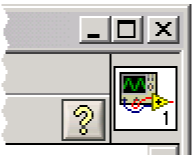
- Begin LabVIEW and start a new VI by selecting *Blank VI* from the *Getting Started* Window. The Front panel and Block Diagram of a new VI appear.
- On the Front Panel, create a Numeric Control.
- Change its data type representation to I32 (signed 32-bit integer.)
- Change its label to “Incoming Handle.”
- Create a Numeric Indicator.
- Change its data type representation to I32.



- Change its label to “Outgoing Handle.”

4.1.2 Create the Terminals of the VI

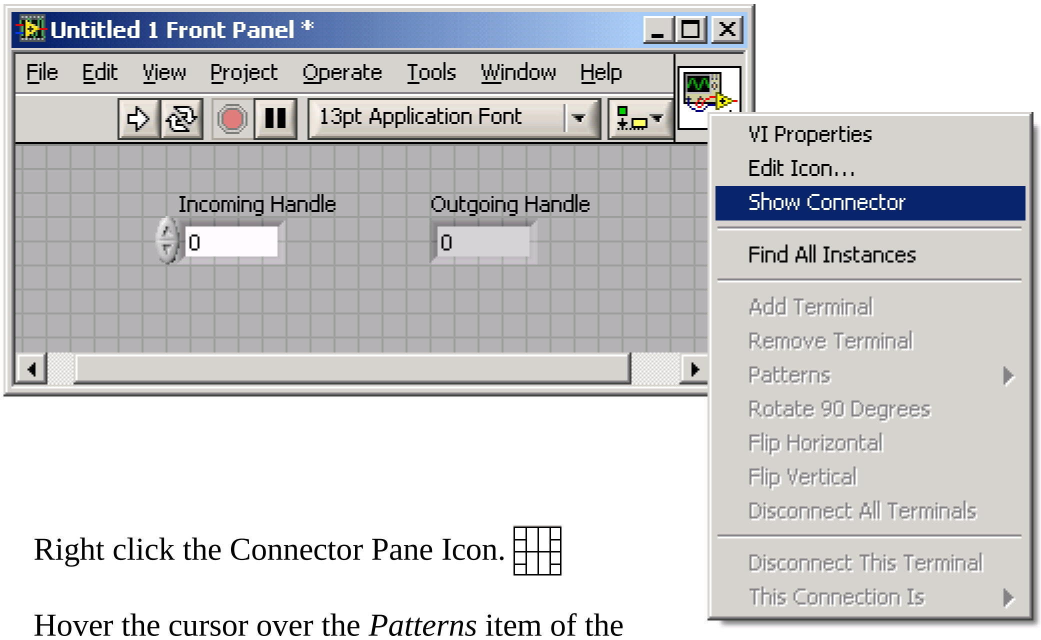
Every time you create a new VI, LabVIEW automatically prepares it to be used as a SubVI. Therefore, your VI has already been assigned a Block Diagram icon and a set of input/output terminals. So far these input and output terminals do not connect to anything within your VI.



- Right Click the icon that appears in the top-right corner of your Front Panel. This is the default icon for a new VI.

- Select *Show Connector* from the menu that appears (see image below.) The Icon should switch from the graphic of an oscilloscope to a group of boxes. This is the Connector Pane of your VI.

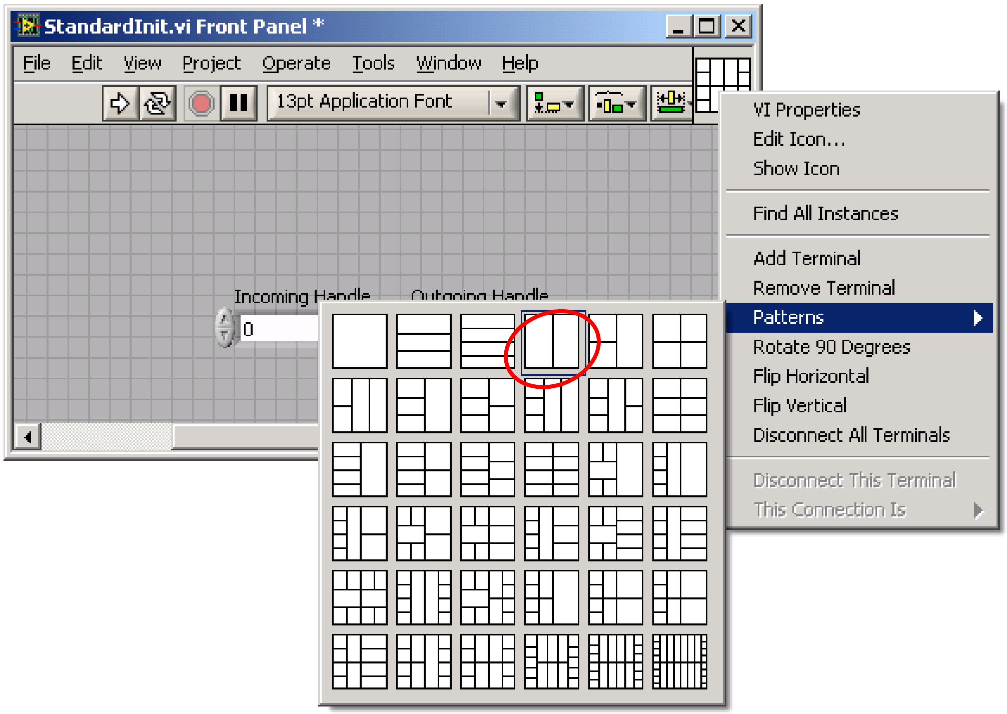
Note
You cannot Show Connectors from the Block Diagram. You will not have access to the options shown in the image below unless you are on the Front Panel.



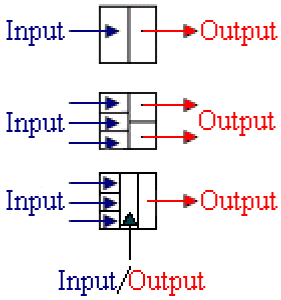
- Right click the Connector Pane Icon.

- Hover the cursor over the *Patterns* item of the menu that appears (seen below.)

- Select the fourth pattern from the left on the top row. In the image below, a red circle has been added to indicate which pattern to choose.



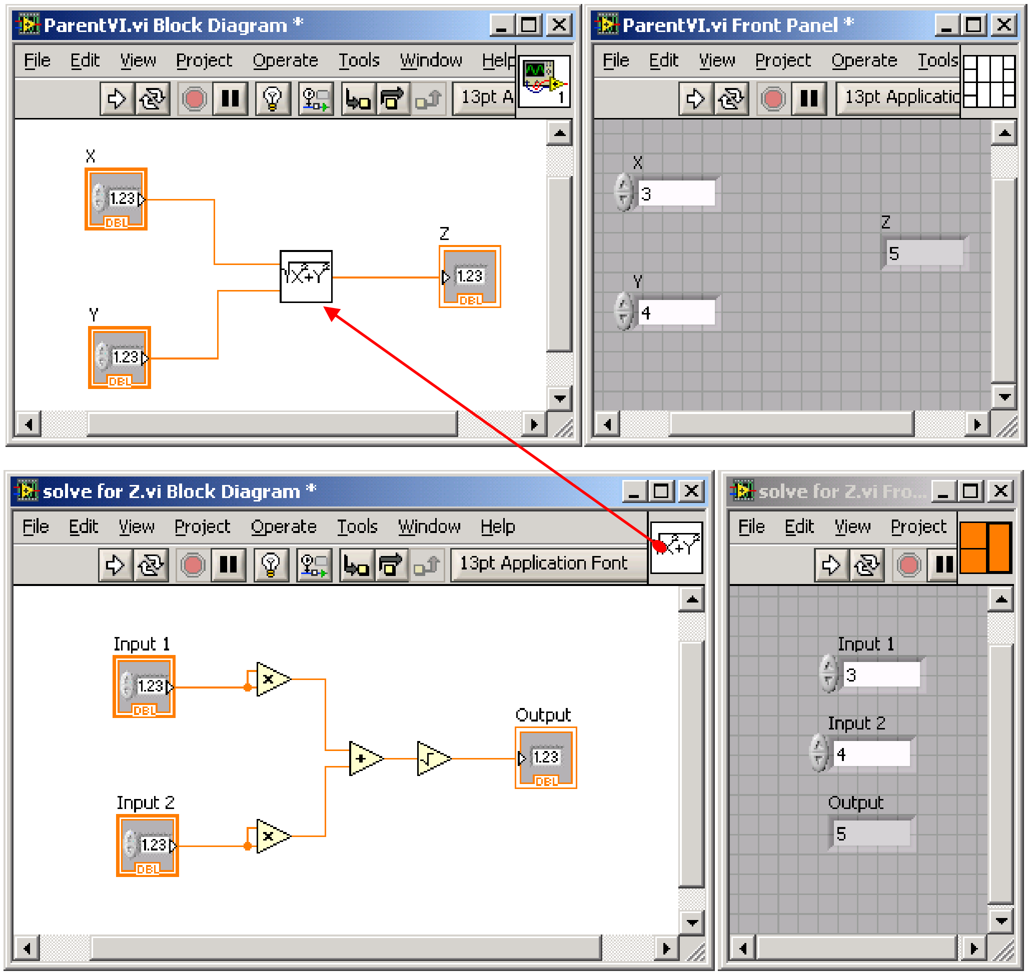
The Connector Pane



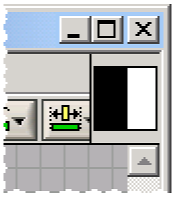
The Connector Pane specifies the input and output terminals of the VI you're creating. If this concept sounds obscure, keep in mind that some VIs are meant to be used within other VIs as SubVIs. Using the Connector Pane is actually quite simple and intuitive, but requires an understanding of the underlying concept of the SubVI. If you haven't already read the National Instruments documentation on SubVIs, please refer to the beginning of this chapter for assistance locating these documents.

Example Input and Output on the Connector Pane

Below are images of two separate VIs; *ParentVI.vi*, and *solve for Z.vi*. Both VIs are configured to accomplish the same task; compute the length of the vector, Z. However, as can be seen, *ParentVi.vi* does none of the math. Instead, it passes the value of its Numeric Controls (X and Y) to *solve for Z.vi*. *solve for Z.vi* passes the data through the appropriate mathematical operators, then stores the computed value inside its Numeric Indicator (Output). The Output Indicator is connected to the output terminal of the *solve for Z.vi* Connector Pane, which allows its value to pass out of *solve for Z.vi*. *ParentVI.vi* then retrieves the value of the Output control and passes the value to its Numeric Indicator (Z.)


ParentVI.vi Block Diagram *
File Edit View Project Operate Tools Window Help
13pt A
X
Y
Z
1.23
DBL
ParentVI.vi Front Panel *
File Edit View Project Operate Tools
13pt Applicatio
X
3
Y
4
Z
5
solve for Z.vi Block Diagram *
File Edit View Project Operate Tools Window Help
13pt Application Font
Input 1
Input 2
Output
solve for Z.vi Fro...
File Edit View Project
Input 1
3
Input 2
4
Output
5

4.1.3 Set the input and output of the VI



- Left click the input (left) half of your VI’s Connector Pane. The side you click should turn black (as seen to the right.)

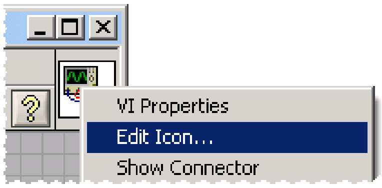
- Left click the Incoming Handle Control. The left half of your Connector Pane should turn blue (if it changes from black to a color other than blue, verify the Incoming Handle control is set to I32.)

- Left Click the output (right) half of your VI’s Connector Pane.

- Left Click the Outgoing Handle Control. The right half of your Connector Pane should turn blue.

4.1.4 Edit the icon

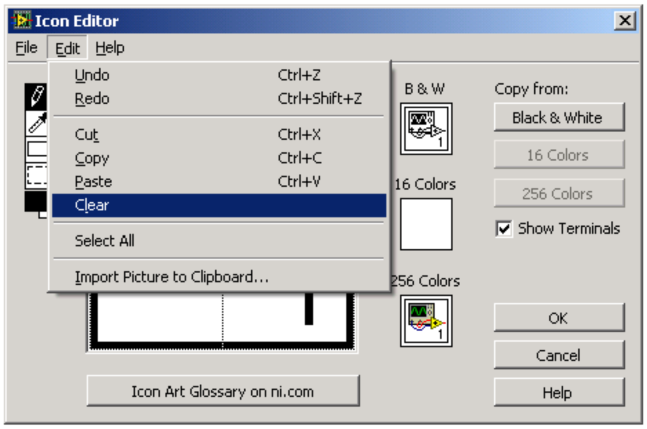
This SubVI will appear within the Block Diagram of future VI's as an icon with input/output terminals. The icon of a SubVI should summarize its purpose; the purpose of this SubVI is to *initialize* a handle for use within other VI's.



- Right click the Connector Pane icon.

- Select *Show Icon* from the menu that appears. The Connector Pane graphic should be replaced with the more familiar graphic of an instrument displaying a sine wave.

- Once again, right click the icon.



- Select *Edit Icon* from the menu that appears. The *Icon Editor* opens (as seen to the right.)

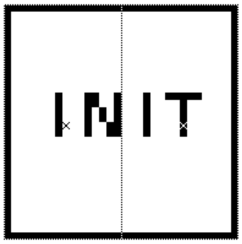
- Clear the image to white. This can be done either by painting white over the entire image, or by selecting *Clear* from the *Edit* menu.

- Select the A text tool from the toolbar on the left side of the Icon Editor.

- Left click slightly left of the center in your icon. You can now apply text to this area.

- Enter: I N I T

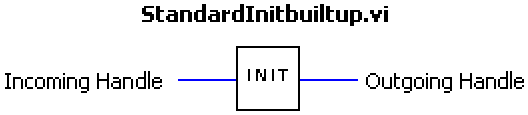
- Draw a black border around the perimeter of the canvas using the pencil, line or unfilled square tool. Do not draw the border tightly around the text; it should encompass the entire drawing area.



- Check the *Show Terminals* option on the right side of the Icon Editor. Your image should appear very similar to the one seen at right.

- Click the *OK* button at the bottom-right of the Icon Editor.

Note
To view the inputs/outputs of a SubVI in a format similar to the image below, select *Help>>Show Context Help*, then left click the icon of the SubVI.



4.1.5 Configure a Call Library Function Node to Call MCL_InitHandleOrGetExisting

The Front Panel is finished. Switch to the Block Diagram (hold *Ctrl* and press *E*.)

- From the Block Diagram, create a Call Library Function Node. Place the Call Library Function Node anywhere on the Block Diagram.

- Enter its *Configure* menu.

- Direct the *Library name or path* to the Madlib.dll using the browse button.

- Under *Function name*, select *MCL_InitHandleOrGetExisting*.

- Left click the *Parameters* tab at the top of the Configure menu.

- In the *Current parameter* area, replace the name “return type” with the name “handle”.

- Change *Type* from “void” to “Numeric”. Two new settings should appear; *Constant*, and *Data Type*.

- Change Data Type to “Signed 32-bit Integer”.

- Left click *OK* to accept the changes.

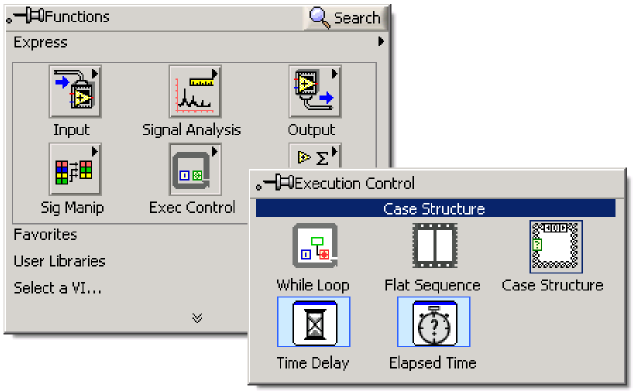
4.1.6 Create the Case Structure

You will now be instructed to create a Case Structure. The Case Structure creation process will closely resemble creation of the Flat Sequence Structure.

- Right click over any empty area of the Block Diagram. The *Functions* menu appears.

- Hover your cursor over the *Exec Control* icon which appears within the *Express* submenu of the Functions menu.

- Left click the *Case Structure* icon. The two menus vanish.



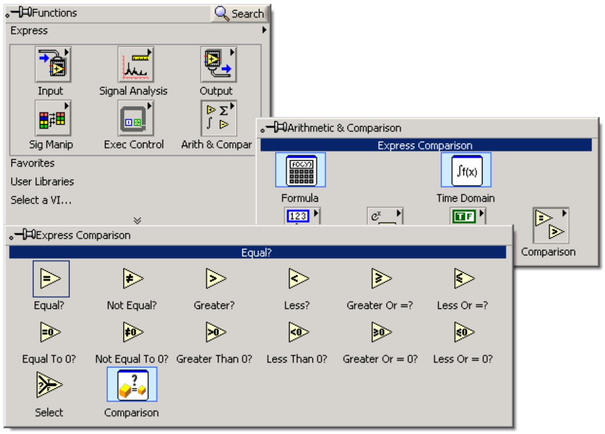
- Left click any empty area of the Block Diagram.
- Move your cursor approximately three inches (75mm) diagonally. Be careful to avoid overlapping any other Block Diagram components.
- Left click again to finish creating a Case Structure (alternatively, you could have left clicked and held the button to specify the initial size of the Case Structure.)
- Any Block Diagram components that were overlapped by the Case Structure at the moment of its creation have been automatically inserted into the Case Structure. If this has happened, drag them out of the Case Structure. The Case Structure should be empty at this point.

4.1.7 Create the Equal Comparison Function

You will now create the Comparison function which is used to check for a valid handle.

The *Equal?* Comparison Function
The Equal? comparison function receives two separate pieces of data, compares them, then outputs a value of True if the pieces of data are equal, or False if the pieces of data are unequal. Like every other LabVIEW function, the inputs are wired into the left side of the Equal? function, and the output is taken from the right side of the Equal? function.

- Right click an empty area of the Block Diagram. The Functions window appears.
- Hover the cursor over the *Arith & Compar* icon . The *Arithmetic & Comparison* window appears (seen below.)



- Hover your cursor over the *Comparison* icon. The *Express Comparison* window appears.
- Left click the *Equal?* function. All Windows vanish and you are left holding an Equal? function.
- Left Click an empty area to the left of the Case Structure to place the Equal? function.

4.1.8 Create the Numeric Constant.

You may recall creating a Numeric Constant in the first chapter. However, the previous instructions followed a shortcut method for creating the Numeric Constant. The following instructions will find the Numeric Constant within the Functions menu.

- Right click any empty area of your Block Diagram. The Functions menu opens.

- Hover your cursor over the Arith & Compar icon. The Arithmetic & Comparison window appears.

- Hover your cursor over the *Numeric* 123 icon. The *Express Numeric* window opens.

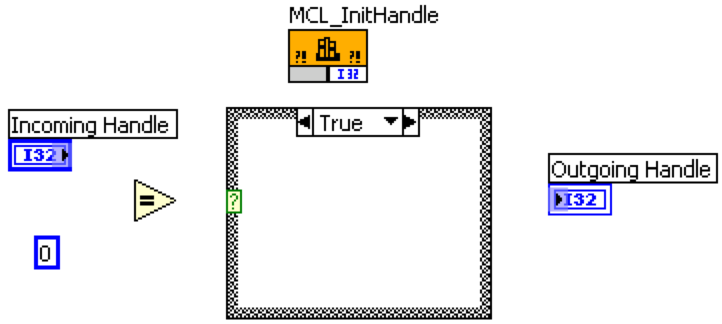
- Left click the *Num Const* 123 icon. The windows vanish and you will be left holding a Numeric Constant.

- Left click an empty area of the Block Diagram that is to the left of the Equal? function. This places the Numeric Constant (with an initial value of “0”.)

4.1.9 Arrange the Components to prepare for wiring

- Arrange the components so your Block Diagram appears similar to the image below.

- Leave the MCL_InitHandle Call Library Function Node outside of the case structure for now.



4.1.10 Make the Comparison Wire Connections

It’s now time to connect the inputs and output of the *Equal?* comparison function.

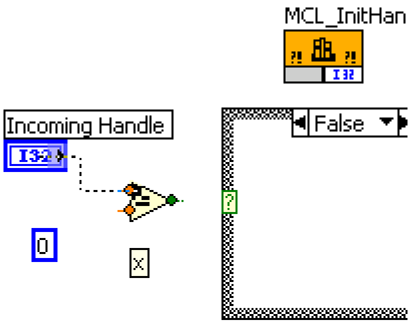
Using a Comparison function to control a Case Structure
In a situation in which a Handle is passed from outside this VI into the Incoming Handle control, the Handle should be passed directly to the Outgoing Handle indicator and right back out of this

VI. However, if no Handle is passed into this VI, an attempt will be made by this VI to get a Handle for the Nano-Drive®. How will this VI know when it has to acquire the Handle itself? Answer: Whenever the Incoming Handle control is empty (i.e. the Incoming Handle control is equal to “0”.)

- Left Click the output (right) side of the Incoming Handle control. You are now dragging a wire out of this control.

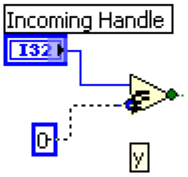
- Left click the top input of the *Equal?* comparison function. A wire has been created connecting the Incoming Handle control to this input.

Note
The ☒ in the image at right appears whenever the top input of a comparison function is highlighted. It’s telling us that the top input is the X in the comparison equation: *is X = Y?*



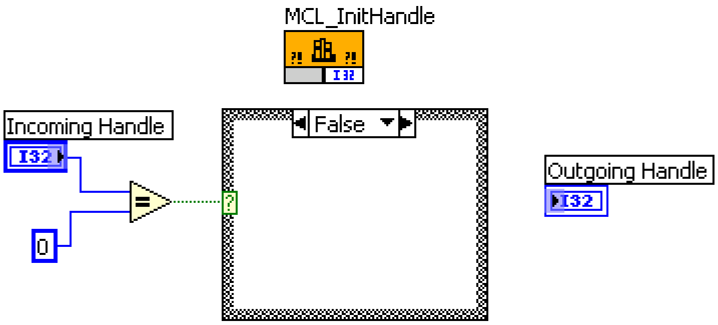
- Left click the output of the Numeric Constant. You are now dragging a wire out of the Numeric Constant.

- Left click the bottom input of the Equal? comparison function to finish creating a wire between the Numeric Constant and the Y input of the comparison function.



- Left click the output (right) side of the Equal? comparison function. You are now dragging a wire out of this comparison function.

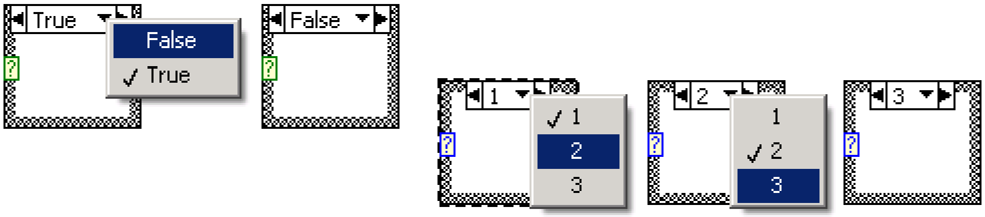
- Left click the Selector Terminal ? of the Case Structure. LabVIEW automatically connects the wire to the input (left) side of Selector Terminal.



4.1.11 Make the Outgoing Handle wire connections

Upon creation, your Case Structure was automatically given two cases; True, and False. Keep in mind that the title of these cases corresponds with the possible outputs of the Equal? function.

Understanding the Case Structure
The five Case Structure cases in the image below represent two individual Case Structures. The cases titled True, and False, represent one entire Case Structure; the cases titled 1, 2, and 3, are the second entire Case Structure.



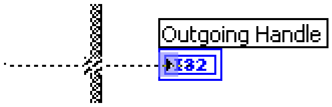
Cycling through cases (frames) of a Case Structure
On the Block Diagram only one case of a Case Structure is visible at a time. To view other cases belonging to the same Case Structure; left click the downward pointing arrow at right of the case title. This action pulls down a list of existing cases. Left click the title of a case to view it.

- Switch to the False case of your Case Structure.

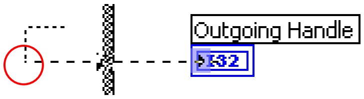
Note
If you find that your Case Structure is missing the True and/or False case, or other cases exist than the True and False cases; refer to the LabVIEW help file on Case Structures for help editing/deleting cases.

- Left Click the input (left) side of your Outgoing Handle indicator. You are now dragging a wire out of this indicator.

- Drag the wire into the Case Structure. You'll see the prototype of a *Tunnel* appear where the wire enters the Case Structure.

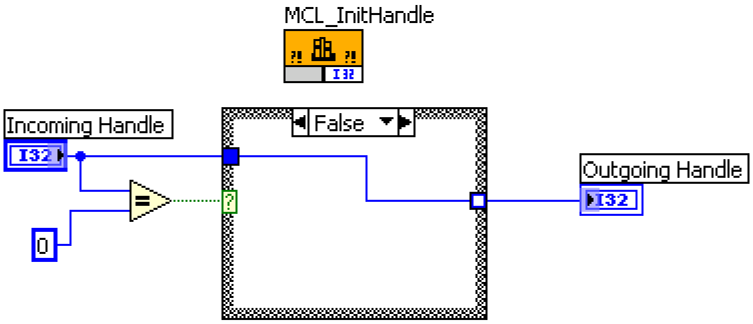


- Left click an empty area within the Case Structure. You're still dragging the wire, but you'll notice it has been pinned down at the point where you clicked. A red circle has been added to the image at right to illustrate where the wire was pinned.



- Drag the wire out the left edge of the Case Structure. Another Tunnel prototype appears where the wire exits the Case Structure.

- Left click the output of the Incoming Handle control. A wire has been created connecting the Incoming Handle control to the Outgoing Handle indicator through the False case of the Case Structure.



Case Structure Outputs
All Structure inputs/outputs pass through Tunnels ■, which often take on a distinct color according to the type of data passing through. However, the Tunnel passing the Incoming Handle wire through the output (right) side of the Case Structure is filled with white representing that it is incomplete: □. This occurred because an output was also created in the True case of the Case Structure. At this point, the output in the True case is erroneously devoid of data.

4.1.12 Prepare the True case of the Case Structure

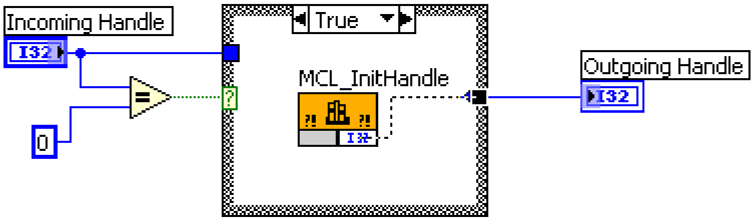
You will now create the code that will execute in the event that the value, "true", is passed to the Case Structure Selector Terminal.

- Switch to the True case of your Case Structure. The wire passing through the Case Structure appears to have vanished. In reality, it still exists but is hidden within the False case.

- Drag the MCL_InitHandle Call Library Function Node into the middle of the True case of your Case Structure.

- Left click any empty area of the Block Diagram to deselect the Call Library Function Node. This is a necessary step before you can create a wire at a terminal of this Call Library Function Node.

- Left click the "handle" output of the MCL_InitHandle function. You are now dragging a wire out of the Call library Function Node.



- Left Click the output (right side) Tunnel of the Case Structure. The wire is created and the Tunnel fills in with blue.

4.1.13 Troubleshooting

The VI is finished; however, you should not yet run it. Take a look at the Run arrow; if it is broken ⇏ , there are errors in the VI; continue with this section for assistance troubleshooting the VI. If your Run arrow is not broken ⇨ , skip the troubleshooting section.

- If your Run arrow appears unbroken ⇨, skip this section.

- Compare your Block Diagram to the images shown at the beginning of this chapter of the finished Block Diagram. Correct any differences you find between the two.

Note
Keep in mind that two cases exist within your Case Structure. Verify that both of the cases are configured correctly. If more than two cases exist, delete all except for the True, and False Cases. To delete a case of the Case Structure: right click the border of the Case Structure, select *Delete this case* from the menu that appears.

- Left Click the broken Run arrow ⇏ The Errors List window appears.

- Read through the list of errors. Read the Details for each error (there are often helpful recommendations here.)

Note
The LabVIEW Error List shows all outcomes of a mistake, not necessarily the mistake itself. Though several errors may be listed, they may all originate from one missing wire connection or unconnected input.

- Double left click any item of the Error List and LabVIEW will find it on the Block Diagram for you.

- If problems persist, restart this chapter with a new VI. In some cases, this may be the best way to discover the error you've overlooked.

Notes on section 4.1.14

The next VI you create will use this SubVI to obtain a handle for the Nano-Drive®.

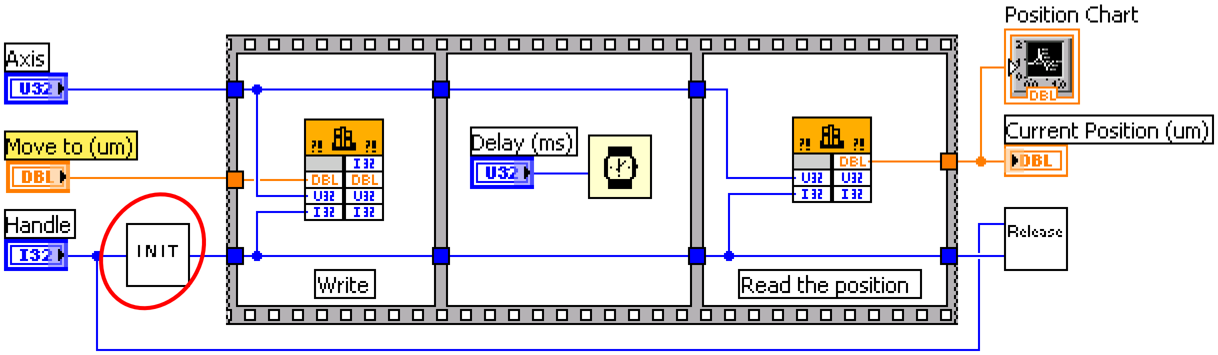
- The Standard Init SubVI is complete. Save this VI in a new folder with the filename: StandardInitSubVI.

Handle With Care

This SubVI is not meant to be run alone. There is no provision in this VI for releasing the Handle it has obtained. Because of this, the VI will continue to hold the Handle *even after the VI has stopped running!* This would disable other applications which try to communicate with the Nano-Drive™. If this happens, you must exit and restart LabVIEW to force the release of all handles held by LabVIEW VIs.

The Standard Init SubVI

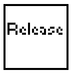
Shown below is an image of the finished Block Diagram following completion of Chapter 3. Circled in red is the portion you've created so far. It should now be clear to you how this portion of the VI works: If the Handle passed into the Init SubVI is equal to zero, the Init SubVI acquires the Handle itself. If the incoming Handle is not zero, it is passed to the output. In either case, a Handle is passed out for use within the parent VI.



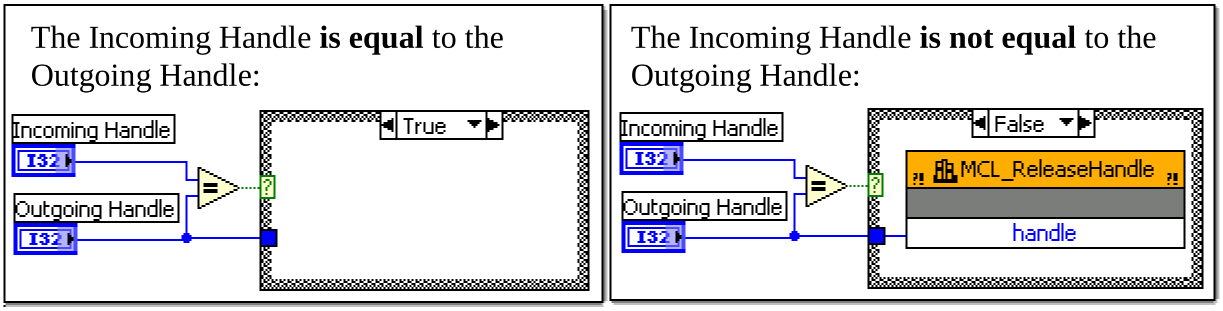
Section 4.2 – Creating the Release SubVI

The process of creating the Release SubVI will closely resemble the previous process by which you created the Init SubVI.

The Release SubVI



Below can be seen the Block Diagram of the Release SubVI which you will be creating in this section of the tutorial. Because a Case Structure is used within this VI, it is again necessary to show two separate states of the same Block Diagram. It is not necessary to see the Front Panel of this VI; the Front Panel simply contains the Incoming Handle and Outgoing Handle Numeric Controls.

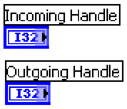


The Incoming Handle **is equal** to the Outgoing Handle:

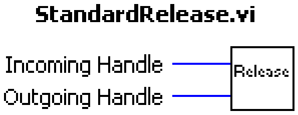
The Incoming Handle **is not equal** to the Outgoing Handle:

(Block Diagram of the finished Release SubVI)

Inputs of the Release SubVI



The Release SubVI has two inputs and no outputs. The inputs are the Incoming Handle and Outgoing Handle Numeric Controls (seen at left.) The names of the inputs refer to the input and output of the Init SubVI (you created previously) on the Read/Write VI Block Diagram.



Stand-Alone VI versus SubVI

The Init, and Release, SubVIs are entirely dependent on the instructions of a concurrently running VI(s) to accomplish anything meaningful or to avoid causing critical malfunctions. A VI meant to stand alone (as its own application) must contain within its Block Diagram all necessary programming to gather all required resources and must use the resources in some meaningful way, handle potential errors, and release the gathered resources back to a non-volatile state. This can, of course, be done with the assistance of SubVIs; the important point is that a VI contains a complete program, whereas a SubVI often only contains a supplementary set of instructions.

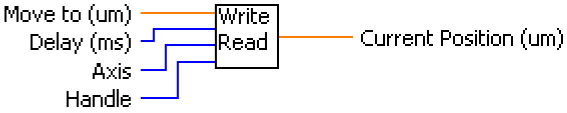
However, it is sometimes necessary that a VI capable of standing alone also be ready for use as a SubVI.

Stand-alone and SubVI Capability

The Read/Write VI that you are creating in this chapter will eventually be used within another VI in a later chapter as a SubVI. However, by the end of this chapter the Read/Write VI will be ready to function as its own useful application. This is not a trivial property. In fact, this VI has been specifically designed to work as both a stand-alone VI, and a SubVI.

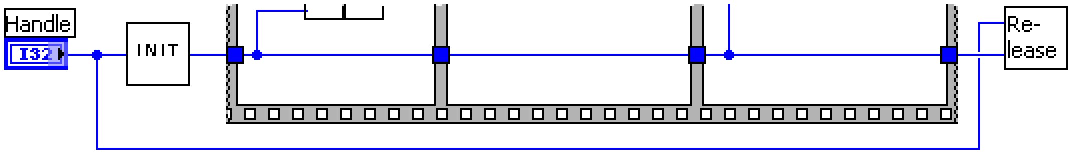
The Read/Write VI as a SubVI

The purpose of the Release SubVI can only be understood with the previous fact in mind. Take a close look at the previous images of the finished Release SubVI, and at the image of the finished Read/Write Block Diagram. The Handle is only released when the Incoming Handle control is different from the Outgoing Handle control. Recall from creating the Init SubVI; this cannot be the case when the Incoming Handle is different than “0”. This means the following: a Handle passing to the Init SubVI happens when the Read/Write VI is being used as a SubVI; therefore, the Handle should not be released (by the Release SubVI) at the termination of the Read/Write SubVI.

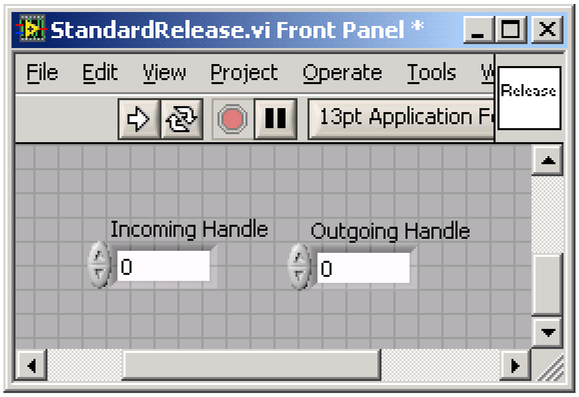


The Read/Write VI as a Stand-Alone Application

It is now apparent that the Read/Write VI does not initialize or release a Handle when used as a SubVI. However, when the Read/Write VI is used as a stand-alone VI, it will both initialize, and release, a handle. The Init SubVI knows to initialize a Handle because it's "Incoming Handle" control is equal to zero. The Release SubVI knows to release the Handle because its "Incoming Handle" control is not equal to its "Outgoing Handle" control.



4.2.1 Create the Two Numeric Controls

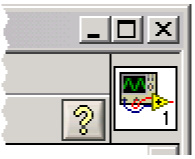

StandardRelease.vi Front Panel *
File Edit View Project Operate Tools W
13pt Application F
Release
Incoming Handle
0
Outgoing Handle
0

- Begin LabVIEW and start a new VI.
- Save this VI in the same folder that contains the StandardInitSubVI.vi under the new filename: StandardReleaseSubVI.
- On the Front Panel, create a Numeric Control.
- Change its data type representation to I32 (signed 32-bit integer.)
- Change its label to "Incoming Handle."
- Create another Numeric Control.
- Change its data type representation to I32.
- Change its label to "Outgoing Handle."

4.2.2 Create the Terminals of the VI

- Right Click the icon that appears in the top-right corner of your Front Panel.

- Select *Show Connector* from the menu that appears. The Icon should have switched from the graphic of an oscilloscope to the Connector Pane of your VI.

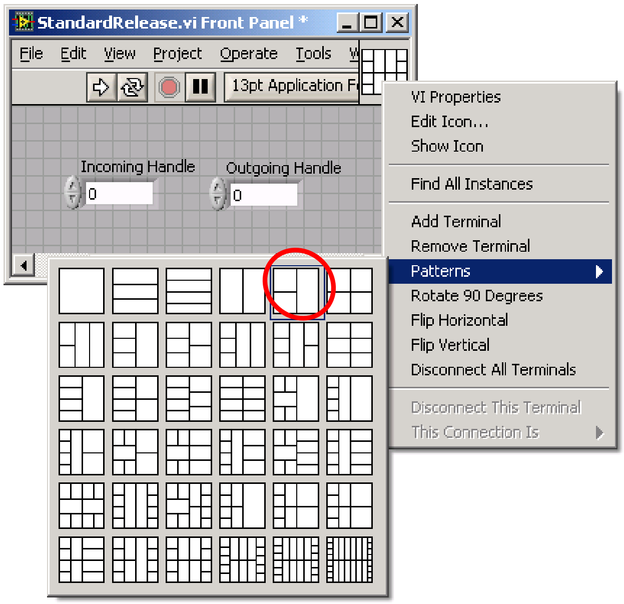


Note
You cannot Show Connectors from the Block Diagram. You will not have access to the options shown in the image below unless you are on the Front Panel.

- Right click the Connector Pane Icon

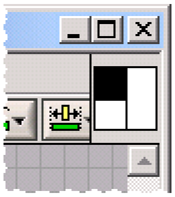
- Hover the cursor over the *Patterns* item of the menu that appears (seen at right.)

- Select the fifth pattern to the right on the first row. In the image to the right, a red circle has been added to indicate which pattern to choose.



4.2.3 Set the input and output of the VI

- Left click the top input of your VI's Connector Pane. The top-left quarter of the Connector Pane should turn black (as seen on the right.)



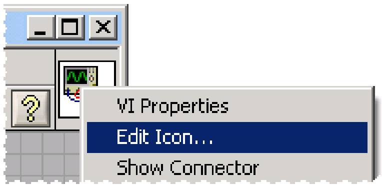
- Left click the Incoming Handle control. The top-left quarter of your Connector Pane should turn blue (if it changes from black to a color other than blue, verify the Incoming Handle control is set as an I32.)

- Left Click the bottom input of your VI's Connector Pane.

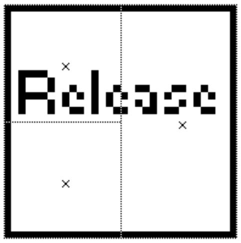
- Left Click the Outgoing Handle Control. The bottom-left quarter of your Connector Pane should turn blue.

4.2.4 Edit the icon

- Right click the Connector Pane icon.



- Select *Show Icon* from the menu that appears. The Connector Pane graphic should have been replaced with the more familiar graphic of an instrument displaying a sine wave.
- Once again, right click the icon.
- Select *Edit Icon* from the menu that appears. The *Icon Editor* opens.
- Clear the image to white. This can be done either by painting white over the entire image, or by selecting *Clear* from the *Edit* menu.
- Select the A text tool from the toolbar on the left side of the Icon Editor.
- Left click close to the left border of the icon. You can now apply text to this area.
- Enter this word: Release



- Draw a black border around the perimeter of the icon using either the pencil, line, or unfilled square, tool.
- Check the *Show Terminals* option on the right side of the Icon Editor. Your image should now appear very similar to the one shown at right.
- Click the *OK* button at the bottom-right of the Icon Editor.

4.2.5 Configure a Call Library Function Node to Call MCL_ReleaseHandle

The Front Panel is finished. Switch to the Block Diagram (hold *Ctrl* and press *E*.)

- From the Block Diagram, create a Call Library Function Node. Placement of the Call Library Function Node is not important at this point.
- Enter its *Configure* menu.
- Direct the *Library name or path* to the Madlib.dll using the browse button.
- Under *Function name*, select *MCL_ReleaseHandle*.

- Left click the *Parameters* tab at the top of the Configure menu.

- Left click the *Add a Parameter* + button once.

- Left click the newly added parameter, “arg1”.

- In the *Current parameter* area; replace the name, “arg1”, with the name: “handle”.

- All other settings should be correct at their default values. Verify that the *Type* is “Numeric”, and the Data Type is “Signed 32-bit Integer”.

  Your Function Prototype should now read:

```
void  MCL_ReleaseHandle(int32_t handle);
```

- Left click *OK* to accept the changes.

4.2.6 Create the Case Structure and the Equal? Comparison Function

- Create a *Case Structure* in an empty area of the Block Diagram.

- Create an *Equal?* function.

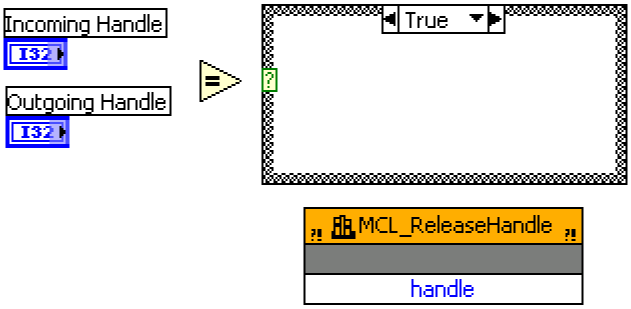
- Place the Equal? function in an empty area to the left of the Case Structure.

4.2.7 Arrange the Components in Preparation for Wiring

- Verify that your Block Diagram contains all components appearing in the image below.

Note
Don’t forget the Call Library Function Node has two distinct icon formats; Names, and No Names. The Call Library Function Node below is shown with the Names format selected.

- Drag each component of the Block Diagram to the approximate position relative to the Case Structure as seen below.



- Drag the MCL_ReleaseHandle Call Library Function Node into the False case of your Case Structure.

4.2.8 Make the Wire Connections

You will now be instructed to make the required wire connections. However, you will no longer be given step-by-step instructions for doing so.

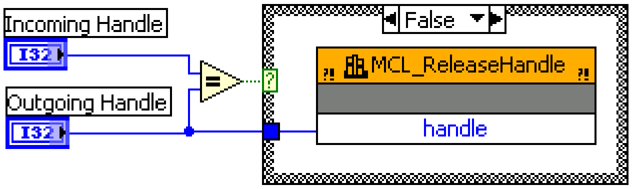
- Connect the output of the Incoming Handle control to the top input of the Equal? comparison function.

- Connect the output of the Outgoing Handle control to the bottom input of the Equal? comparison function.

- Connect the output of the Equal? comparison function to the Selector Terminal of the Case Structure.

- Connect the "handle" input of the MCL_ReleaseHandle Call Library Function Node to the output of the Outgoing Handle control.

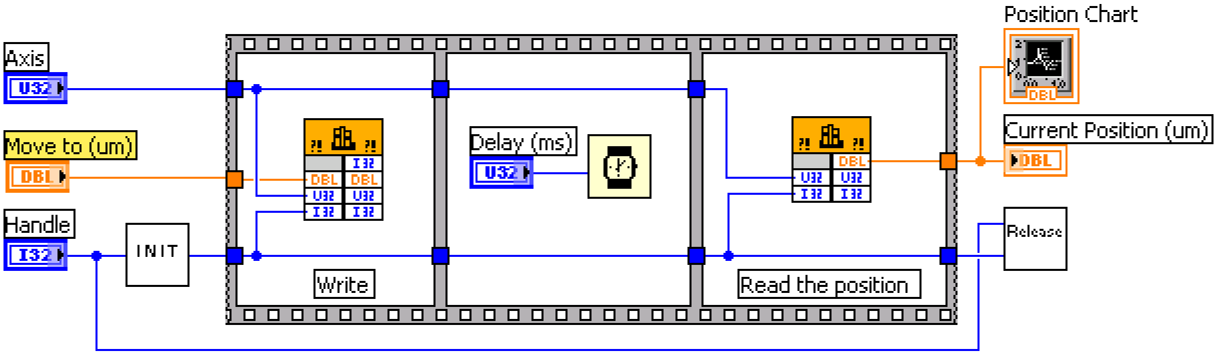
- Verify that your Block Diagram matches the image below before continuing. The True case of your Case Structure should be empty.

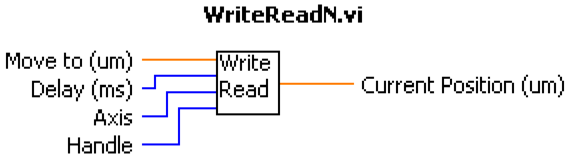


Section 4.3 – Create the Write/Read SubVI

Project Goal

In the following Section, you will create a Write/Read VI which incorporates the SubVIs you've created. As suggested by the title of this section, the importance of the Write/Read VI is that it too can be used as a SubVI.



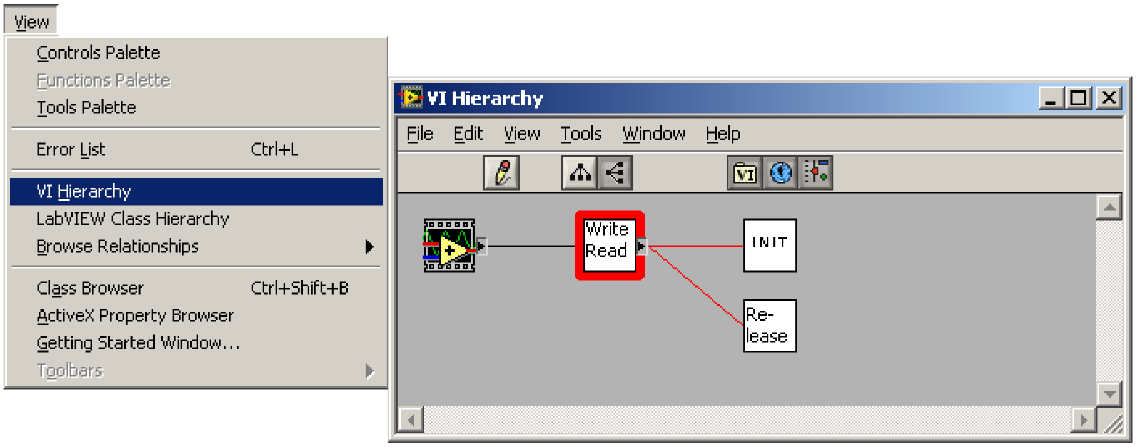


VI Hierarchy

The image below shows the hierarchy of the VIs created in this chapter. The following is an excerpt from the LabVIEW help document (*index* VI Hierarchy window):

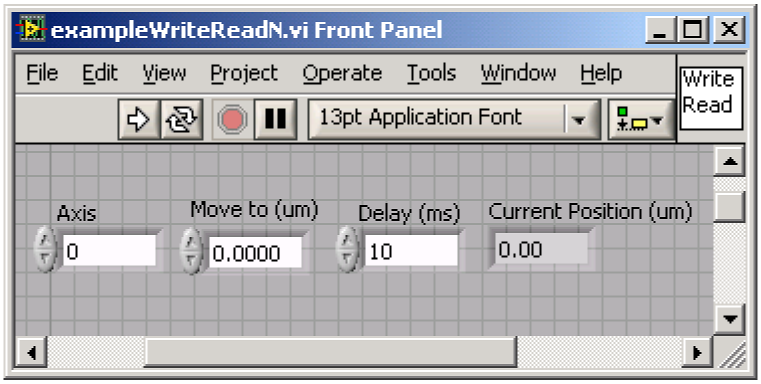
"Use this window to view the subVIs and other nodes that make up the VIs in memory and to search the VI hierarchy. This window displays all open LabVIEW projects and targets, as well as the calling hierarchy for all VIs in memory, including type definitions and global variables."

The VI Hierarchy window can be accessed at any time from the *View* menu.



4.3.1 Start a New VI, Add the Front Panel Components

- Start a New VI, save it as: exampleWriteReadN.
- From the Front Panel, create four Numeric Controls.
- Name the Controls: Axis, Move to (um), Delay (ms), Handle.
- Change the Representation of the "Axis" control to U32.
- Change the Representation of the "Move to (um)" control to DBL.
- Change the Representation of the "Delay (ms)" control to U32.
- Change the Representation of the "Handle" control to I32.
- Edit the Properties of the "Move to (um)" control.
- Switch to the Display Format tab.
- Edit the Display Format to display as a Floating Point with 4 digits of precision.
- Create a Numeric Indicator.
- Name the indicator: "Current Position (um)".
- Change the Representation of the "Current Position (um)" indicator to DBL.
- Edit the Display Format of the Indicator to display as a Floating Point with 2 Digits of Precision.


exampleWriteReadN.vi Front Panel
File Edit View Project Operate Tools Window Help
Write
Read
13pt Application Font

Axis: 0
Move to (um): 0.0000
Delay (ms): 10
Current Position (um): 0.00

4.3.2 Create and Configure the Chart

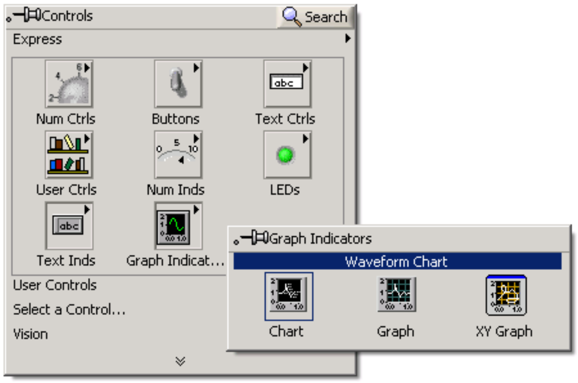
You will now be instructed to create a Waveform Chart that will plot the position value (0-100) of the Nano-Drive® on the Y-axis, maintaining a history of several write/read operations at a time. Every time the VI is run, the Chart will increment the X-axis.

The Waveform Chart

The Waveform Chart acts like any other indicator. Data is passed to the Waveform Chart via the Block Diagram, the data is then plotted according to its value onto the Y-axis of the Waveform Chart. After plotting a Y-axis value the Waveform Chart increments its X-axis readying itself to plot additional data onto the Y-axis.

Note

When programming the Block Diagram it is important to conform to the conventions of the LabVIEW environment. Avoid extracting data from Indicators; avoid manipulating the values contained within Controls. The History data of the Waveform Chart is an array, but using it as such would be an unconventional, and potentially hazardous, misuse of an indicator.



- Right Click an empty area of the Front Panel. The Functions menu appears.

- From the Functions Palette; select the *Express>>Graph Indicator* category. The *Graph Indicators* window appears.

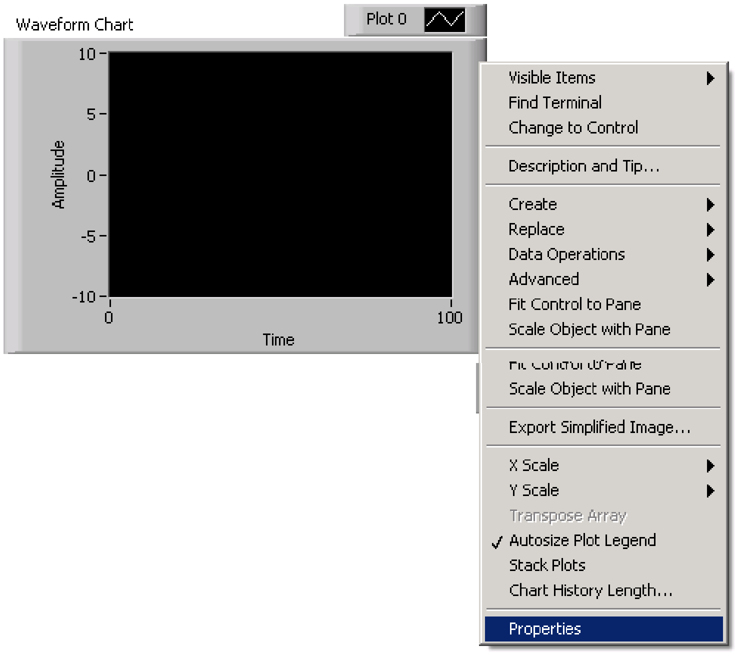
- Left Click the *Waveform Chart* icon. The windows vanish, and you are left holding the footprint of a Waveform Chart.

- Left Click an empty area of your Front Panel to place the Waveform Chart.

- Right click an edge of the Waveform Chart, and select *Properties* from the shortcut menu. The *Chart Properties* window appears.

- From the Appearance tab, change the Label to "Position Chart".

- Left click the *Scales* tab.

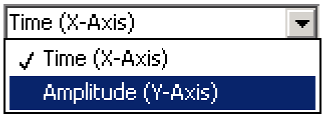


- Change the *Name* of the X-Axis to “Run #”.

- Change the *Maximum* of the X-Axis to “10”.

- Left click the pull-down menu at the top of the Scales tab.

- Select “Amplitude (Y-Axis)” from the list.



- Change the *Name* of the Y-Axis to Position (um).

- Left click the *Autoscale* checkbox to remove the check mark from it. The Minimum and Maximum range of the Chart should now be editable.

- Change the Minimum of the Y-Axis to “0”.

- Change the Maximum of the Y-Axis to “100”.

- Left click OK to accept the changes and exit the Chart Properties menu.

4.3.3 Create and Configure the Flat Sequence Structure, and Timer

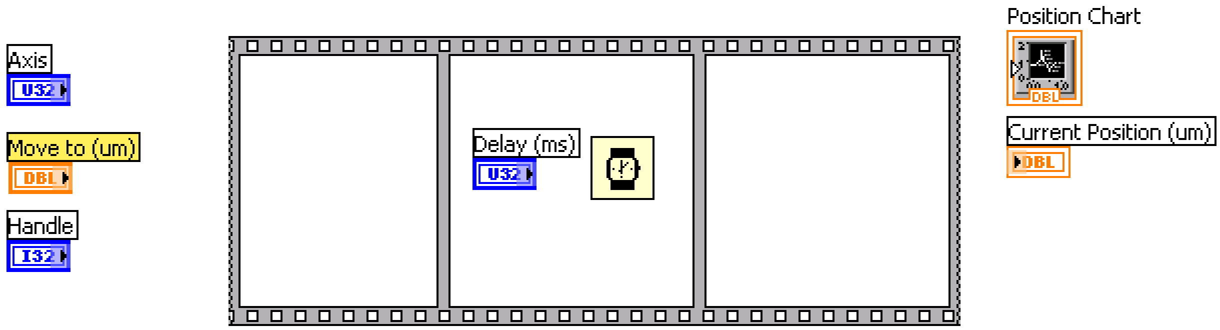
- From the Block Diagram, create a Flat Sequence Structure.

- Add two frames to the Flat Sequence Structure for a total of three frames.

- Create a *Wait (ms)* Timer in the second frame of the Flat Sequence Structure.

- Drag the “Delay (ms)” control inside the second frame of the Flat Sequence Structure, and place it to the left of the Wait (ms) Timer.

- Resize the frames of the Structure, and position the other Block Diagram components to match the image below.



4.3.4 Create the Function Call for MCL_SingleWriteN

- Create a Call Library Function Node in the first frame of your Flat Sequence Structure.

- Enter the Configure menu of the Call Library Function Node.

- Under the Function tab, change the Library name or path to the address of the MADlib.dll

- From the *Function name* pull-down menu, select MCL_SingleWriteN.

- Switch to the Parameters tab.

- Add three new parameters + .

- Left click the first of the new parameters, arg1, and configure the *Current parameter* area to the following: Change the Name to “position”, Type to “Numeric”, Data type to “8-byte Double”, and Pass to “Value”.

- Left click the second of the new parameters, arg2, and configure the Current parameter area to the following: change the Name to “axis”, Type to “Numeric”, Data type to “Unsigned 32-bit Integer”, and Pass to “Value”.

- Left click the third of the new parameters, arg3, and configure the Current parameter area to the following: change the Name to “handle”, Type to “Numeric”, Data type to “Signed 32-bit Integer”, and Pass to “Value”.

- Verify that your Function Prototype appears as follows:

```
void MCL_SingleWriteN(double position, uint32_t axis, int32_t handle);
```

- Left click OK to accept the changes and exit the Call Library Function configuration menu.

4.3.5 Create the Function Call for MCL_SingleReadN

- Create a Call Library Function Node in the third frame of your Flat Sequence Structure.

- Enter the Configure menu of the Call Library Function Node.

- Under the Function tab, change the Library name or path to the address of the MADlib.dll

- From the *Function name* pull-down menu, select MCL_SingleReadN.

- Switch to the Parameters tab.

- Left click the parameter, return type, and configure the Current parameter area to the following: change the Name to “current position”, Type to “Numeric”, and Data type to “8-byte Double”.

- Add two new parameters.

- Left click the first of the new parameters, arg1, and configure the Current parameter area to the following: change the Name to “axis”, Type to “Numeric”, Data type to “Unsigned 32-bit Integer”, and Pass to “Value”.

- Left click the second of the new parameters, arg2, and configure the Current parameter area to the following: change the Name to “handle”, Type to “Numeric”, Data type to “Signed 32-bit Integer”, and Pass to “Value”.

- Verify your Function Prototype appears as follows:

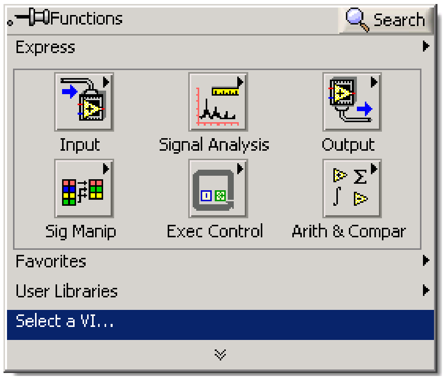
```
double MCL_SingleReadN(uint32_t axis, int32_t handle);
```

- Left click OK to accept the changes and exit the Call Library Function configuration menu.

4.3.6 Add the Init and Release SubVIs

You will now add the Init, and Release, SubVIs. As you will see, the SubVIs can be added to the VI like any other component you've added so far.

- Right click an empty area of the Block Diagram. The Functions palette appears.



- Left click the *Select a VI* menu item. A new window appears with the title: *Select a Vi to Open*

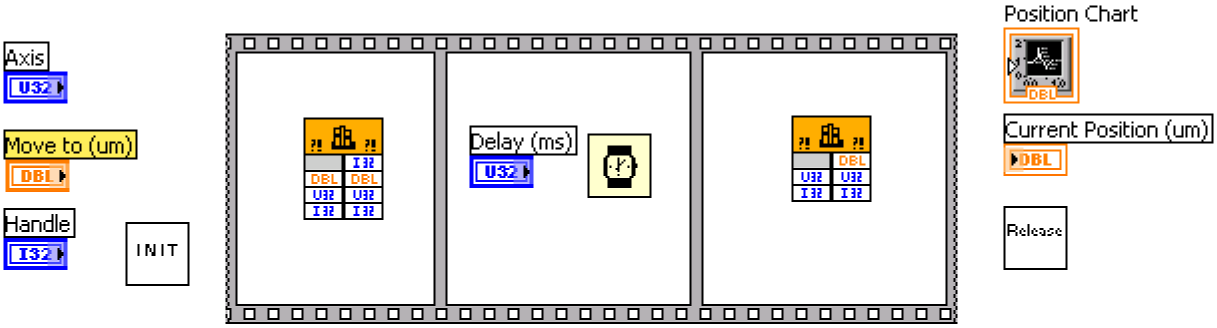
- Use the Select a VI to Open window to locate your StandardInitSubVI.vi.

- Left click your StandardInitSubVI.vi; it should appear in the *File name:* field near the bottom of the window.

- Left click the OK which appears at the bottom-right of the window. The window vanishes, and you are left holding the icon of your StandardInitSubVI.vi

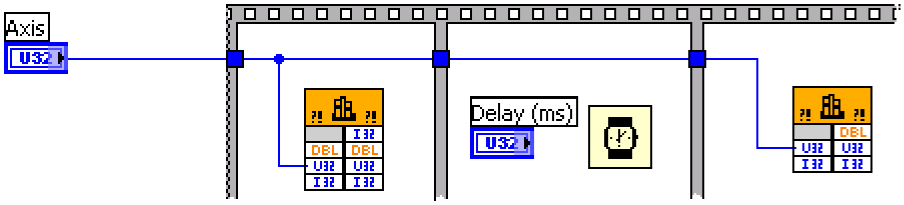
- Left click an empty area at the left of the Flat Sequence Structure to place the StandardInitSubVI.vi

- Follow the same process to locate and place the StandardReleaseSubVI.vi to the right of the Flat Sequence Structure.

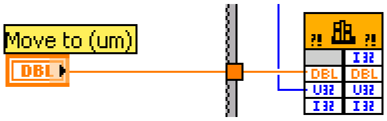


4.3.7 Make the Wire Connections

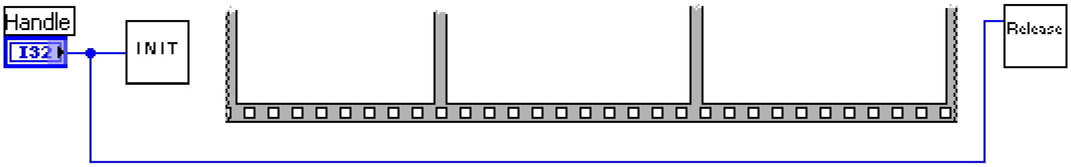
- Connect the output of the ‘Axis” control to the “Axis” input of both Call Library Function Nodes. Make sure to pass the wire through the Flat Sequence Structure as seen in the image below.



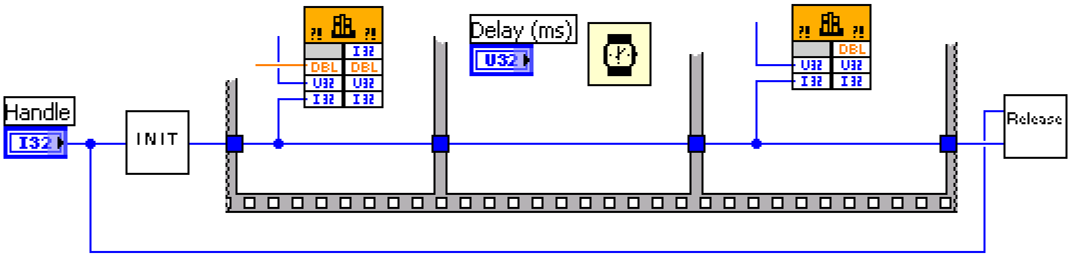
- Connect the output of the “Move to (um)” control to the “Position” input of the MCL_SingleWriteN Call Library Function Node.



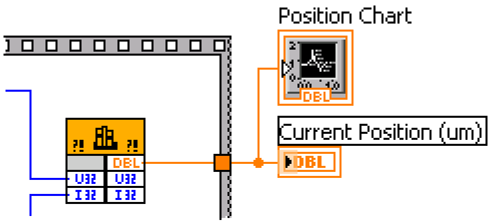
- Connect the “Handle” control to both the INIT SubVI input, and the Release SubVI top input (Incoming Handle.) The wire connecting to the Release SubVI should not pass through the Flat Sequence Structure (as seen in the image below.)



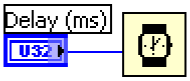
- Connect the output of the INIT SubVI to the Handle inputs of both Call Library Function Nodes, as well as the bottom input of the Release SubVI (Outgoing Handle.)



- Connect the top-most output of the MCL_SingleReadN Function Node (current position) to the input of both the “Current Position (um)” control, and “Position Chart” Waveform Chart.



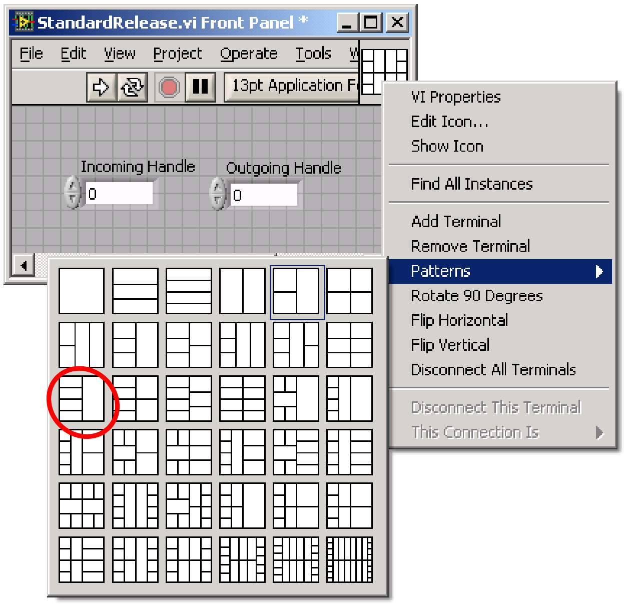
- Connect the output of the "Delay (ms)" control to the input of the Wait (ms) Timer.



4.3.8 Create the Terminals of the VI

- From the Front Panel, right click the icon of the VI.

- Select *Show Connector* from the menu that appears. The Icon should have switched from the graphic of an oscilloscope to the Connector Pane of your VI.



Note
You cannot Show Connectors from the Block Diagram. You will not have access to the options shown in the image at right unless you are on the Front Panel.

- Right click the Connector Pane Icon .

- Hover the cursor over the *Patterns* item of the menu that appears (seen below.)

- Select the third pattern down from the top-left corner. In the image at right, a red circle has been added to indicate which pattern to choose.

4.3.9 Set the input and output of the VI

- Left click the top input (left side) of your VI's Connector Pane. The top-left eighth of the Connector Pane should turn black as seen to the right.

- Left click the "Move to (um)" control. The top-left eighth of your Connector Pane should turn orange (if it changes from black to a color other than orange, verify the Move to (um) Control is set as a DBL.)

- Left click the second input of your VI"s Connector Pane.

- Left click the 'Delay (ms)' control. The higher middle-left eighth of your Connector Pane should turn blue.

- Left click the next, or third input of your VI's Connector Pane.

- Left click the "Axis" control. The third input of your Connector Pane should turn blue.

- Left click the bottom input of your VI's Connector Pane.

- Left click the "Handle" control. The fourth input of your Connector Pane should turn blue.

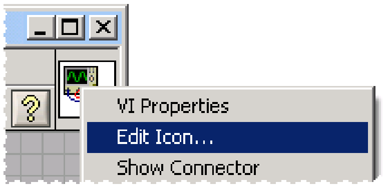
- Left click the right (output) half of your VI's Connector Pane.

- Left click the "Current Position (um)" indicator. The right half of your VI's Connector Pane should turn orange.

4.3.11 Edit the Icon

- Right click the Connector Pane icon.

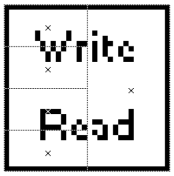
- Select *Show Icon* from the menu that appears. The Connector Pane graphic will be replaced with the graphic of an instrument displaying a sine wave.



- Once again, right click the icon.

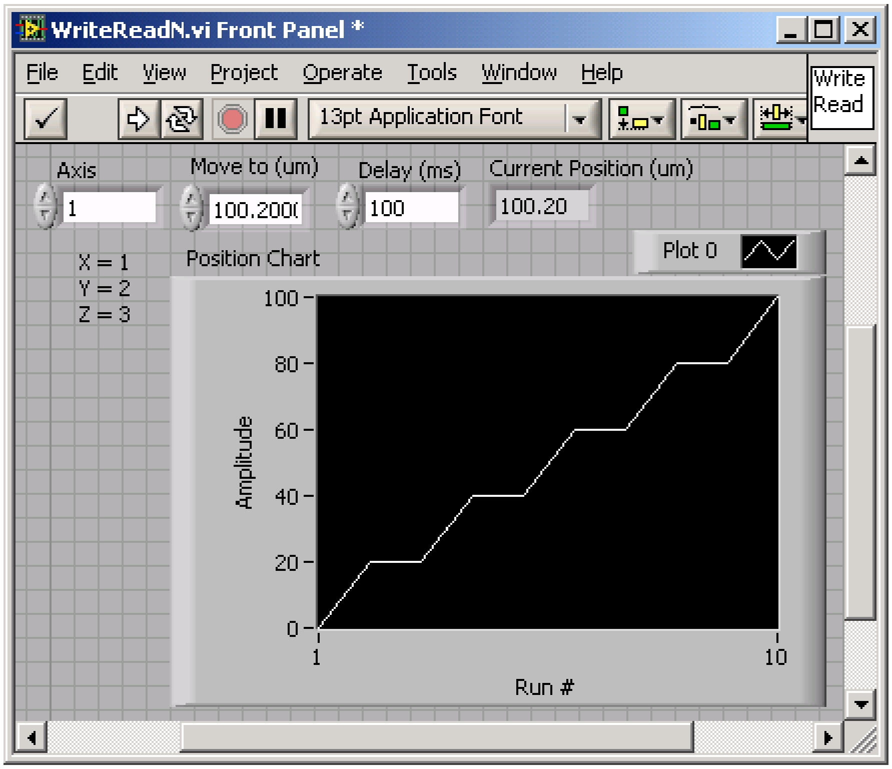
- Select *Edit Icon* from the menu that appears. The *Icon Editor* opens (as seen below.)

- Edit the Icon graphic to appear similar to the image at right.



Section 4.4 – Run the Write/Read VI

You now have a VI that will write a position command to the Nano-Drive® and read the position from the Nano-Drive®. Though this VI performs similarly to the VI created during Chapter 3, this VI is much better suited to serve as a SubVI within the VI you will create in Chapter 5.


WriteReadN.vi Front Panel *

Axis: 1

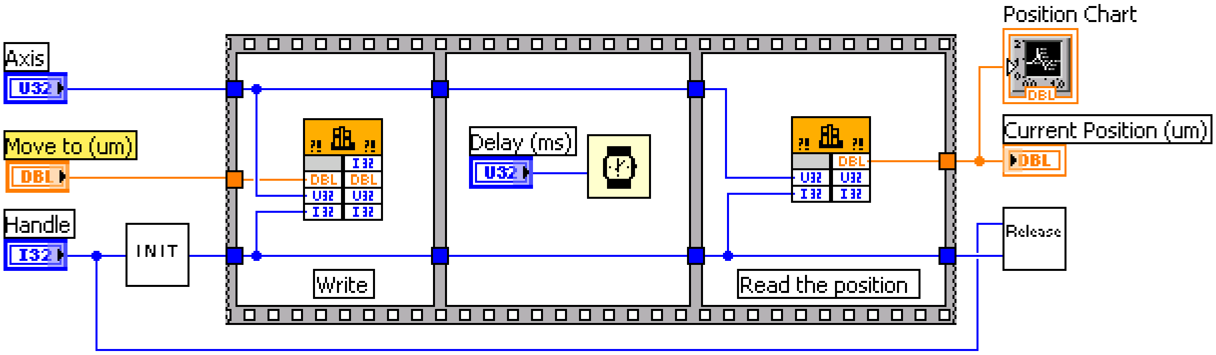
Move to (um): 100.200(

Delay (ms): 100

Current Position (um): 100.20

X = 1
Y = 2
Z = 3

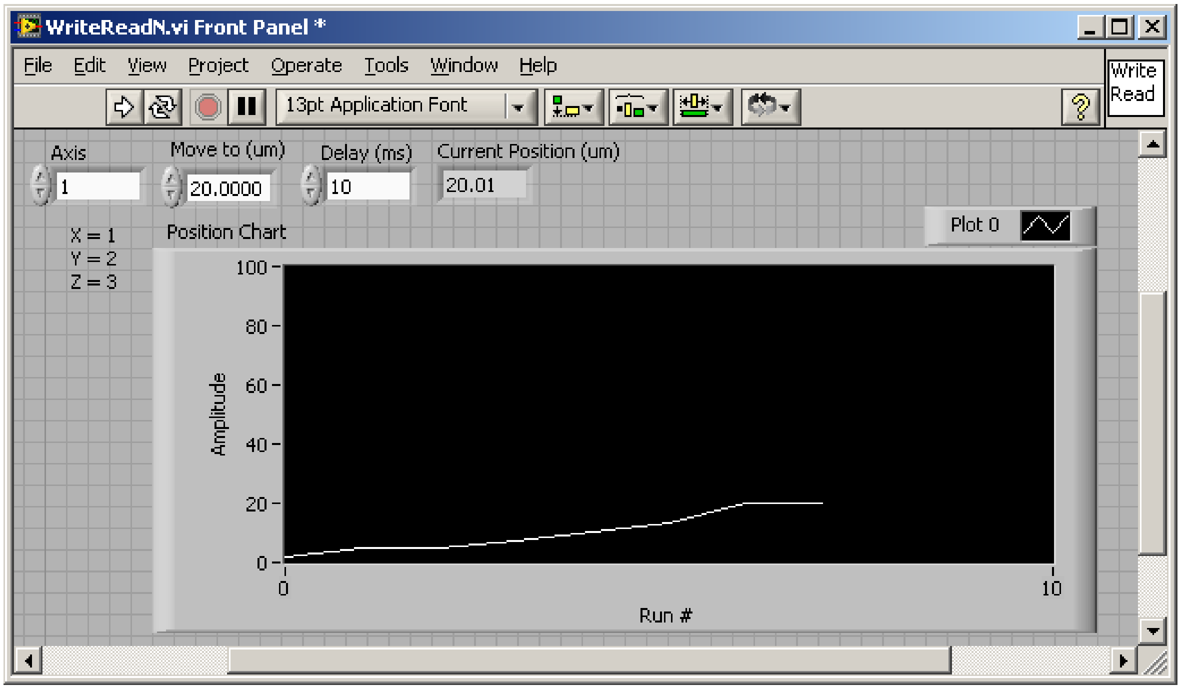
(Front Panel of the finished Chapter 4 VI after a series of executions)



(Block Diagram of the Finished Chapter 4 VI)

4.4.1 Prepare to Run the VI

- Save the VI under the filename: exWriteReadN
- Turn on the Nano-Drive®.
- Switch to the Front Panel of your VI.
- Select an axis to move (X = 1, Y = 2, Z = 3).
- Enter a position to move to into the “Move to (um)” control. Choose a value within the range 0-50.
- Set the “Delay (ms)” control to “10”
- Run the VI. The “Current Position (um)” indicator displays a number from 0-50 according to the position read from the specified axis at 10 milliseconds following the Write command. The Graph has drawn a line from point: (0,0), to point: (1,X); where X is the position of the specified axis.



WriteReadN.vi Front Panel *

Axis: 1

Move to (um): 20.0000

Delay (ms): 10

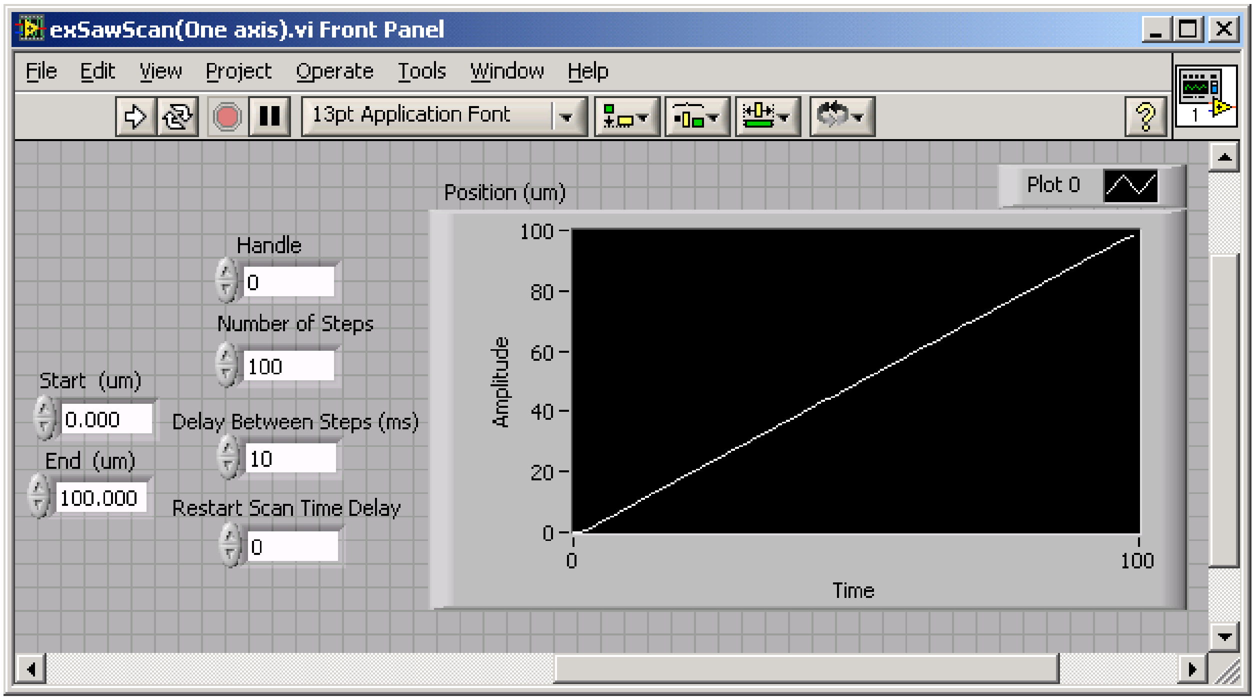
Current Position (um): 20.01

X = 1
Y = 2
Z = 3

(Image of the Completed VI’s Front Panel after a series of Runs.)

Chapter 5: Create a VI to Scan an Axis

The VI shown below will be created in Chapter 5.


exSawScan(One axis).vi Front Panel
File Edit View Project Operate Tools Window Help
13pt Application Font
Position (um)
Plot 0
Amplitude
100
80
60
40
20
0
0
100
Time

Handle: 0

Number of Steps: 100

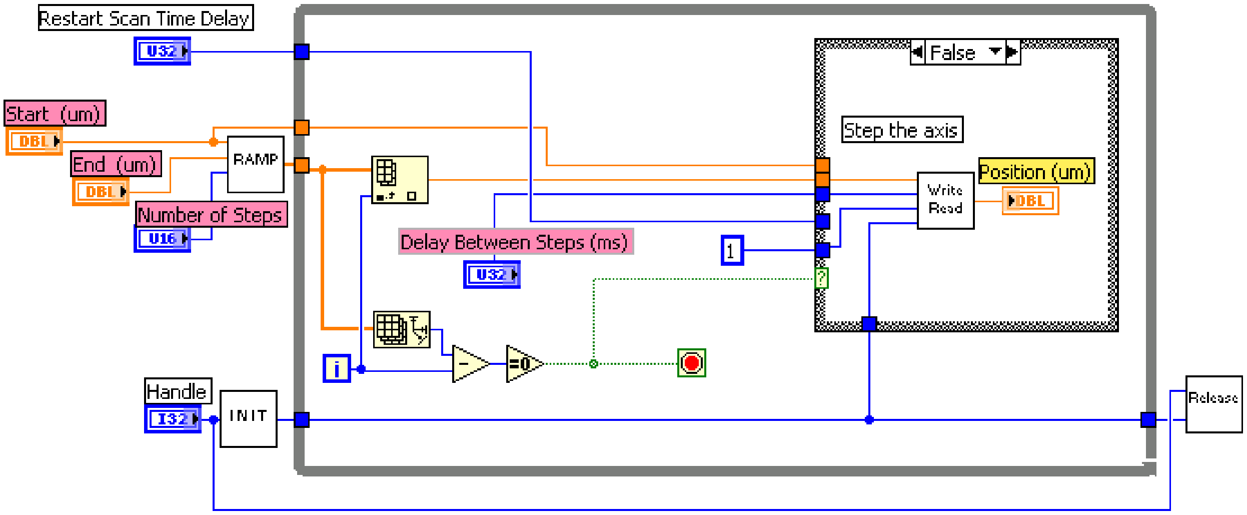
Start (um): 0.000

Delay Between Steps (ms): 10

End (um): 100.000

Restart Scan Time Delay: 0

(Image of the finished Front Panel by the end Chapter 5)



(Image of the finished Block Diagram by the end of Chapter 5)

Goal of Chapter 5

Many applications for nanopositioners require that one or more axis be scanned while data are acquired. In some cases, the Z axis is scanned while images are taken with a camera. In other cases, the X and Y axis may be raster scanned to produced an image, for example in scanned probe microscopy. This Chapter is about scanning, and consists of two sections. During the first section you will program a SubVI to calculate an array of values to be used as position commands. During the second section you will program a VI to command the Nano-Drive® according to values obtained from the SubVI created in the first section.

Topics Include:

1. Introduction to Arrays
2. Introduction to the For Loop
3. Using Shift Registers to carry data through iterations of a looping Structure

MADlib.dll Functions Used Within This Chapter Include:

1. void MCL_ReleaseHandle(int handle);
2. int MCL_InitHandle();
3. int MCL_SingleWriteN(double nposition, unsigned int axis, int handle);
4. double MCL_SingleReadN(unsigned int axis, int handle);

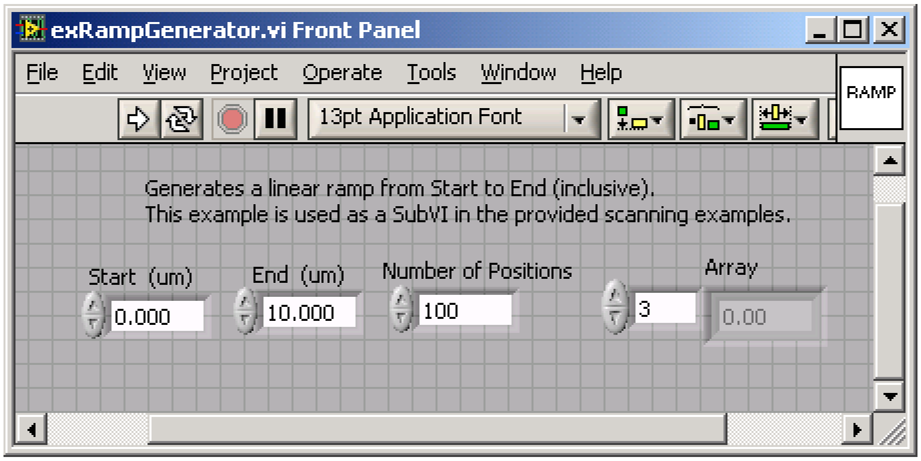
SubVIs Used Within This Chapter Include:

3. StandardInit.vi / StandardInitSubVI.vi
4. StandardRelease.vi / StandardReleaseSubVI.vi
5. RampGenerator.vi / exRampGenerator.vi *(new)*

Section 5.1 – Create the Ramp Generator SubVI

Project Goal

Unlike other VIs you’ve created so far, this VI will not directly communicate with the Nano-Drive™. Instead, the sole purpose of this VI is to create an array of position values which can be subsequently written to the Nano-Drive® DAC.



Generates a linear ramp from Start to End (inclusive).
This example is used as a SubVI in the provided scanning examples.

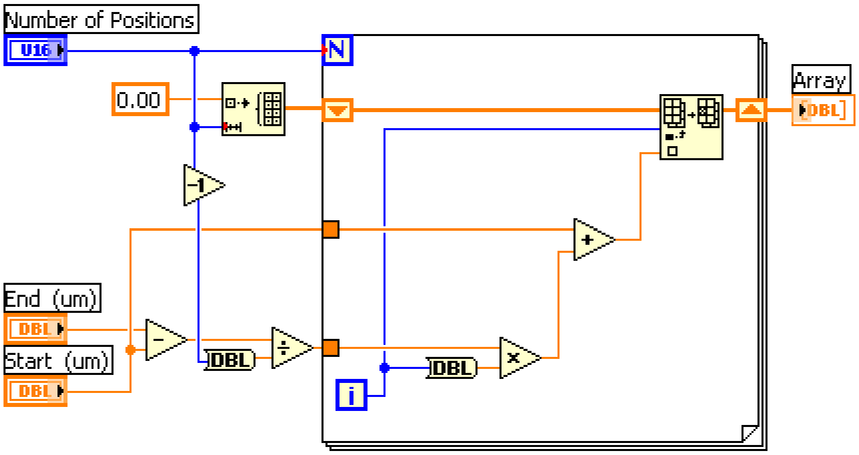
Start (um) 0.000

End (um) 10.000

Number of Positions 100

Array 3 0.00

(Image of the finished Front Panel of the ramp generating SubVI)



(Image of the finished Block Diagram of the ramp generating SubVI)

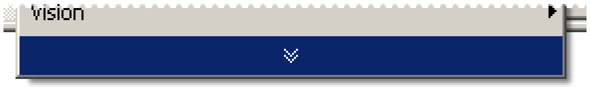
5.1.1 *Create the Three Numeric Controls*

- Begin LabVIEW and start a new VI by selecting *Blank VI* from the *Getting Started* Window. The Front panel and Block Diagram of a new VI appear.

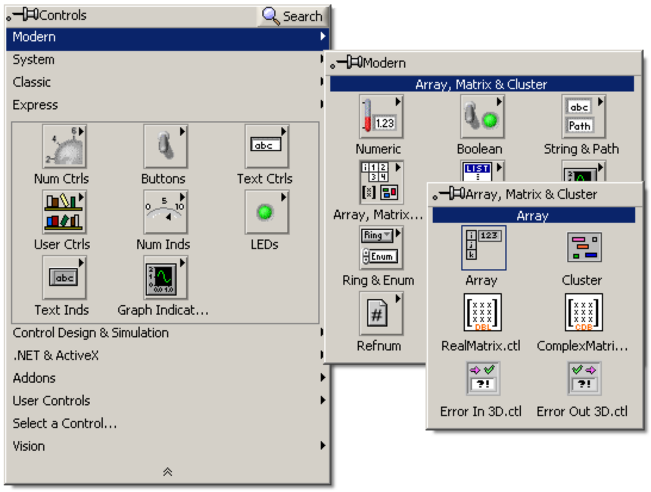
- Save the VI as: exRampGenerator.vi
- On the Front Panel, create a Numeric Control.
- Change its data type representation to U16 (Unsigned 16-bit integer.)
- Change its label to “Number of Positions”.
- Create a second Numeric Control.
- Change its data type representation to DBL.
- Change its label to “End (um)”.
- Create a third Numeric Control
- Change its data type representation to DBL.
- Change its label to “Start (um)”.

5.1.1 *Create the Array Indicator*

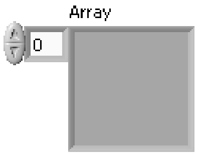
- Right click an empty area of the Front Panel to access the Controls palette.
- Expand the Controls Palette.



- Open the **Modern>>Array, Matrix & Cluster** sub-palette
- Select *Array*.
- Place the array on an empty area of the Front Panel

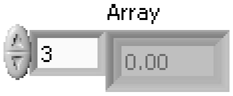


Arrays in LabVIEW



Arrays are used to link multiple controls/indicators together for simpler data manipulation. As an example; imagine a VI that calculates five points for "Y" along the line: Y=X+2, in equal "X" increments from: X = 1, to X = 5. Typically, to display the "Y" value of the five points would require five separate Numeric Indicators. An Array could be used to contain all five of the indicators, and link them to each other. Inside the Array, the indicators are identified not by name, but by their position within the array (i.e. first, second, third, etc…).

Cycling Through the Values Contained Within an Array

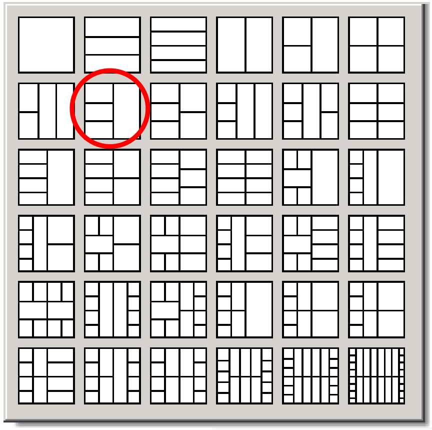


The many values stored within an Array can be individually accessed according to their position in the Array by cycling the control at the left of the position value (in the image at right, the value of position 3 of the array is being viewed.) Continuing our example from before, position "0" of the Array would contain the value "3" (Y=1+2), position "1" would contain the value "4" (Y=2+2), and so on.

- On the Front Panel, create a Numeric Indicator with data type: Double.

- Drag and drop the Numeric Indicator into the empty grey box of the Array Indicator. Each position within the Array now contains its own instance of a Numeric Indicator with data type: Double.

5.1.2 Create the Terminals of the VI



- From the Front Panel, edit the Connector Pane of your VI.

- Select the second pattern down, and to the right, from the top-left corner. In the image above, a red circle has been added to indicate which pattern to choose.

Note
You cannot Show Connectors from the Block Diagram. You will not have access to the options shown in the image at right unless you are on the Front Panel.

5.1.3 Set the input and output of the VI

- Left click the top input (left side) of your VI's Connector Pane. The top-left sixth of the Connector Pane should turn black.

- Left click the "Start (um)" control. The top-left sixth of your Connector Pane should turn orange (if it changes from black to a color other than orange, verify the "Start (um)" control is set to DBL.)

- Left click the second input of your VI's Connector Pane.

- Left click the "End (um)" control. The second input of your Connector Pane should turn orange.

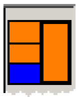
- Left click the bottom input of your VI's Connector Pane.

- Left click the "Number of Positions" control. The bottom input of your Connector Pane should turn blue.

- Left click the right (output) half of your VI's Connector Pane.

- Left click the "Array" indicator.

- Your VI's Connector Pane should now resemble the image at right:

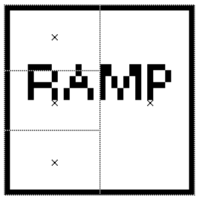


5.1.4 Edit the icon

- From the Front Panel, right click the Connector Pane icon.

- Select *Show Icon* from the menu that appears.

- Edit the icon of this VI to appear similar to the image at right:



5.1.5 Initialize an Array

You will now be instructed to create an Initialize Array function. The Initialize Array Function will create an array for your VI to fill with position values, incrementing from "Start (um)" to "Finish (um)".

Array of values
The finished Saw Scan VI will command the Nano-Drive® to each of many individual positions. Instead of calculating the next position before each motion command, it is simpler to pre-calculate all position values within a SubVI. We can then pass the array of values to another VI that handles the Nano-Drive® movement. An Array is simply a conventional way to store large quantities of values.

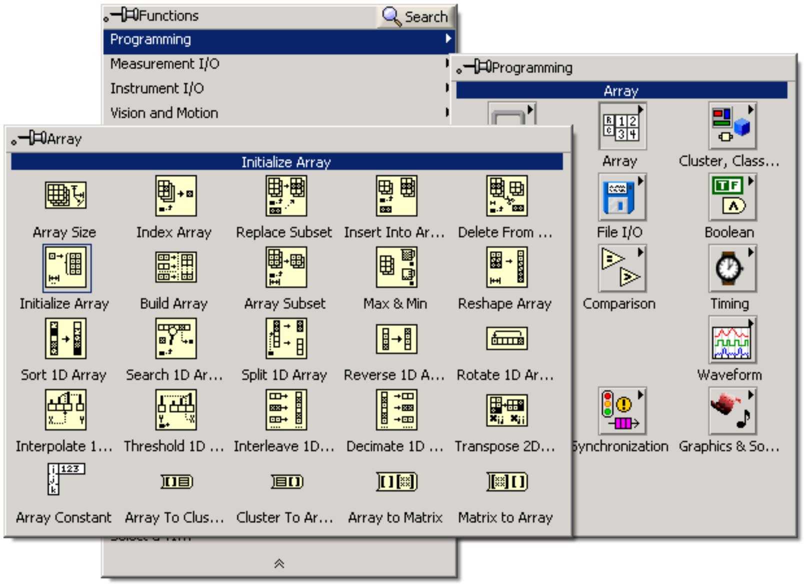
- The Front Panel is finished. Switch to the Block Diagram (hold *Ctrl* and press *E*.)

- Right Click an empty area of the Block Diagram to access the Functions Palette.

- Open the *Programming>>Array* category of the Functions Palette.

Note
In order to see the Programming sub-menu, you may have to left click the dashed down arrow to expand the Functions Palette.

- Left click the *Initialize Array* icon.



- Place the Initialize Array function over any empty area of the Block Diagram.

The Initialize Array Function

As mentioned before, the Initialize Array function creates an array for use within the Block Diagram. Using the Initialize Array function we can specify an initial value for each element of the array, as well as the total number of elements which will exist inside the array. The array is taken from the output (right) of this function.

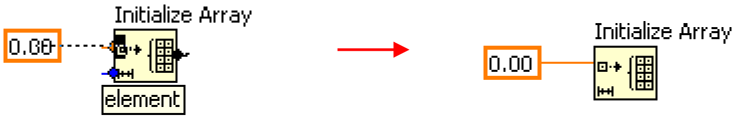
- Create a Numeric Constant with a value of “0”.

- Place the Numeric Constant slightly to the left of the Initialize Array function

- Change the Representation of the Numeric Constant to DBL.

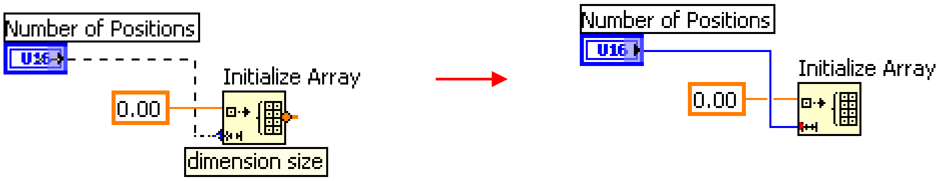
Note
The *Adapt to entered data* option will automatically uncheck after you specify a Data Type Representation for the Constant. In this case, this option will not work for us.

- Create a wire connecting the Numeric Constant to the Initialize Array function’s top input (labeled *element*.)



- Drag the Number of Positions Control above, and to the left, of the Initialize Array function.

- Create a wire connecting the Number of Positions Control to the Initialize Array function’s bottom input (labeled *dimension size*.)



5.1.6 Create the For Loop

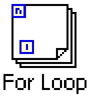
You will soon create a Loop structure. The concept of Loops in LabVIEW can be a difficult topic, especially if this is not similar to something you’ve dealt with in previous programming experiences. LabVIEW Help documentation would be an excellent resource to any who feel this tutorial inadequately covers this topic.

- Right click over any empty area of the Block Diagram to access the Functions palette.

- Open the *Programming>>Structures* category from the Functions palette.

- Left click the *For Loop* icon (seen below).

The For Loop



Similar to the Flat Sequence structure, the For Loop will contain components of your Block Diagram within a single frame, or *subdiagram*. Because of this, data must *tunnel* into, and out of, the For Loop.

The Loop Count Terminal

Every For Loop you create is automatically created with a Loop Count terminal in its top-left corner. The Loop Count terminal is used to specify a number of times to consecutively run the For Loop subdiagram. Wiring a numeric input, “n”, to the Loop Count terminal commands the For Loop to run “n” times.

- Left click an empty area of the Block Diagram to begin placing the For Loop.

- Drag your cursor diagonally a few inches and left click again to finish placing a For Loop. Any components that were overlapped by the For Loop during the creation process have been automatically integrated into the For Loop. Remove the overlapped components by dragging them outside of the For Loop.

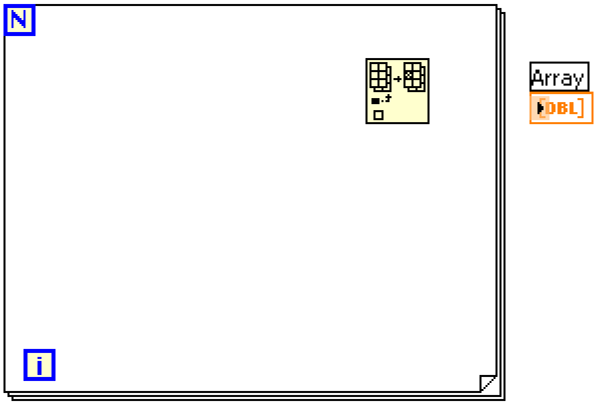
5.1.7 Create the Replace Array Subset Function

- Right Click an empty area of the Block Diagram to open the Functions Palette.

- Open the *Programming>>Array* category of the Functions Palette.

- Left click the *Replace Subset* icon.

- Place the Replace Array Subset function inside the For Loop.



5.1.8 Create the Mathematical Operators

- Right Click an empty area of the Block Diagram to open the Functions Palette.

- Open the Arith & Compar>>Numeric category from the Functions Palette.

- Left click the *Subtract* icon. The windows vanish, and you are left holding a Subtract function.

- Left Click an empty area of the Block Diagram at left of the For Loop to place the Subtract function.

- Follow the same process to locate and place the Divide function outside and to the left of the For Loop.

- Create the Multiply and Add functions and place them within the For Loop.

The Loop Iteration Terminal

Every For Loop you create is automatically created with a Loop Iteration terminal. The Loop Iteration terminal simply outputs a numeric value equal to the number of times the For Loop has repeated (outputs a value of "0" during the first run). Recall the picture from the beginning of Chapter 5 of the finished Block Diagram; the Loop Iteration terminal is used within the ramp-generating calculations.

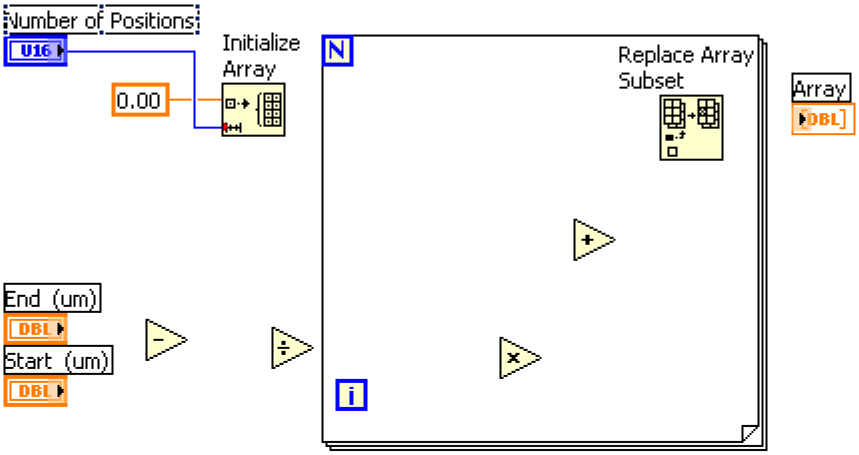
5.1.9 Arrange the Components in Preparation for Wiring

- Verify Your Block Diagram contains all components appearing in the image below.

- Drag each component of the Block Diagram to the approximate position relative to the For Loop, as seen below.

Note

Multiple components can be selected, and dragged, together at a one time. Left click and hold to create a selection box around the components. Include/remove components into the group by holding the Shift key while selecting.



5.1.10 Make the Wire Connections

- Connect the output of the "End (um)" control to the top input of the Subtract function.

- Connect the output of the "Start (um)" control to the bottom input of the Subtract function.

- Connect the output of the Subtract the function to the top input of the Divide function.

- Connect the output of the “Number of Positions” control to the bottom input of the Divide function.

- Connect the output of the Divide function to the top input of the Multiply function. This wire will have to *tunnel* into the For Loop

- Connect the Loop Iteration output to the bottom input of the Multiply function.

- Connect the output of the Multiply function to the bottom input of the Add function.

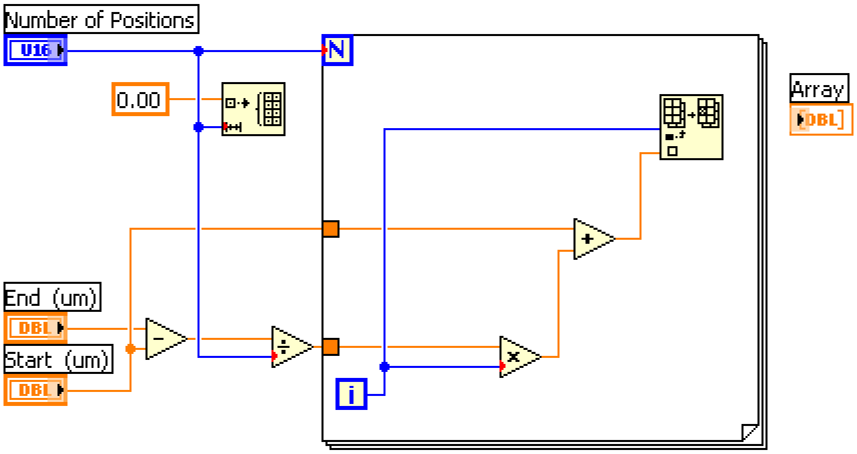
- Connect the output of the “Start (um)” control to the top input of the Add function.

- Connect the output of the Add function to the Replace Array Subset function’s bottom input, labeled *new element/subarray*.

- Connect the Loop Iteration Output to the Replace Array Subset function’s middle input, labeled “index”.

- Connect the output of the “Number of Positions” control to the Loop Count input.

- Verify that your wire connections are correct by comparing your Block Diagram to the image below:

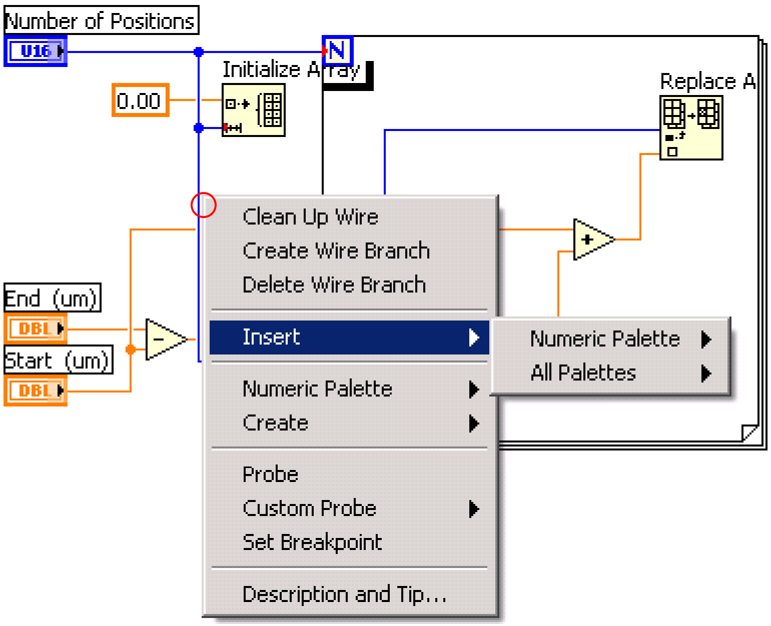


5.1.11 Add the Decrement Operator, and Both Data Type Converters

You must now add a *Decrement* Operator and Data Type Converter to the wire which connects the Number of Positions Control’s output to the bottom input of the Division function. However, the Decrement Operator and Data Type Converter you are about to add must not affect the Data passed from the Number of Positions Control to the Initialize Array function.

- Right click the wire (circled in red in the image below), which connects the "Number of Positions" output to the bottom input of the Division function. where it runs vertically toward the Division function, at a point where the wire (from this point) connects on one end to only the Division function.

- Select *Insert* from the shortcut menu that appears.



- Select *Numeric Palette*. The palette containing Numeric Operators opens.

- Left click the Decrement icon. The Decrement function has been automatically inserted into the wire.

- Right Click the same wire at the horizontal section between the Decrement Operator and the Division function.

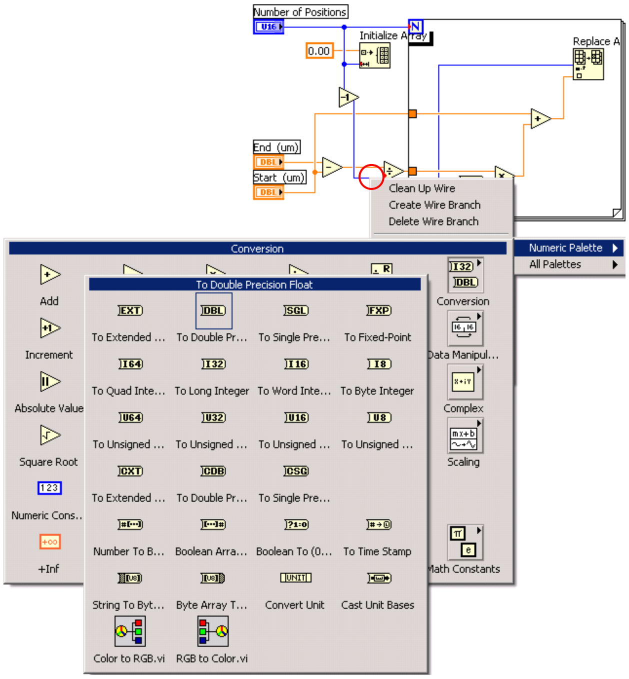
- Select Insert from the shortcut menu.

- Select the Numeric Palette.

- From the Numeric Palette; select Conversion. The Conversion Palette opens.

- Select *To Double Precision Float*. The windows vanish, and a Data Type Converter has been inserted into the wire.

- Right click the wire which passes data from the Loop Count output to the bottom input of the Multiplication function at a horizontal section.

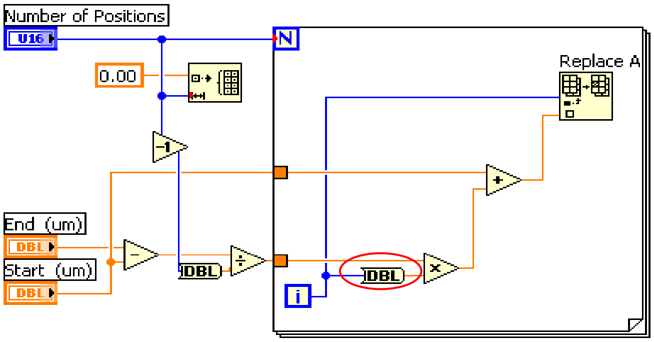


- Select Insert from the shortcut menu.

- Once again, select the Numeric palette.

- From the Numeric palette; open the Conversion palette.

- From the Conversion palette; select the To Double Precision Float icon. LabVIEW has automatically inserted the Data Type Converter into the wire for you.



5.1.12 Add, and connect, the Shift Registers for the For Loop

Your Ramp SubVI is nearly complete. The current VI is ready to generate the ramp values; however, it fails to store the values anywhere useful. You will now insert a shift register to retain the data through successive iterations of the For Loop.

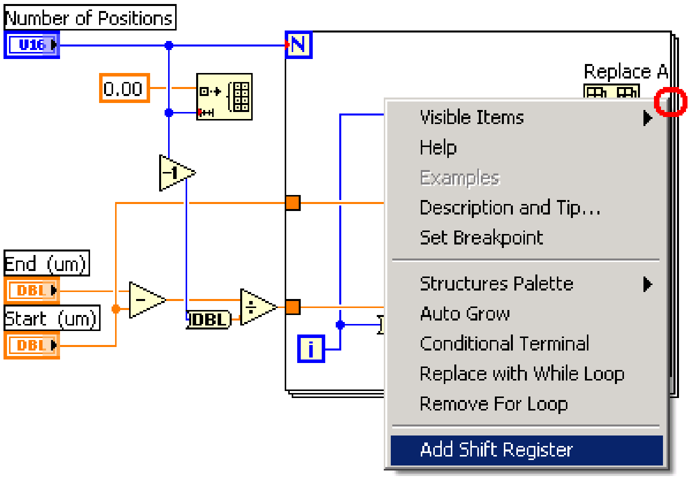
- Right click the right edge of the For Loop. A red circle has been added to the image below, right click at this point.

- Select *Add Shift Register* from the shortcut menu that appears. A Shift Register has been created on the right, and left, edge of the For Loop.

- Drag a Shift Register vertically to place it in line with the Replace Array Subset function. Notice the Shift Registers move together.

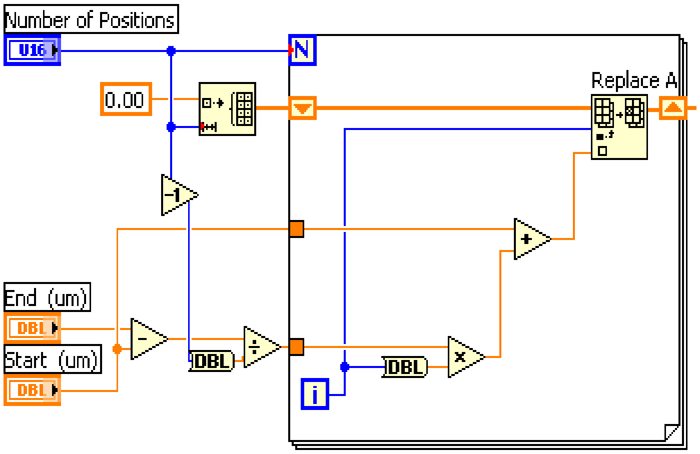
- Create a wire to connect the output of the Initialize Array function to the input of the left shift register.

- Create a wire to connect the output of the left Shift Register to the Replace Array Subset function's top input, labeled "array".

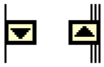


- Connect the output of the Replace Array Subset function to the input of the right Shift Register.

- Connect the output of right Shift Register to the input of the Array indicator

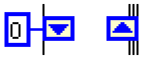


The Shift Register



The Shift Register consists of two separate pieces, together allowing the programmer to pass data from an iteration of a looping structure to the next. Upon completion of the For Loop the right-most Shift Register will output its most recent value. The value passed through Shift Registers can be of any type, including Array or Cluster.

Initializing a Shift Register



Before a Shift Register can be used, it must be initialized. At left can be seen a Shift Register being initialized with an I32. The Value at the output of the left-most Shift Register upon the first iteration of the For Loop would be “0”. In our Ramp SubVI, the Shift Register is initialized with an array of values; therefore, each iteration of the For Loop must not only specify new values for the Shift Register, but must also specify which element of the array will hold the value.

5.1.13 Test the Ramp VI

- Switch to the front Panel of your VI.

- Enter the number “10” into the “End (um)” control.

- Enter the number “101” into the “Number of Positions” control.

- Enter the number “5” into the “Start (um)” control.

- Run the VI.

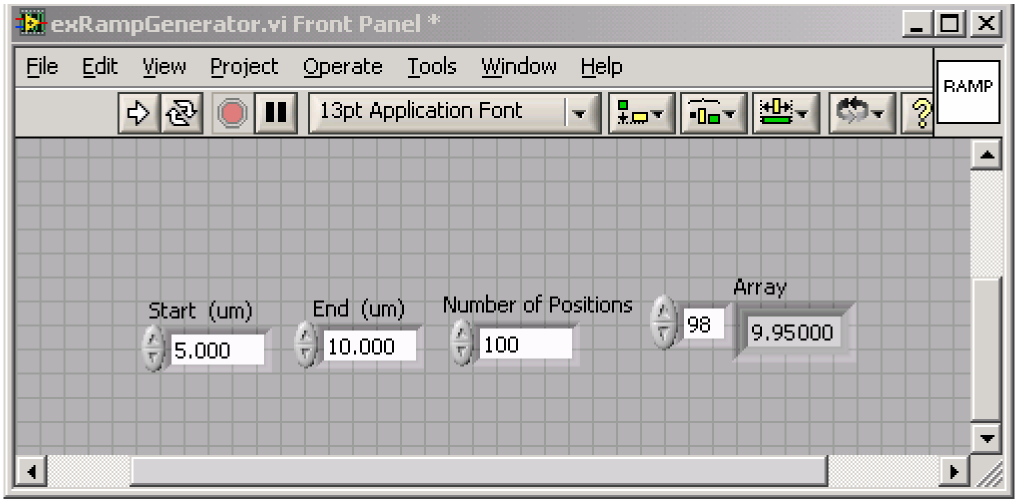
- Notice the first element (0) of the Array now holds the value 5.

Viewing values stored in an Array

From the Front Panel, use the arrow controls beside the Array indicator to view the values of other elements. The values should be incrementing toward the "End (um)" value, 10. The amount of the increments should be:

```
( End - Start ) / ( Number of Positions - 1)
```

This equates to increments of .05 for the test values you entered previously.


exRampGenerator.vi Front Panel *
File Edit View Project Operate Tools Window Help
13pt Application Font
RAMP

Start (um): 5.000

End (um): 10.000

Number of Positions: 100

Array: 98 | 9.95000

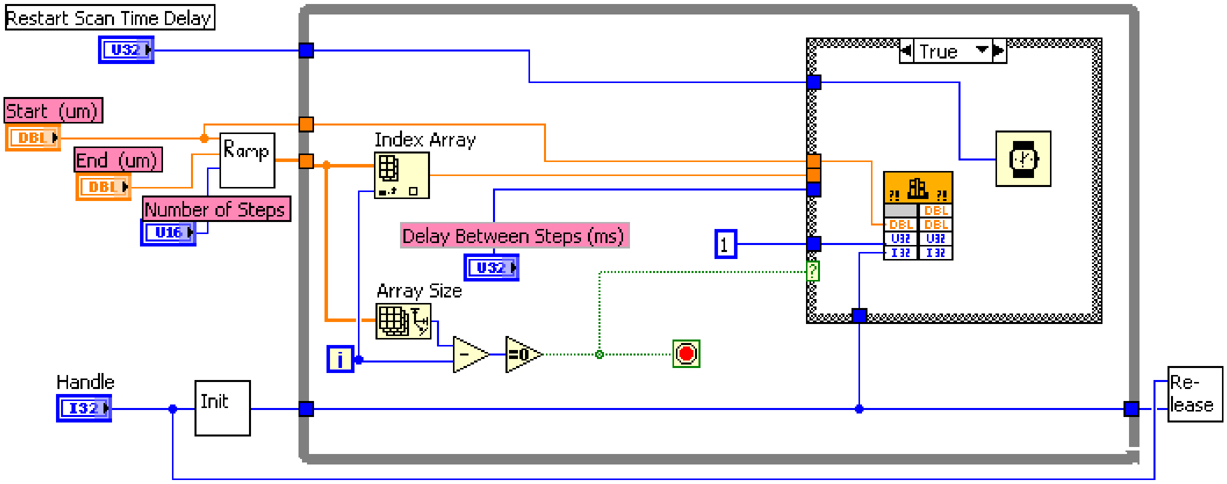
Section 5.2 – Create the Saw Scan VI

Project Goal

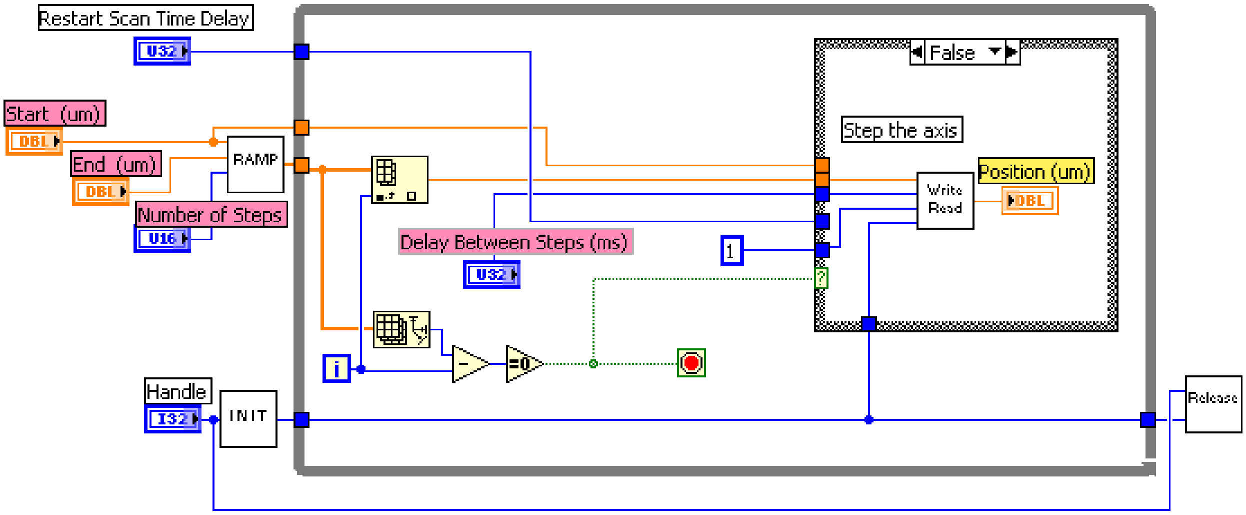
It is now time to create the Saw Scan VI which uses the array generated by the Ramp SubVI to command an axis to each of several positions. Seen below are two instances of the same VI, each depicting a unique state of the Case Structure. The Call Library Function Node seen below in the True case of the Case Structure calls for the function *MCL_SingleWriteN()*.

Note

The True case of the Case Structure happens along with a command to stop the while loop. Therefore, it can be easily observed that the True case is meant to run only once.



(Image of the finished Saw Scan Block Diagram in the True case)



(Image of the finished Saw Scan Block Diagram in the False case)

5.2.1 Start a New VI, Add the Front Panel Components

- Start a New VI, save it as: exSawScan.
- From the Front Panel, create six Numeric Controls.
- Name the controls: Start (um), End (um), Delay Between Steps (ms), Handle, Number of Steps, Restart Scan Time Delay.
- Change the Representation of the “Start (um)” control to DBL.
- Change the Representation of the “End (um)” control to DBL.
- Change the Representation of the “Delay Between Steps (ms)” control to U32.

- Change the Representation of the “Handle” control to I32.

- Change the Representation of the “Number of Steps” control to U16.

- Change the Representation of the “Restart Scan Time Delay” control to U32.

- Edit the Display Format of the “Start (um)”, and “End (um)”, control to display as a Floating Point with 4 digits of precision.

5.2.2 Create and Configure the Graph

- Right click an empty area of the Front Panel. The Controls Palette appears.

- From the Express Palette of the Functions Menu; hover your cursor over the *Graph Indicator* Icon. The *Graph Indicators* menu appears.

- Left Click the *Waveform Chart* icon. The windows vanish, and you are left holding the footprint of a Waveform Chart.

- Left Click an empty area of your Front Panel to place the Waveform Chart.

- Right click an edge of the Waveform Chart, and select P*roperties* from the shortcut menu. The *Chart Properties* window appears.

- From the Appearance tab, change the Label to “Position (um)”.

- Left click the *Scales* tab.

- Change the *Name* of the X-Axis to “Step #”.

- Change the *Maximum* of the X-Axis to “100”.

- Left click the pull-down menu at the top of the Scales tab.

- Select “Amplitude (Y-Axis)” from the list.

- Change the *Name* of the Y-Axis to Position (um).

- Left click the *Autoscale* checkbox to remove the check mark from it. The Minimum and Maximum range of the Chart should now be editable.

- Change the Minimum of the Y-Axis to “0”.

- Change the Maximum of the Y-Axis to “100”.

- Left click OK to accept the changes and exit the Chart Properties menu.

5.2.3 *Create and Configure the While Loop*

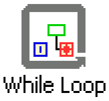
- From the Block Diagram, Right click an empty area to access the Functions palette.

- From the Functions palette, open the Express>>Execution Control category.

- Select *While Loop*.

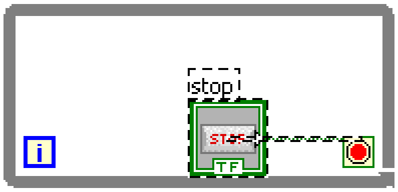
- Place the While Loop on the Block Diagram. The While Loop is placed in the same manner as the Flat Sequence/Case Structure.

The While Loop



The While Loop and For Loop are very similar in functionality. The main difference being the For Loop repeats a specific number of times; whereas, the While Loop continues to repeat its subdiagram until commanded to stop. Like the For Loop, the While Loop possesses a Loop Iteration terminal; however, the Loop Count terminal is replaced by the Loop Condition terminal.

- If a Boolean Control, labeled “stop”, was automatically created with the While Loop; delete it now. Delete the wire that was connecting it to the Loop Condition input.



- Drag all other Block Diagram Components to the left side of the While loop.

- Resize the While Loop. Make it large enough to fit all the components that will later be inside it (as seen in the image below, or at the beginning of this chapter.)

5.2.4 *Create and Configure the Case Structure*

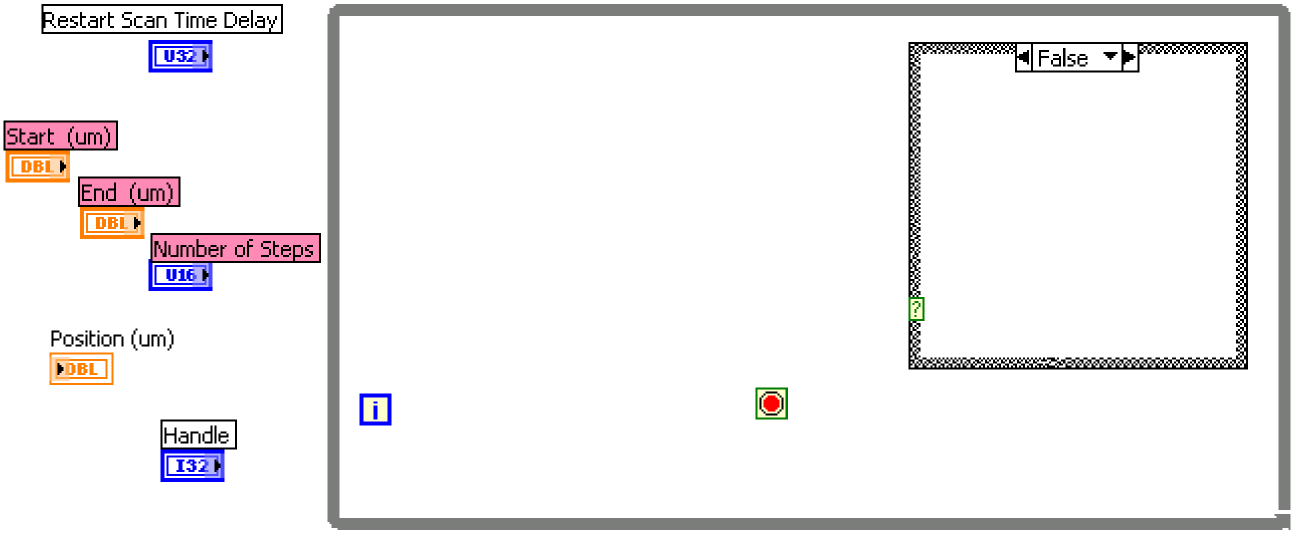
- Right click an empty area to access the Functions palette.

- From the Functions palette, open the Execution Control palette.

- Select Case Structure from the Execution Control palette.

- Create the Case Structure inside the While Loop with a size of approximately one quarter the size of your While Loop.

- Drag the Case Structure to the upper-right corner of your While Loop.



5.2.5 Create the Function Call for MCL_SingleWriteN

- Create a Call Library Function Node in the True Case of your Case Structure.

- Enter the Configure menu of the Call Library Function Node.

- Under the Function tab, change the Library name or path to the address of the MADlib.dll

- From the *Function name* pull-down menu, select MCL_SingleWriteN.

- Switch to the Parameters tab.

- Add three new parameters.

- Left click the first of the new parameters, arg1, and configure the *Current parameter* area to the following: change the Name to “position”, Type to “Numeric”, Data type to “8-byte Double”, and Pass to “Value”.

- Left click the second of the new parameters, arg2, and configure the Current parameter area to the following: change the Name to “axis”, Type to “Numeric”, Data type to “Unsigned 32-bit Integer”, and Pass to “Value”.

- Left click the third of the new parameters, arg3, and configure the Current parameter area to the following: change the Name to “handle”, Type to “Numeric”, Data type to “Signed 32-bit Integer”, and Pass to “Value”.

- Compare your Function Prototype to the following:

```
void MCL_SingleWriteN(double position, uint32_t axis, int32_t handle);
```

- Left click OK to accept the changes and exit the Call Library Function configuration menu.

- Create a Wait (ms) timer inside the True Case of the Case Structure at the right of the MCL_SingleWriteN function call.

5.2.7 Add the Init, Release, and the Ramp SubVIs

- Right click an empty area of the Block Diagram. The Functions palette appears.

- Left click the *Select a VI* menu item. A new window appears with the title: *Select a Vi to Open*

- Use the Select a VI to Open window to locate your StandardInitSubVI.vi.

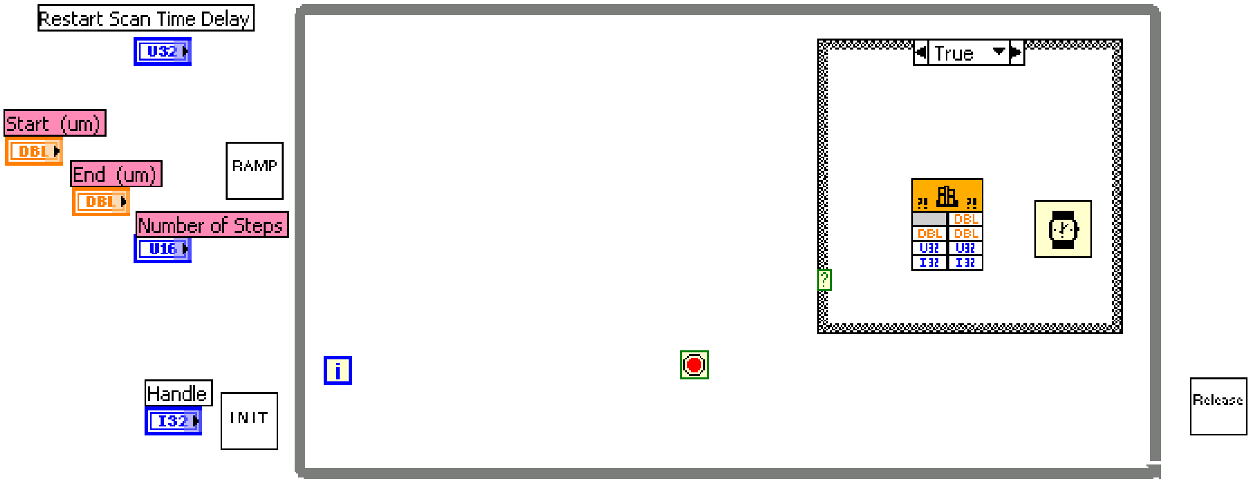
- Left click your StandardInitSubVI.vi; it should appear in the *File name* field near the bottom of the window.

- Left click the OK which appears at the bottom-right of the window. The window vanishes, and you are left holding the icon of your StandardInitSubVI.vi

- Left click an empty area left of the While Loop to place the StandardInitSubVI.vi

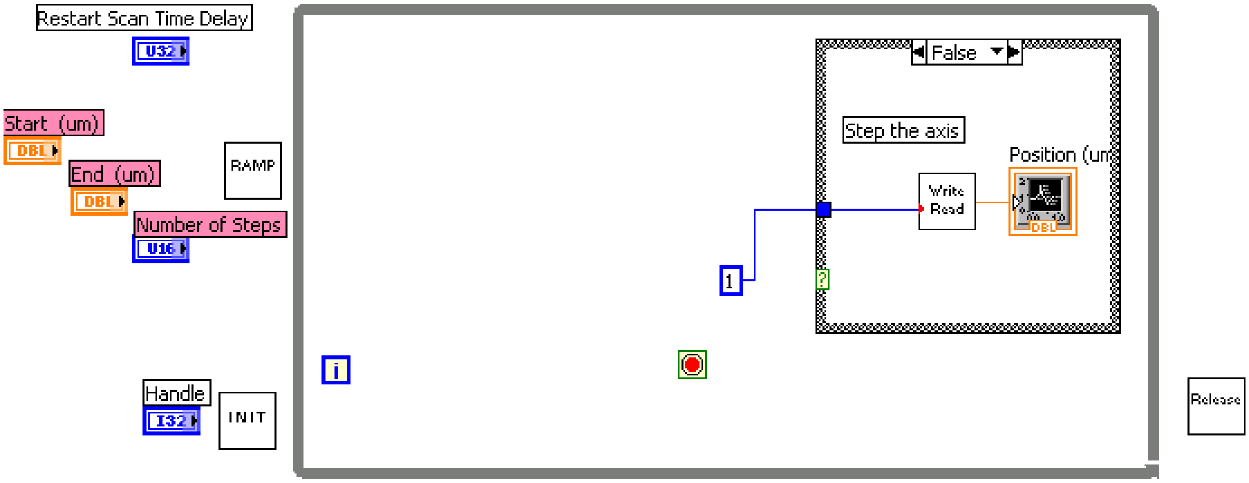
- Follow the same process to locate and place the StandardReleaseSubVI.vi to the right of the While Loop.

- Follow the same process to place the exRampGenerator.vi SubVI outside, and to the left, of the While Loop.



5.2.8 Place, and Prepare, the Write\Read SubVI

- Switch to the False case of your Case Structure.
- Right click to open the Functions palette.
- Choose the *Select a VI…* item to browse your Hard Drive for user created VIs.
- Locate, and open, the Write/Read SubVI (exWriteReadN.vi) which you created previously.
- Place the Write/Read SubVI into the false case of the Case Structure.
- Drag the “Position (um)” Waveform Chart inside the False case of the Case Structure and place it to the right of the Write/Read SubVI.
- Create a wire to connect the output of the Write/Read SubVI to the input of the “Position (um)” Waveform Chart.
- Create a Numeric Constant with the value “1” inside the while loop. Place it outside and slightly to the left of the Case Structure.
- Create a wire to connect the Numeric Constant to the “axis” input of the Write/Read SubVI. This wire will have to *tunnel* through the left side of the Case Structure.



5.2.9 *Create the Array functions, and Arithmetic & Comparison Operators*

- Select the *Index Array* function from the Array palette (the Array Palette can be found in the Programming category of the Functions palette.)

- Place the Index Array function inside the While Loop near the middle of the left edge.

- Select the *Array Size* function from the Array palette.

- Place the Array Size function below the Index Array function.

- Select a Subtract function from the Numeric palette (found in the Arithmetic & Comparison category of the Functions palette).

- Place the Subtract function slightly below and to the right of the Array Size function.

- Select the *Equal to 0?* Function from the *Programming>>Comparison* category of the Functions palette (also found in the *Express>>Arithmetic & Comparison>>Comparison* category of the Functions palette).

- Place the Equal to 0? function slightly to the right of the Subtract function. LabVIEW may automatically create a wire connecting the output of the Subtract function to the input of the Equal to 0? function; this wire is perfectly acceptable but not necessary since you will make this wire connection later.

The Loop Condition Terminal

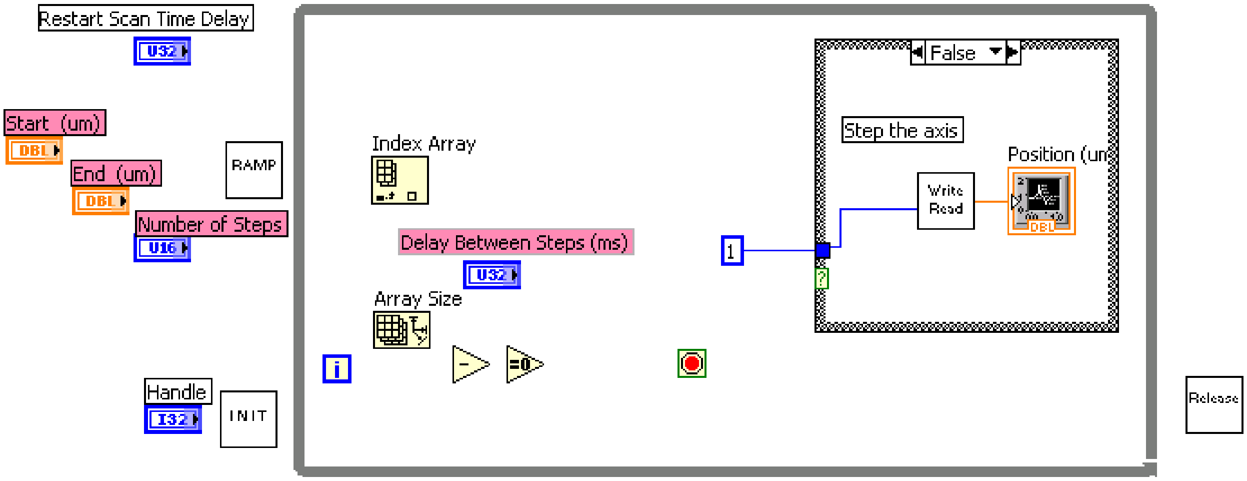
Every While Loop you create is automatically created with a Loop Condition terminal. The only way to stop a While Loop is to pass a value of "True" to the Loop Condition terminal. How can we tell when to stop our SawScan VI's While Loop? The answer is simple: the value of the Loop Iteration terminal will equal the size of the Ramp SubVI's array only after each position command has been withdrawn from the array; therefore, the while loop should stop.

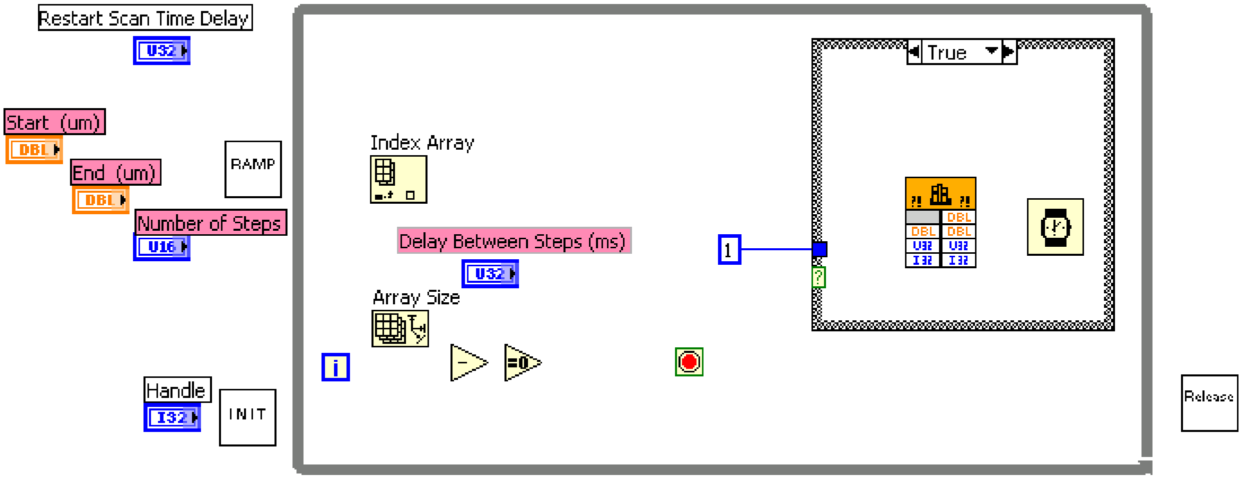
5.2.10 Arrange the Components in Preparation for Wiring

Your VI should now contain all the components seen in the images below. If this is not the case, browse the previous set of instructions for the step(s) you've overlooked or try creating the component(s) according to the images.

- Arrange the components of your Block Diagram to closely match the positioning of the components as seen in the images below (because the VI contains a Case Structure, two images are shown below, each depicting a different possible state of the Case Structure.)

Note
To view the label of a function (such as the array functions below): right click the function, and select *visible items>>label*.





- Verify that the Data Type Representations of all your Numeric Controls are correct before continuing.

5.2.11 Make the Wire Connections (False Case of the Case Structure)

- Switch to the False case of the Case Structure.
- Create a wire to connect the “Start (um)” control to the Ramp SubVI’s top input, labeled “start (um)”.
- Create a wire to connect the “End (um)” control to the Ramp SubVI’s middle input, labeled “end (um)”.
- Create a wire to connect the “Number of Steps” control to the Ramp SubVI’s bottom input, labeled “Number of Positions”.
- Connect the output of the Ramp SubVI to the Index Array function’s top input, labeled “array”. The wire will tunnel into the While Loop to make this connection.
- Connect the output of the Ramp SubVI to the input of the Array Size function. Either tunnel into the While Loop to make the connection or start the wire at a point on the existing Ramp SubVI output wire which is already inside the While Loop.
- Connect the output of the Loop Iteration counter to the Index Array function’s bottom input, labeled “index”.
- Connect the output of the Loop Iteration counter to the bottom input of the Subtract function.
- Connect the output of the Array Size function to the top input of the Subtract function.
- Connect the output of the Subtract function to the input of the Equal to Zero? comparison operator.
- Connect the output of the Equal to Zero? comparison operator to the input of the Loop Condition terminal.
- Connect the output of the Equal to Zero? comparison operator to the input of the Case Selector of the Case Structure.
- Connect the output of the “Handle” control to the input of the Init SubVI.
- Connect the output of the “Handle” control to the Release SubVI’s top input, labeled “Incoming Handle”. Do not tunnel through the While Loop to make this connection.

- Connect the output of the Init SubVI to the bottom input of the Release SubVI, labeled “Outgoing Handle”. This wire must tunnel through into the left edge and out the right edge of the While Loop.

- Connect the output of the Init SubVI to the Write/Read SubVI’s bottom input, labeled “Handle”. Either tunnel into the While Loop to make this connection or start the wire at a point on the existing Init SubVI output wire which is already inside the While Loop.

- Connect the output of the “Delay Between Steps (ms)” control to the “Delay (ms)” input of the Write/Read SubVI. The wire must tunnel through the left edge of the Case Structure to make this connection.

- Connect the output of the Index Array function to the Write Read SubVI’s top input, labeled “Move to (um)”.

5.2.12 Make the Wire Connections (True Case of the Case Structure)

- Switch to the True case of the Case Structure.

- Create a wire to connect the “Start (um)” control to the MCL_SingleWriteN function’s top input, labeled “position”. The wire will not only tunnel through the left edge of the While Loop, but also through the left edge of the Case Structure.

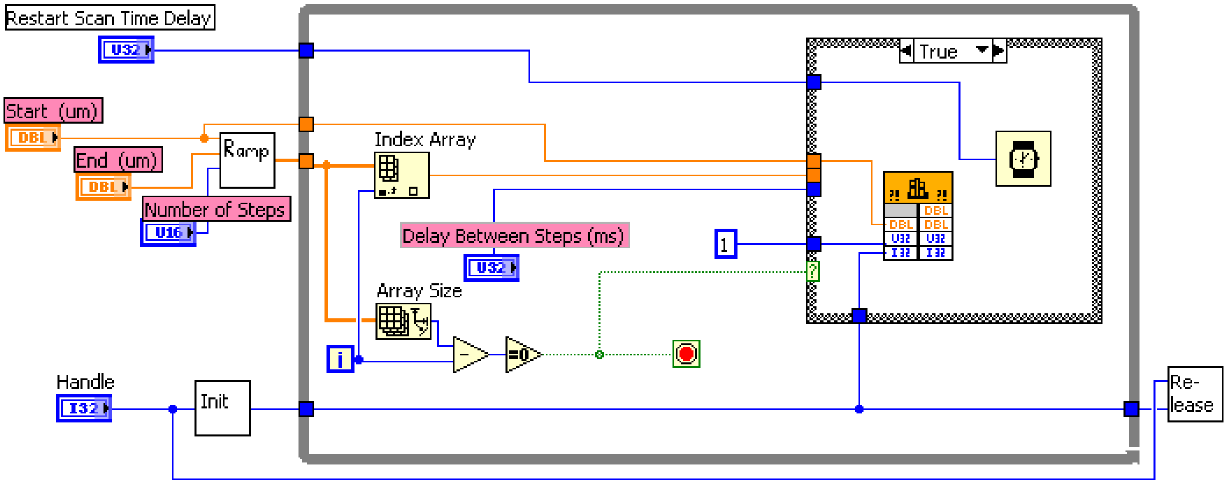
- Create a wire connecting the output of the Numeric Constant to the MCL_SingleWriteN function’s middle input, labeled “axis”. The Numeric Constant should already have a tunnel into the False case of the Case Structure. The output of this tunnel can also be used within the True case.

- Create a wire connecting the Init SubVI’s output to the MCL_SingleWriteN function’s bottom input, labeled ‘handle”. Once again, a tunnel should already exist to pass data from the Init SubVI into the Case Structure. Using this tunnel would be an alternative to creating a second tunnel.

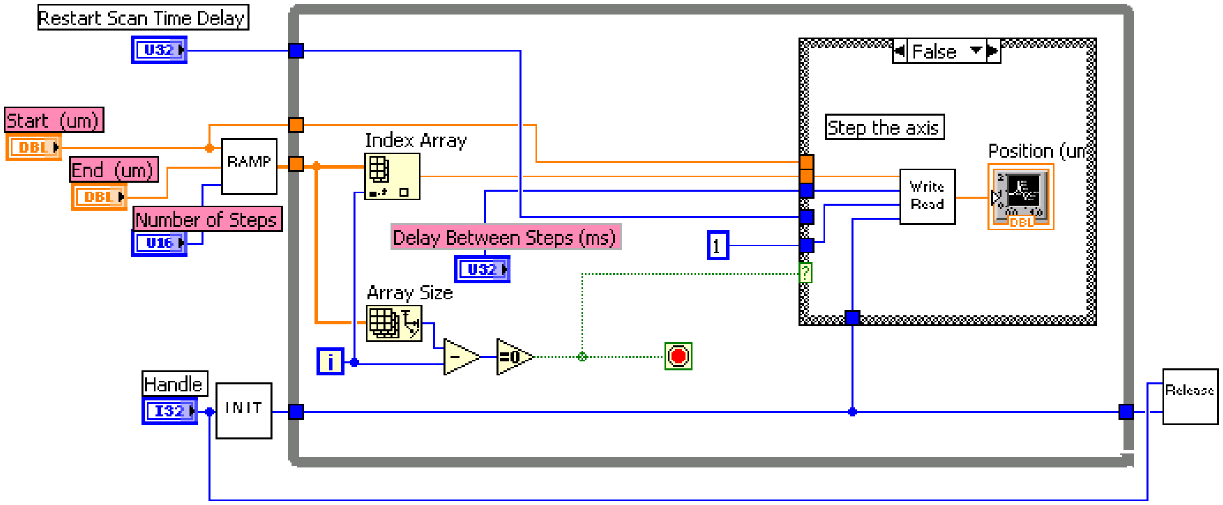
- Create a wire to connect the output of the “Restart Scan Time Delay” control to the input of the Wait (ms) timer. The wire will tunnel into the While loop, then into the Case Structure, to make this connection.

Section 5.3 – Prepare the VI to Scan an Axis

Project Goal
The Saw Scan VI is complete. Compare your Block Diagram to the images below before proceeding. If necessary, change the value of the Numeric Constant to select a different axis (X = 1, Y = 2, Z = 3). You will now enter valid default parameters for the VI, and will soon run the VI.



(Image of the finished Block Diagram for the Saw Scan VI in the True case)



(Image of the finished Block Diagram for the Saw Scan VI in the False case)

5.3.1 Set Default Values for the Front Panel Components

- Switch to the Front Panel of your exSawScan.vi.

- Enter the Value “100” into the “End (um)” control.

- Right click the “End (um)” control to access its shortcut menu.

- Select the Data Operations category from the “End (um)” control’s shortcut menu.

- Select *Make Current Value Default* from the list of Data Operations items.

- Enter the value “100” into the “Number of Steps” control and follow the same process as above to make this value the default value for this control.

- Enter the value “10” into the “Delay Between Steps (ms)” control and make this value the default for this control.

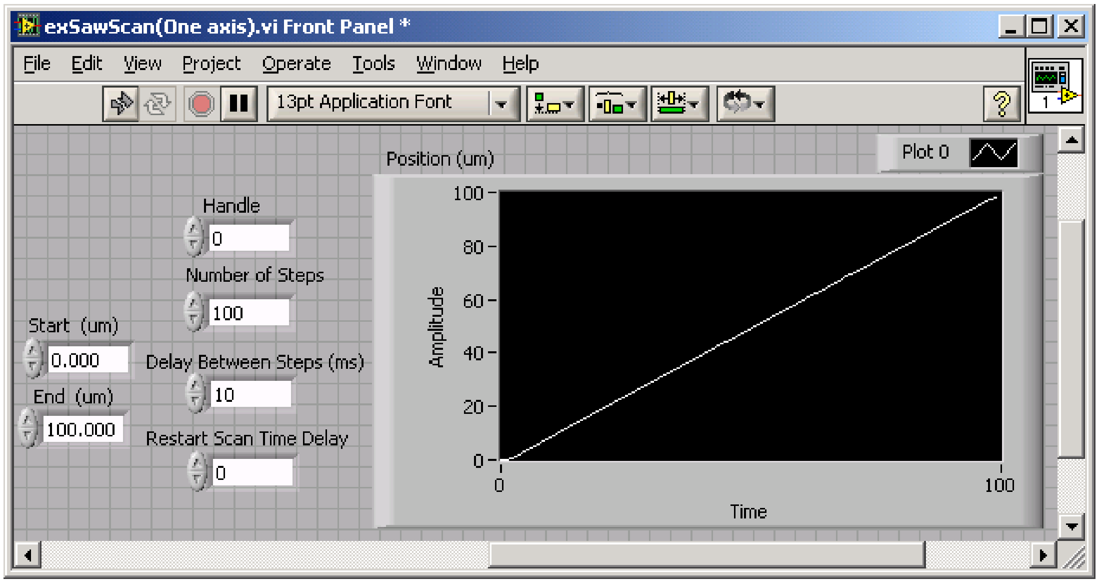
- All other controls can remain at their initial default value, “0”.

5.3.2 Run the VI

- Run the VI. The “Position (um)” Waveform Chart will be filled with values which should form the image of a ramp.

Note
The Stage may not be at position “0” when this VI begins. If this is the case, the charted position data will begin with a curve slanting toward the ramp. This is the response of the stage as it attempts to align with the string of commands.

- The VI is finished. Save this VI before proceeding with this tutorial.


exSawScan(One axis).vi Front Panel *

Handle: 0
Number of Steps: 100
Start (um): 0.000
End (um): 100.000
Delay Between Steps (ms): 10
Restart Scan Time Delay: 0

Chapter 6: Waveform Acquisition

Sometimes, it is desirable to exceed the Windows latency (1millesecond) when moving a nanopositioners. For example you may know in advance the scanning parameters for the axis you wish to scan, but you may wish to update the position faster than every millisecond, or you may not wish to send each data point sequentially over the USB interface. Alternatively, you may wish to have well defined timing between each position which LabVIEW cannot achieve in Windows. In these cases, the solution is waveform acquisition. The VI shown below will be created in Chapter 6. The VI will teach you how to download an array of predefined axis positions, and read back the resulting positions.

.

Goal of Chapter 6

Up to this point, we have refrained from reading/writing more than one position at a time. In Chapter 6, you will create a VI which will read/write hundreds of points at a time. Accomplishing this requires the use of a new set of MADlib.dll functions, as well as a slight increase in the complexity of the VIs we create.

Topics Include:

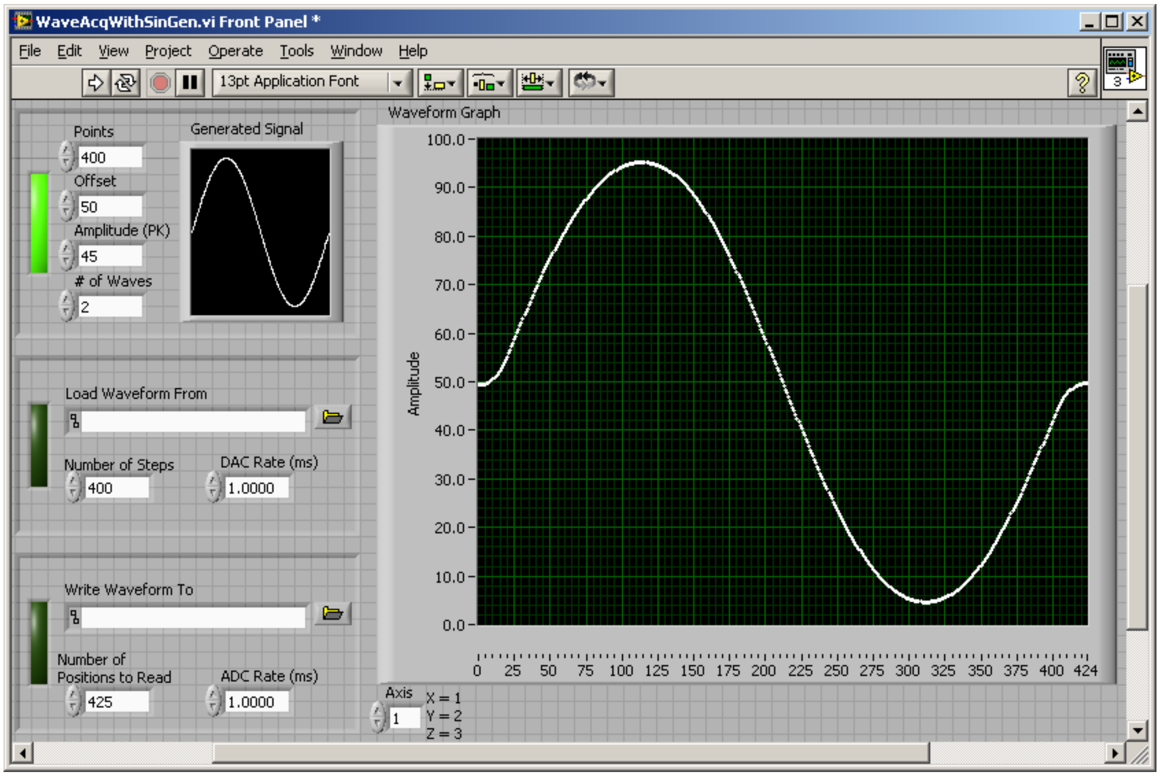
1. Writing data to text files.
2. Reading data from text files.
3. Introduction to Trigonometric functions.
4. Waveform Acquisition.

MADlib.dll Functions Used Within This Chapter Include:

1. void MCL_ReleaseHandle(int handle);
2. int MCL_InitHandle();
3. int MCL_Setup_LoadWaveFormN(unsigned int axis, unsigned int DataPoints, double milliseconds, double waveform, int handle);
4. int MCL_Setup_ReadWaveFormN(unsigned int axis, unsigned int DataPoints, double milliseconds, int handle);
5. int MCL_TriggerWaveformAcquisition(unsigned int axis, unsigned int DataPoints, double waveform, int handle);

SubVIs Used Within This Chapter Include:

1. StandardInit.vi / StandardInitSubVI.vi
2. exSineGenerator *(new)*



WaveAcqWithSinGen.vi Front Panel *

File Edit View Project Operate Tools Window Help

13pt Application Font

Points: 400

Offset: 50

Amplitude (PK): 45

# of Waves: 2

Load Waveform From:

Number of Steps: 400

DAC Rate (ms): 1.0000

Write Waveform To:

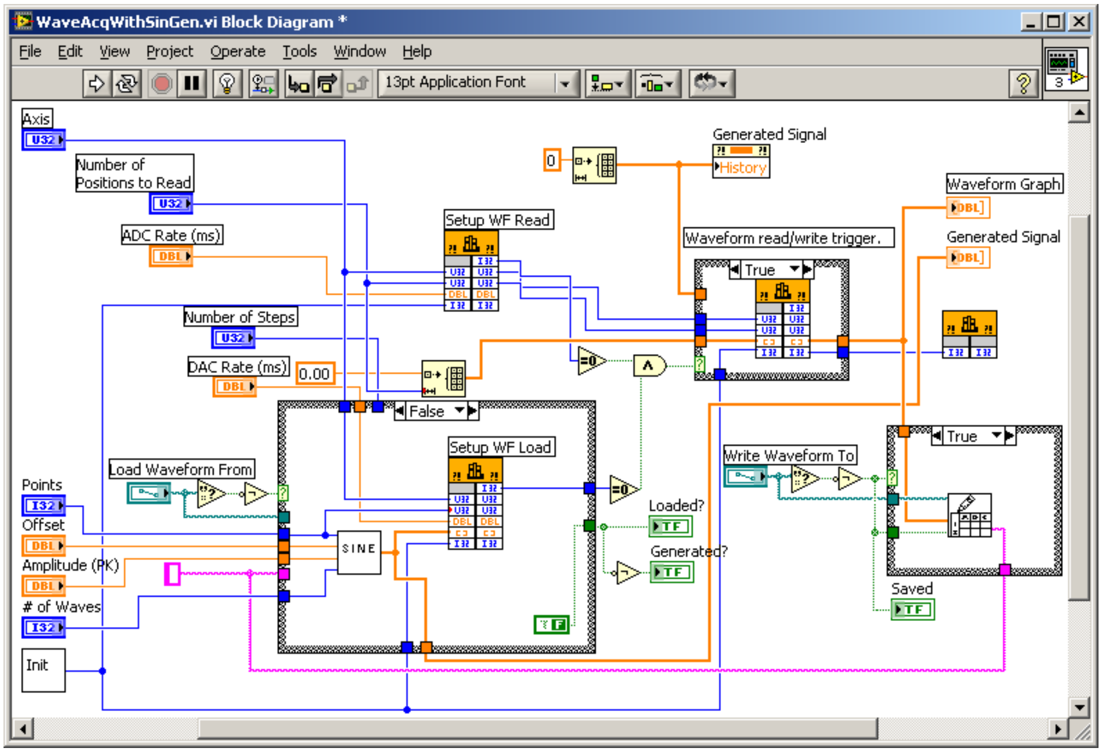
Number of Positions to Read: 425

ADC Rate (ms): 1.0000

Axis: 1

X = 1
Y = 2
Z = 3

(Image of the finished Front Panel by the end of Chapter 6)

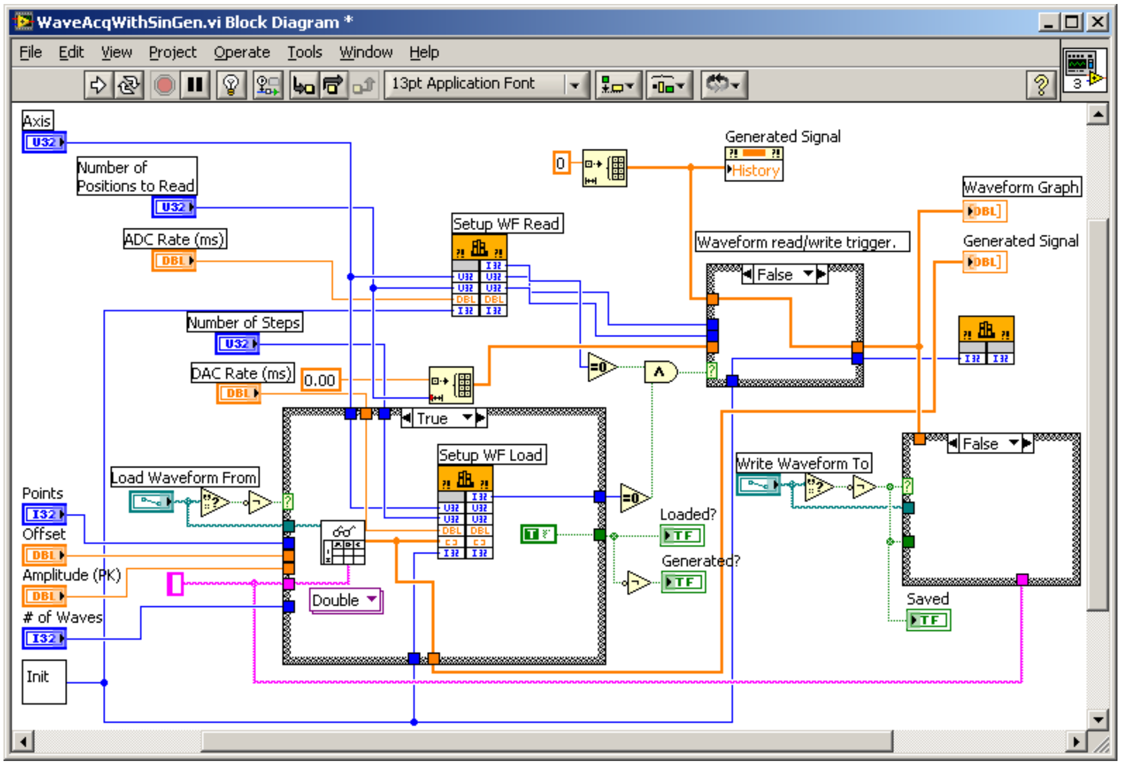


WaveAcqWithSinGen.vi Block Diagram *

File Edit View Project Operate Tools Window Help

13pt Application Font

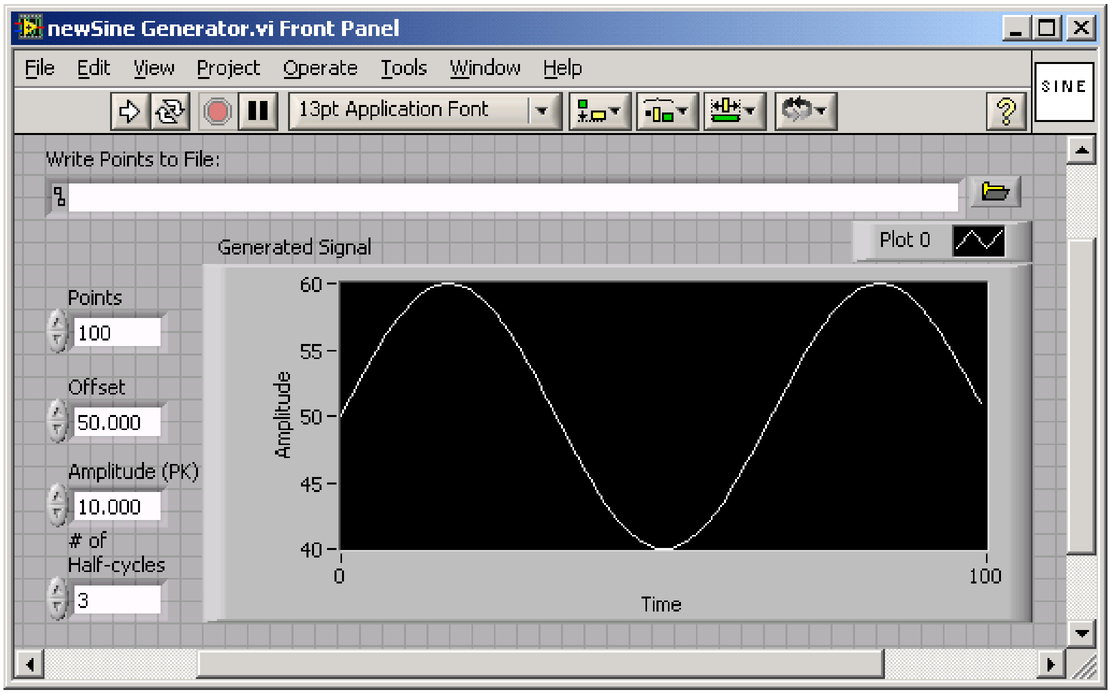
(Image 1 of 2 showing the finished Block Diagram by the end of Chapter 6)



(Image 2 of 2 showing the finished Block Diagram by the end of Chapter 6)

Section 6.1 – Create the Sine Wave Generating SubVI

The VI shown below will be created in Section 1 of Chapter 6.



Write Points to File:

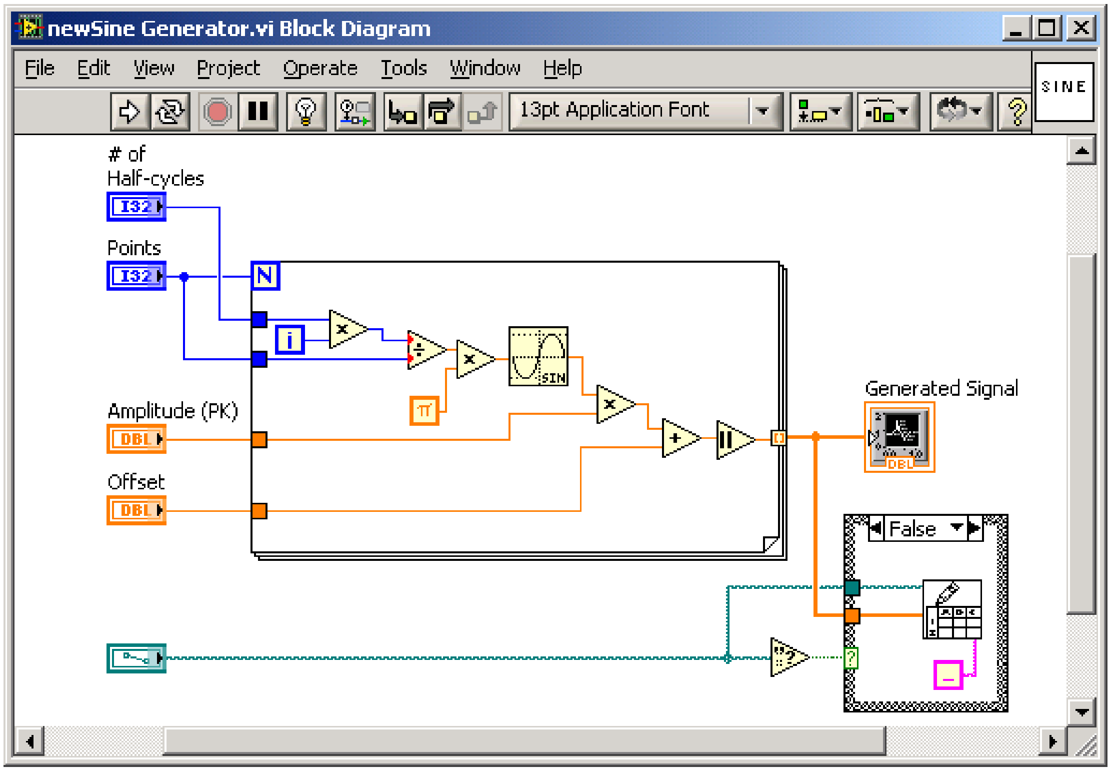
Points 100

Offset 50.000

Amplitude (PK) 10.000

# of Half-cycles 3

(Image of the finished Sine Wave Generator Front Panel)



(Image of the finished Sine Wave Generator Block Diagram)

6.1.1 Create the Four Numeric Controls, and Single Waveform Chart

- Begin LabVIEW and start a new VI by selecting *Blank VI* from the *Getting Started* Window. The Front panel and Block Diagram of a new VI appear.

- Save the new VI as: exSineGenerator

- On the Front Panel, create four Numeric Controls.

- Name the controls: # of Half-cycles, Points, Amplitude (PK), Offset.

- Make the "# of Half-cycles" data type representation I32 (Signed 32-bit integer.)

- Make the "Points" data type representation I32.

- Make the "Amplitude (PK)" data type representation DBL.

- Make the "Offset" data type representation DBL.

- Create a Waveform Chart.

- Change its label to “Generated Signal”.

6.1.2 Create the File Path Control

- Right Click an empty area of the Front Panel to access the Controls palette.
- Open the *Express>>Text Ctrls* category of the Functions palette
- Select the File Path Control from the Text Ctrls category. The windows vanish, and you should now be holding the footprint for a File Path Control.
- Left click an empty area of the Front Panel to place the File Path Control.

File Path Control

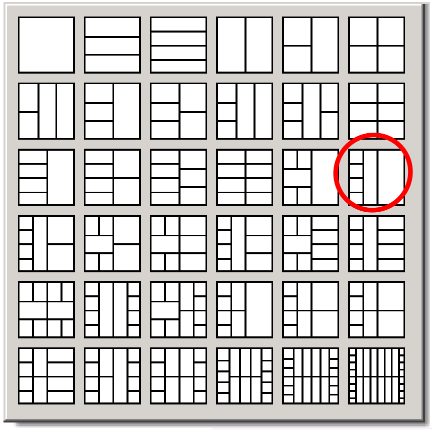
Eventually you will be programming this VI to generate a number of position values, each representing a piece of a sine wave. This array of values can than be saved onto your Hard drive, into a location specified by the File Path Control.

- Change the File Path Control’s label to “Write Data Points to File:”

Note
Like most other components, the File Path Control appears differently on the Block Diagram depending on whether the property, *View As Icon*, is checked.

6.1.3 Create the Terminals of the VI

- From the Front Panel, right Click the icon of the VI.
- Select *Show Connector* from the menu that appears. The Icon should have switched from the graphic of an oscilloscope to the Connector Pane.
- Right click the Connector Pane Icon .
- Hover the cursor over the *Patterns* item of the menu that appears.



- Select the third pattern down, and sixth to the right. In the image above, a red circle has been added to indicate which pattern to choose.

6.1.4 Set the input and output of the VI

With this particular pattern, your VI has five useable inputs.

- Left click the top-left input of your VI's Connector Pane. The top-left of the Connector Pane should turn black.

- Left click the "Points" control. The first input of your Connector Pane should turn blue (if it changes from black to a color other than blue, verify the "Points" control is set as a DBL.)

- Left Click the second left-side input of your VI's Connector Pane.

- Left Click the "Offset control". The second input of your Connector Pane should turn orange.

- Left Click the next highest left-side input of your VI's Connector Pane.

- Left Click the "Amplitude (PK)" control. The lower middle-left sixteenth of your Connector Pane should turn orange. This portion of the Connector Pane will now pass data to the "Amplitude (PK)" control.

- Left Click the bottom left-side input of your VI's Connector Pane.

- Left Click the "# of Half-cycles" control. The bottom left input of your Connector Pane should turn blue.

- Left click the middle input which spans the entire vertical area of the Connector Pane.

- Left click the File Path Control. The middle input of your Connector Pane should turn teal.

- Left Click the right (output) half of your VI's Connector Pane.

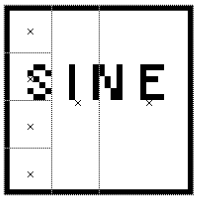
- Left Click the Waveform Graph.

6.1.5 Edit the icon

- Right click the Connector Pane icon.

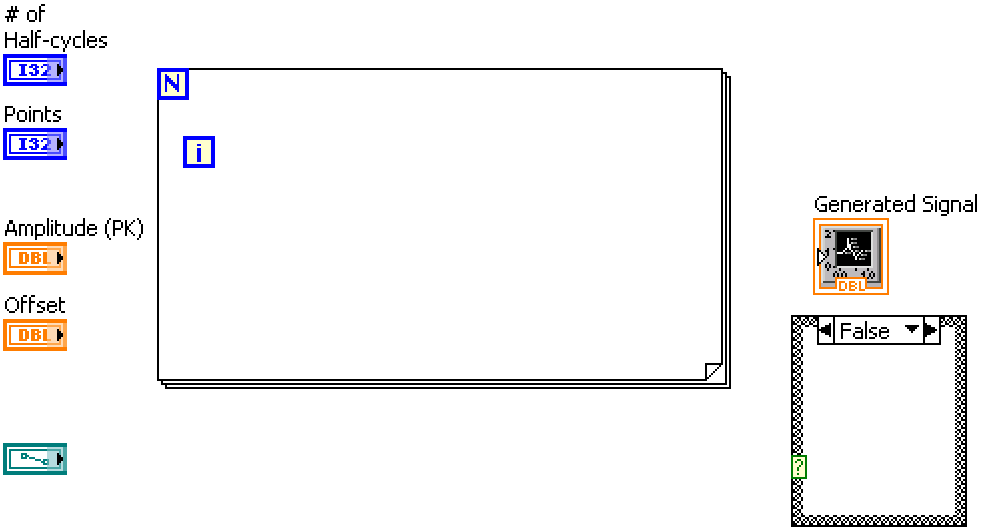
- Select *Show Icon* from the menu that appears.

- Edit the icon to resemble the image at right.



6.1.6 Create the For Loop and Case Structure

- Switch to the Block Diagram of your VI.
- Create a For Loop in an empty area of your Block Diagram.
- Create a Case Structure at the right of the For Loop.
- Move and resize the For Loop and Case Structure to match the image below.

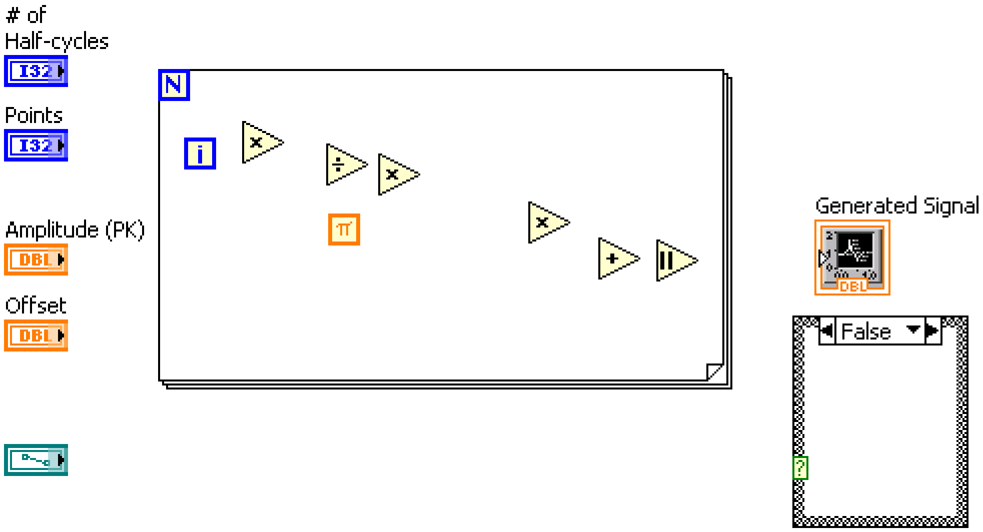


6.1.8 Create the Mathematical Operators

- Right Click an empty area of the Block Diagram to open the Functions Palette.
- Open the Express>>Arithmetic & Comparison>>Numeric category of the Functions Palette.
- Left click the *Absolute Value* icon. The windows vanish, and you are left holding an Absolute Value function.
- Left Click an empty area of the Block Diagram within the For Loop to place the Absolute Value function.
- Follow the same process to locate, and place: a Divide function, 3 Multiply functions, and an Add function. Place all of these within the For Loop.

6.1.9 Create the Math Constant, Pi

- Right Click an empty area of the Block Diagram to open the Functions Palette.
- Open the Express>>Arithmetic & Comparison>>Numeric category of the Functions Palette.
- Open the Math Constants category
- Left click the Pi icon. The windows vanish, and you are left holding a Pi constant.
- Left click an empty area inside the For Loop to place the Pi constant.



6.1.10 Create the Sine Trigonometric Function.

- Right Click an empty area of the Block Diagram to open the Functions Palette.
- Open the Express>>Arithmetic & Comparison>>Math category of the Functions Palette.
- Open the Trigonometric category.
- Left click the Sine icon. The windows vanish, and you are left holding a Sine function.
- Left click an empty area inside the For Loop to place the Sine function.

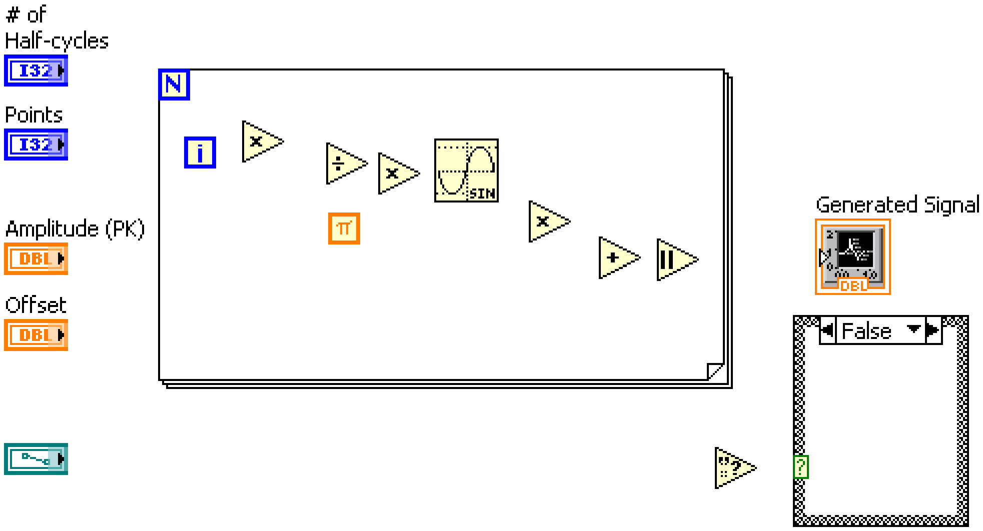
6.1.11 Create the Comparison Operator, Empty String/Path

- Right click over any empty area of the Block Diagram. The Functions Palette appears.

- Open the Programming>>Comparison category.

- Select the *Empty String/Path?* icon. The windows vanish, and you are left holding an Empty String Path? function.

- Left click an empty area of the Block Diagram below the For Loop to place the Empty String/Path? comparison operator.



6.1.12 Create the Write to Spreadsheet File.vi

- Right click over any empty area of the Block Diagram. The Functions Palette appears.

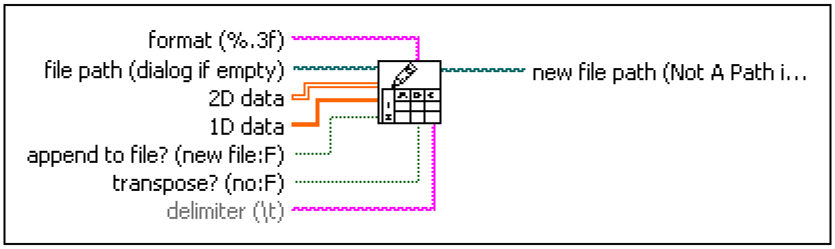
- Open the Programming>>File I/O category.

- Select the *Write to Spreadsheet File.vi*. The windows vanish, and you are left holding a Write to Spreadsheet File.vi SubVI.

- Left click an empty area within the False case of the Case Structure to place the Write to Spreadsheet File.vi SubVI.

Write to Spreadsheet File.vi

The Write to Spreadsheet File.vi is, in fact, a SubVI. If you double-left click it, the Front Panel of the Write to Spreadsheet File.vi will open. This SubVI allows us to create a text file to store values extracted from our VI. The image below shows the many inputs and outputs of this SubVI; you will only use a few functions of this SubVI.

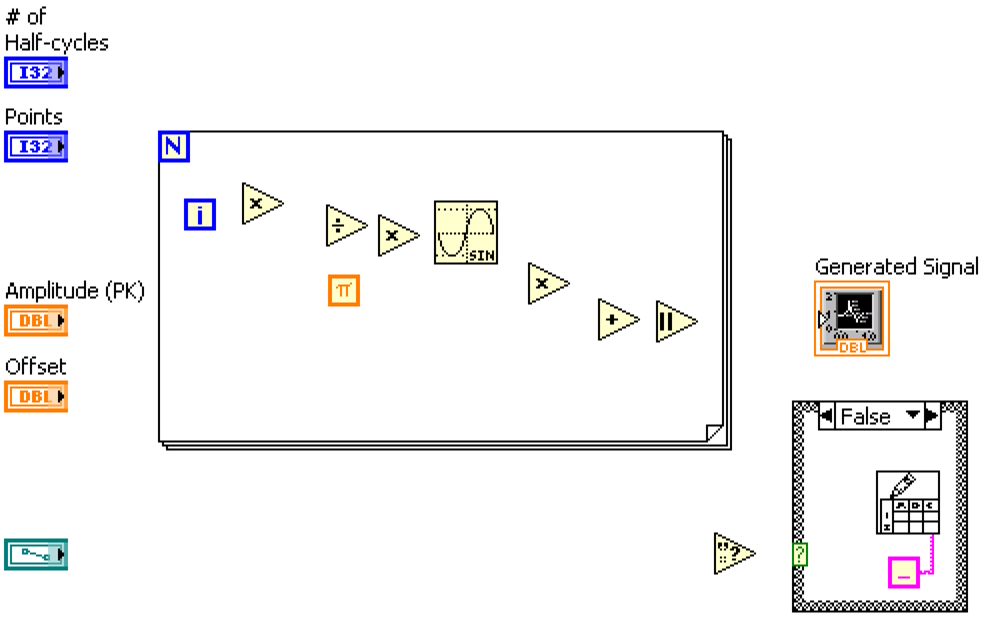

format (%.3f)
file path (dialog if empty)
2D data
1D data
append to file? (new file:F)
transpose? (no:F)
delimiter (\t)
new file path (Not A Path i...

6.1.13 Specify a delimiter for the Write to Spreadsheet File.vi

- Right click an empty area of the Block Diagram to access the Functions Palette.
- Open the *Programming>>String* category.
- Select the Space Constant.
- Place the Space Constant within the False case of the Case Structure.
- Create a wire to connect the output of the Space Constant to the *delimiter (\t)* input of the Write to Spreadsheet File.vi SubVI.

6.1.14 Arrange the Components in Preparation for Wiring

- Verify that your Block Diagram contains all components appearing in the image below.
- Drag each component of the Block Diagram to the approximate position relative to the For Loop, as seen below.



6.1.15 Make the Wire Connections

- Create a wire to connect the output of the “# of Half-cycles” control to the top input of the left-most Multiply function. The wire must tunnel into the For Loop to complete this connection.

- Create a wire to connect the output of the Points control to the input of the Loop Count terminal.

- Create another wire from the output of the Points control to the bottom input of the Divide function. The wire must tunnel into the For Loop to complete this connection.

- Create a wire to connect the output of the Amplitude (PK) control to the bottom input of the right-most Multiply function. The wire must tunnel into the For Loop to complete this connection.

- Create a wire to connect the output of the Offset control to the bottom input of the Add function. The wire must tunnel into the For Loop to complete this connection.

- Create a wire to connect the output of the Write Points to File: control to the input of the Empty String/Path? function.

- Create another wire from the output of the Write Points to File: control connecting to the Write to Spreadsheet File.vi SubVI at the *file path* input. The wire must tunnel into the Case Structure to complete this connection.

- Create a wire to connect the output of the Empty String/Path? function to the input of the Case Selector terminal.

- Create a wire to connect the output of the Loop Iteration counter to the bottom input of the left-most Multiply function.

- Create a wire to connect the output of the left-most Multiply function to the top input of the Divide function.

- Create a wire to connect the output of the Divide function to the top input of the center Multiply function.

- Create a wire to connect the output of the Pi constant to the bottom input of the center Multiply function.

- Create a wire to connect the output of the center Multiply function to the input of the Sine function.

- Create a wire to connect the output of the Sine function to the top input of the right-most Multiply function.

- Create a wire to connect the output of the right-most Multiply function to the input of the Add function.

- Create a wire to connect the output of the Add function to the top input of the Absolute Value function.

- Create a wire to connect the output of the Add function to the input of the Generated Signal waveform chart. Notice the tunnel icon the wire passes through to exit the For Loop (▣) is different than the usual tunnel icon (■).

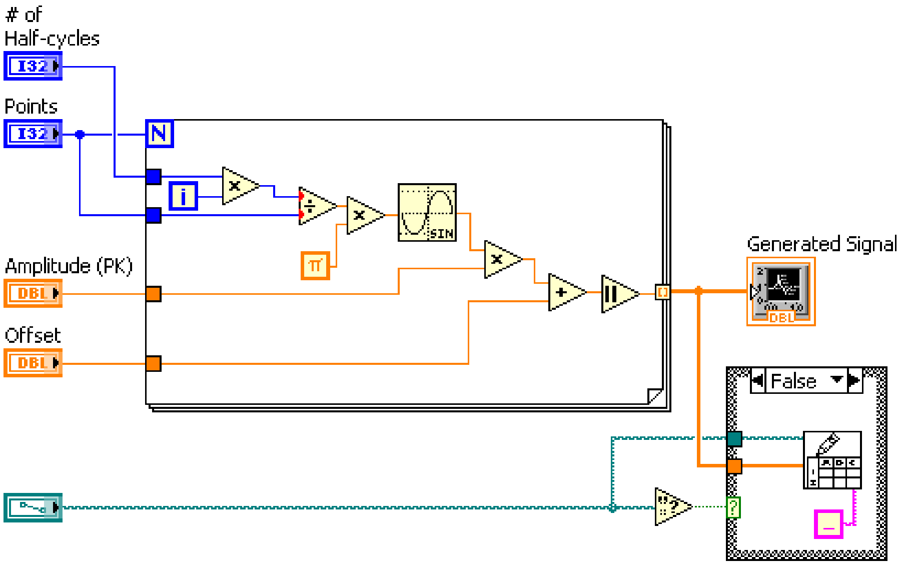
For Loop: Automatic Array Creation
In this VI, the value calculated during each iteration of the For Loop is passed outside the For Loop at the end of the iteration. Because each iteration of the For Loop produces a unique value, the total number of output values will be equal to the value passed to the Loop Count terminal from the Points control.

Auto-Indexed Tunnel

When you tunneled a wire through the For Loop, connecting the Add function's output to the waveform chart, LabVIEW automatically interpreted that you want to pass not just one value, but an array of values. The tunnel icon containing brackets instead of a solid color represents that an array is being automatically created containing the output value for each iteration of the For Loop.

- Create a wire to connect the output of the Auto-Indexed Tunnel to the Write to Spreadsheet File.vi SubVI at the *1D data* input. The wire will tunnel into the Case Structure to make this connection.

- Verify that your wire connections are correct by comparing your Block Diagram to the image below:



6.1.16 Test the Sine Generator VI

- Return to the Front Panel of this VI.

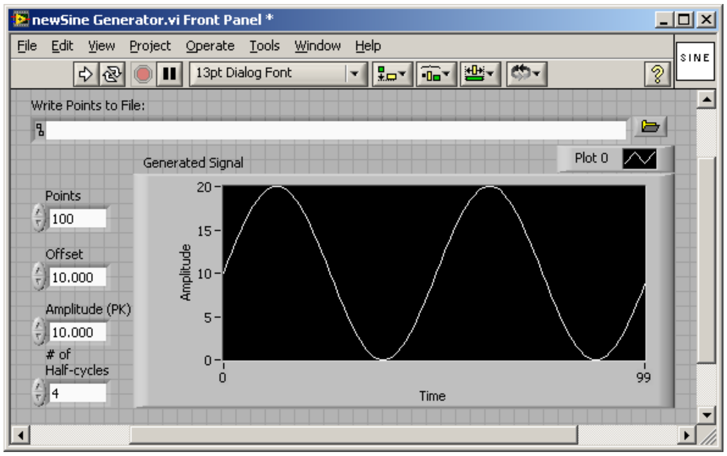
- Enter the value “100” into the “Points” control

- Enter the value “10” into the “Offset” control.

- Enter the value “10” into the “Amplitude (PK)” control.

- Enter the value “4” into the “# of Half-cycles” control.

- Run the VI. Two full cycles of a sine wave should appear on the Generated Signal chart. Each wave is equal to the others, with an Amplitude of ten, and spanning a Time equal to the “Points” control divided by the “# of Half-cycles” Control.



Points: 100

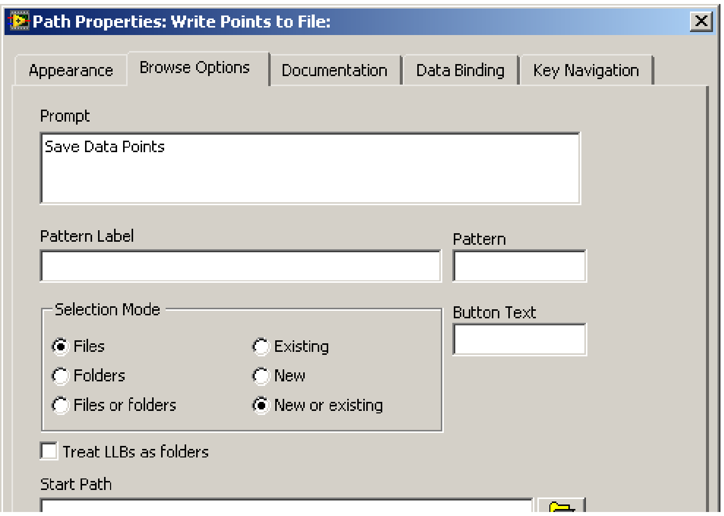
Offset: 10.000

Amplitude (PK): 10.000

# of Half-cycles: 4

6.1.17 Configure the File Path Control

- Enter the properties menu of the File Path Control.
- Switch to the *Browse Options* tab.
- Within the *Selection Mode* area, choose the item: "New or Existing".
- Edit the *Prompt* text field to say: "Save Data Points".
- Verify that all other Browse Options items match the image at right.
- Left click OK to accept the changes and exit the properties menu.



Prompt: Save Data Points

Pattern Label:

Pattern:

Button Text:

Selection Mode

● Files ○ Existing

○ Folders ○ New

○ Files or folders ● New or existing

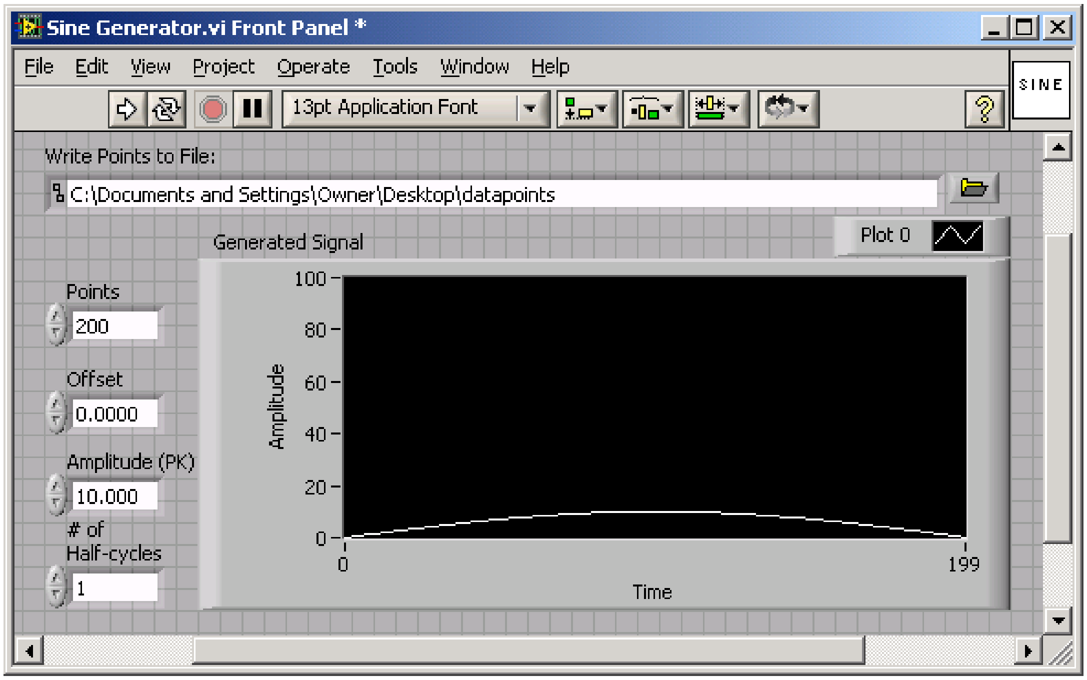
□ Treat LLBs as folders

6.1.18 Create a Datapoints File

- From the Front Panel, enter the value "200" into the "Points" control.
- Enter the value "10" into the "Amplitude (PK)" control.
- Enter the value "0" into the "Offset" Control.
- Enter the value "1" into the "# of Half-cycles" control.

- Left click the *Browse* icon 📂 located to the right of the Write Points to File: control.

- Choose a location and Filename for the datapoints file (you will not need to specify a file type.)


Sine Generator.vi Front Panel *
File Edit View Project Operate Tools Window Help
13pt Application Font
SINE
Generated Signal
Plot 0
Amplitude
100
80
60
40
20
0
0
199
Time

Write Points to File: C:\Documents and Settings\Owner\Desktop\datapoints

Points: 200

Offset: 0.0000

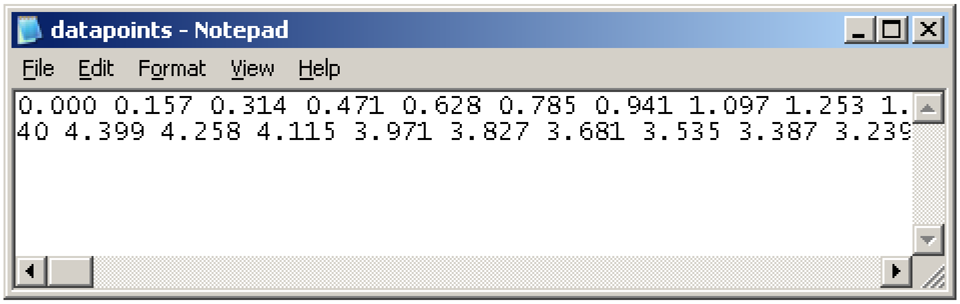
Amplitude (PK): 10.000

# of Half-cycles: 1

- Run the VI. This time a file containing each value of the datapoint array has been saved to the location you specified.

- Locate the datapoint file on your hard drive.

- Open the file with a text editor. The file contains the calculated amplitude values for your sine wave. The values are separated by a single space, as specified on your Block Diagram.

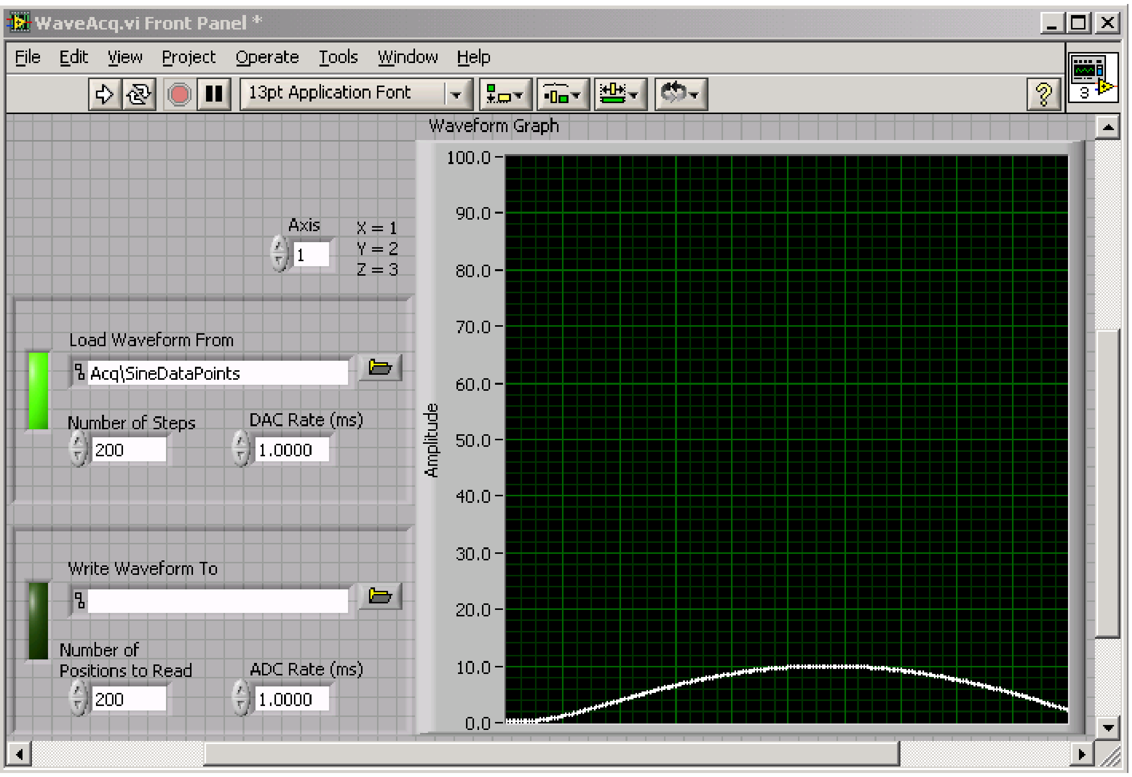


```
0.000 0.157 0.314 0.471 0.628 0.785 0.941 1.097 1.253 1.
40 4.399 4.258 4.115 3.971 3.827 3.681 3.535 3.387 3.239
```

- Don't edit this file, it will be used at the end of the next section.

Section 6.2 – Create the Waveform Acquisition VI

The VI shown below will be created in Section 2 of Chapter 6.



Axis: 1

Load Waveform From: Acq\SineDataPoints

Number of Steps: 200

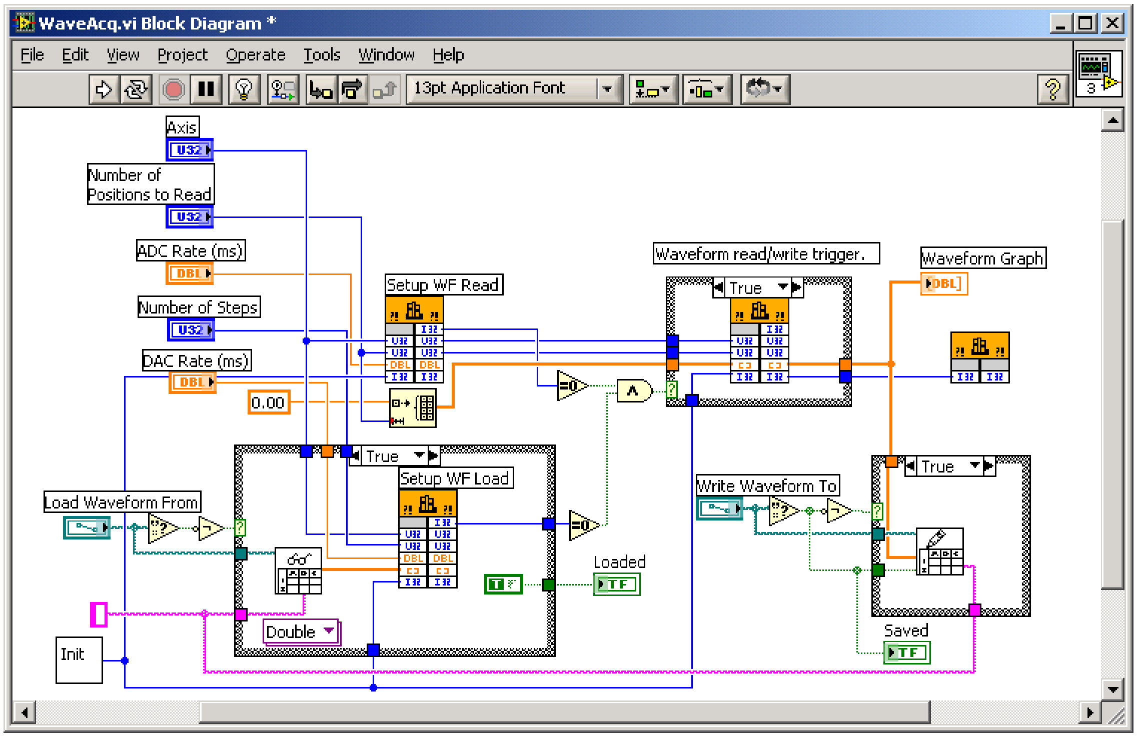
DAC Rate (ms): 1.0000

Write Waveform To:

Number of Positions to Read: 200

ADC Rate (ms): 1.0000

(Image of the finished Chapter - Section 2 Front Panel after one run.)



(Image of the finished Chapter 6 – Section 2 Block Diagram)

6.2.1 Start a New VI, Add the Front Panel Components

Waveform Acquisition
Waveform acquisition can be used to overcome timing uncertainty due to the Windows polling interval (~1ms) and achieve timing resolution down to 33usec (30kHz) depending on the system. This increased time resolution allows more complex motions to be generated.

- Start a New VI, save it as: exWaveAcq

- From the Front Panel, create five Numeric Controls.

- Name the controls: Number of Steps, DAC Rate (ms), Number of Positions to Read, ADC Rate (ms), Axis.

- Change the Representation of the "Number of Steps" control to U32.

- Change the Representation of the "DAC Rate (ms)" control to DBL.

- Edit the Display Format of the "DAC Rate (ms)" control to display as a Floating Point with 4 digits of precision.

- Change the Representation of the "Number of Positions to Read" control to U32.

- Change the Representation of the "ADC Rate (ms)" control to DBL.

- Edit the Display Format of the "ADC Rate (ms)" control to display as a Floating Point with 4 digits of precision.

- Change the Representation of the "Axis" control to U32.

- Create two File Path Controls.

- Name the File Path Controls: "Load Waveform From", and "Write Waveform To".

- Create a Waveform Graph (make sure to create a Waveform *Graph*, not a Waveform Chart.)

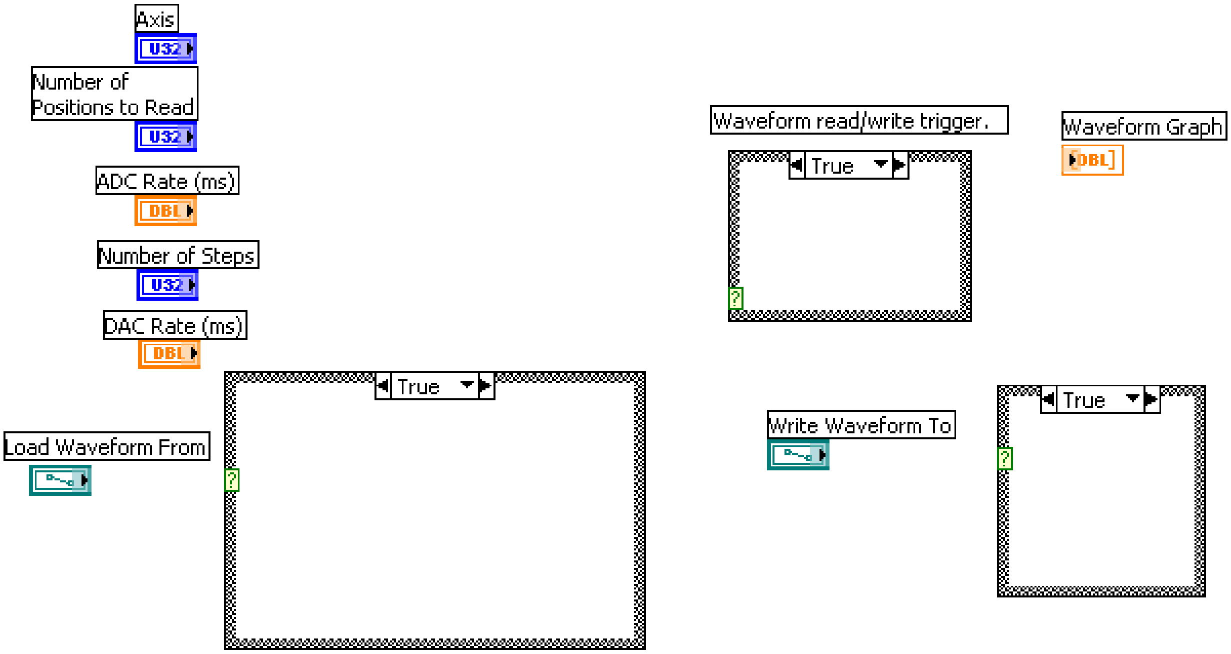
6.2.2 Create the Three Case Structures

- Switch to the Block Diagram.

- Create three Case Structures.

- Space the Case Structures horizontally. Leave enough room between the Structures for the wiring that will be done later.

- The left and right Case Structures should be approximately the same height on your Block Diagram. Drag the middle Case Structure upward so its bottom edge is approximately in line with the top edge of the left and right Case Structures.



6.2.3 Add the Init SubVI and Function Call for MCL_ReleaseHandle

- Add your StandardInitSubVI.vi to the Block Diagram at the left of all the Case Structures.

- Create a Call Library Function Node to the right, and above, the right-most Case Structure.

- Enter the Configure menu of the Call Library Function Node.

- Under the Function tab, change the Library name or path to the address of the MADlib.dll

- From the *Function name* pull-down menu, select MCL_ReleaseHandle.

- Switch to the Parameters tab.

- Add one new parameter.

- Left click the new parameter, arg1, and configure the *Current parameter* area to the following: change the Name to “handle”, Type to “Numeric”, Data type to “Signed 32-bit Integer”, and Pass to “Value”.

- Left click OK to accept the changes and exit the Call Library Function configuration menu.

6.2.5 Prepare the Conditions for the Case Structures

- Create an Empty String/Path? function. This function can be found within the Programming>>Comparison Operators category of the Functions Palette.

- Place the Empty String/Path? function at the left of the left-most Case Structure.

- Create another Empty String/Path? function, and place it at the left of the right-most Case Structure.

- Right click an empty area of the Block Diagram to access the Functions Palette.

- Open the Express>>Arithmetic & Comparison>>Boolean category of the Functions Palette.

- Left click the *Not* icon to create a Not function.

- Place the Not function at left of the left-most Case Structure.

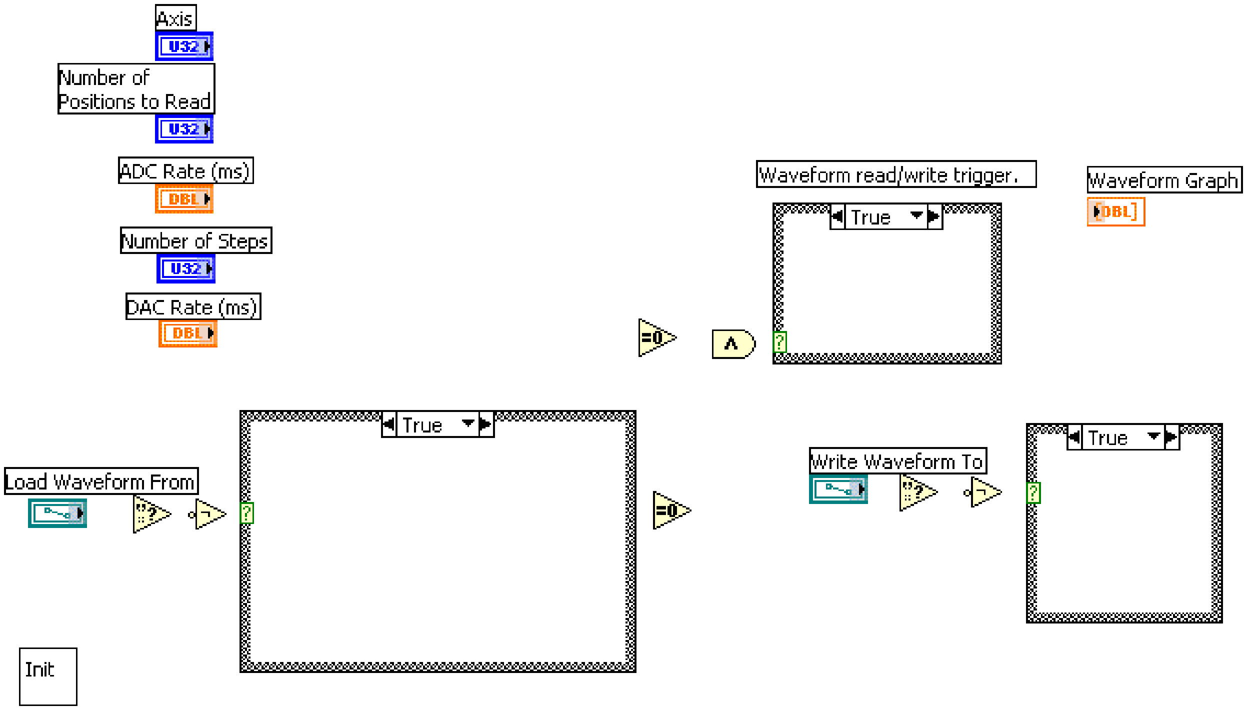
- Create a second Not function and place it to the left of the right-most Case Structure.

- Create an *Equal to 0?* function from the Programming>>Comparison category of the Functions Palette (also found in the Express>>Arithmetic & Comparison>>Comparison category of the Functions palette).

- Place the Equal to 0? function at left of the center Case Structure.

- Create a second Equal to 0? function and place it at the right of the left-most Case Structure.

- Create a Boolean And function at the left of the center Case Structure. The And function can be found within the *Express>>Arithmetic & Comparison>>Boolean* category of the Functions Palette.



6.2.6 *Create the Write/Read from Spreadsheet File.vi SubVIs*

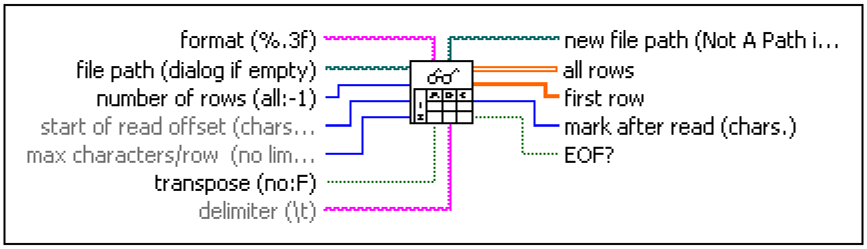
- Create a Write to Spreadsheet File.vi SubVI. This SubVI can be found within the Programming>>File I/O category of the Functions palette.

- Place the Write to Spreadsheet File.vi SubVI into the True case of the right-most Case Structure.

- Create a Read from Spreadsheet File.vi SubVI. This SubVI can also be found within the Programming>>File I/O category of the Functions palette.

Read from Spreadsheet File.vi

The Read from Spreadsheet File.vi is a SubVI which will open and interpret the data of a file at the location passed into its “file path” input. This SubVI is essentially the counterpart to the Write to Spreadsheet File.vi you are already familiar with.

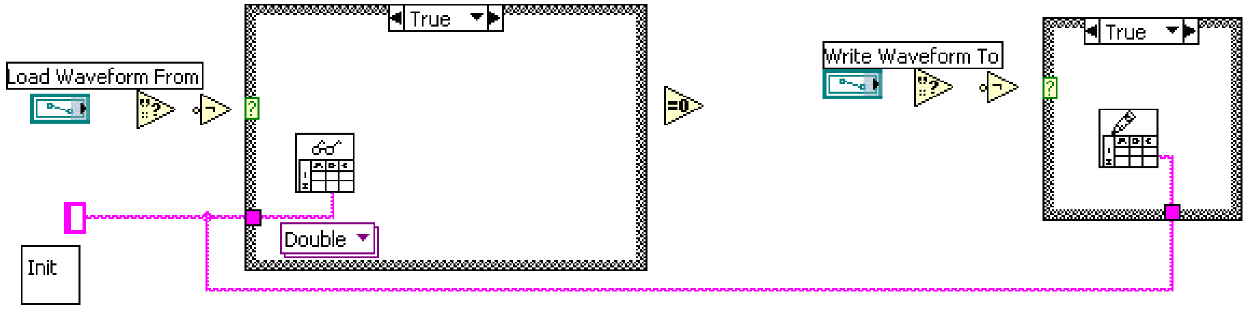

format (%.3f)
file path (dialog if empty)
number of rows (all:-1)
start of read offset (chars...
max characters/row (no lim...
transpose (no:F)
delimiter (\t)
new file path (Not A Path i...
all rows
first row
mark after read (chars.)
EOF?

- Place the Read from Spreadsheet File.vi into the True case of the left-most Case Structure. Once placed, a pull-down menu appears underneath the SubVI containing the items: Double, Integer, and String.

- Select Double from the Read from Spreadsheet File.vi SubVI's *Polymorphic VI* pull-down menu.

- Create an Empty Space constant on the left side of your Block Diagram. The Empty Space Constant can be found within the *Programming>>String* category of the Functions Palette.

- Create a wire to connect the output of the Empty Space constant to the *delimiter (\t)* input of both SubVI's. The wire must tunnel into the Case Structures which contain the SubVIs to make the connections.



6.2.7 Create the Setup Waveform Load Call Library Function Node

- Create a Call Library Function Node in the True Case of the left-most Case Structure.

- Change its Label to "Setup WF Load". With the *Label* property checked, simply double-left click a label to edit it.

MCL_Setup_LoadWaveFormN
MCL_Setup_LoadWaveFormN is a function within the Madlib.dll. This function prepares a waveform (an array of values) to be loaded to the Nano-Drive®DAC. A detailed description of this function can be found within the Madlib_*_*.doc.

- Enter the Configure menu of the Call Library Function Node.

- Under the Function tab, change the Library name or path to the address of the MADlib.dll

- From the *Function name* pull-down menu, select MCL_Setup_LoadWaveFormN.

- Switch to the Parameters tab.

- Left click the “return type” parameter and configure the *Current parameter* area to the following: change the Name to “error code”, Type to “Numeric”, and Data type to “Signed 32-bit Integer”.

- Add five new parameters.

- Left click the first of the new parameters, arg1, and configure the *Current parameter* area to the following: change the Name to “axis”, Type to “Numeric”, Data type to “Unsigned 32-bit Integer”, and Pass to “Value”.

- Left click the second of the new parameters, arg2, and configure the Current parameter area to the following: change the Name to “DataPoints”, Type to “Numeric”, Data type to “Unsigned 32-bit Integer”, and Pass to “Value”.

- Left click the third of the new parameters, arg3, and configure the Current parameter area to the following: change the Name to “milliseconds”, Type to “Numeric”, Data type to “8-byte Double”, and Pass to “Value”.

- Left click the fourth of the new parameters, arg4, and configure the *Current parameter* area to the following: change the Name to “waveform”, Type to “Array”, checkmark the Constant checkbox, change the Data type to “8-byte Double”, Dimensions to 1, Array format to Array Data Pointer, and Minimum size to <None>.

- Left click the fifth, and last, of the new parameters, arg5, and configure the Current parameter area to the following: change the Name to “handle”, Type to “Numeric”, Data type to “Signed 32-bit Integer”, and Pass to “Value”.

- Compare your Function Prototype to the following:

```
int32_t MCL_Setup_LoadWaveFormN(uint32_t axis, uint32_t DataPoints, double
          milliseconds, const double *waveform, int32_t handle);
```

- Left click OK to accept the changes and exit the Call Library Function configuration menu.

6.2.8 Create the Setup Waveform Read Call Library Function Node

- Create a Call Library Function Node above the left-most Case Structure.

- Change its Label to “Setup WF Read”. With the *Label* property checked, double-left click a label to edit it.

MCL_Setup_ReadWaveFormN

MCL_Setup_ReadWaveFormN is a function within the Madlib.dll. This function prepares for a waveform (an array of values) read from the Nano-Drive®ADC. A detailed description of this function can be found within the Madlib_*_*.doc.

- Enter the Configure menu of the Call Library Function Node.
- Under the Function tab, change the Library name or path to the address of the MADlib.dll
- From the *Function name* pull-down menu, select MCL_Setup_ReadWaveFormN.
- Switch to the Parameters tab.
- Left click the “return type” parameter and configure the *Current parameter* area to the following: change the Name to “error code”, Type to “Numeric”, and Data type to “Signed 32-bit Integer”.
- Add four new parameters.
- Left click the first of the new parameters, arg1, and configure the *Current parameter* area to the following: change the Name to “axis”, Type to “Numeric”, Data type to “Unsigned 32-bit Integer”, and Pass to “Value”.
- Left click the second of the new parameters, arg2, and configure the Current parameter area to the following: change the Name to “DataPoints”, Type to “Numeric”, Data type to “Unsigned 32-bit Integer”, and Pass to “Value”.
- Left click the third of the new parameters, arg3, and configure the Current parameter area to the following: change the Name to “milliseconds”, Type to “Numeric”, Data type to “8-byte Double”, and Pass to “Value”.
- Left click the fourth, and last, of the new parameters, arg4, and configure the Current parameter area to the following: change the Name to “handle”, Type to “Numeric”, Data type to “Signed 32-bit Integer”, and Pass to “Value”.
- Compare your Function Prototype to the following:

```
int32_t MCL_Setup_ReadWaveFormN(uint32_t axis, uint32_t DataPoints, double
                    milliseconds, int32_t handle);
```

- Left click OK to accept the changes and exit the Call Library Function configuration menu.

6.2.9 Create the Trigger Waveform Write/Read Call Library Function Node

- Create a Call Library Function Node in the True Case of the center Case Structure.

- Change its Label to “Waveform read/write trigger”.

MCL_TriggerWaveformAcquisition
MCL_TriggerWaveformAcquisition is a function within the Madlib.dll. After setting up a Waveform read/write using the MCL_Setup_ReadWaveFormN and MCL_Setup_LoadWaveFormN functions, use this function to execute the Waveform read/write. A detailed description of this function can be found within the Madlib_*_*.doc.

- Enter the Configure menu of the Call Library Function Node.

- Under the Function tab, change the Library name or path to the address of the MADlib.dll

- From the *Function name* pull-down menu, select MCL_TriggerWaveformAcquisition.

- Switch to the Parameters tab.

- Left click the “return type” parameter and configure the *Current parameter* area to the following: change the Name to “error code”, Type to “Numeric”, and Data type to “Signed 32-bit Integer”.

- Add four new parameters.

- Left click the first of the new parameters, arg1, and configure the *Current parameter* area to the following: change the Name to “axis”, Type to “Numeric”, Data type to “Unsigned 32-bit Integer”, and Pass to “Value”.

- Left click the second of the new parameters, arg2, and configure the Current parameter area to the following: change the Name to “DataPoints”, Type to “Numeric”, Data type to “Unsigned 32-bit Integer”, and Pass to “Value”.

- Left click the third of the new parameters, arg3, and configure the *Current parameter* area to the following: change the Name to “waveform”, Type to “Array”, remove the checkmark from the Constant checkbox, change the Data type to “8-byte Double”, Dimensions to 1, Array format to Array Data Pointer, and Minimum size to <None>.

- Left click the fourth, and last, of the new parameters, arg4, and configure the Current parameter area to the following: change the Name to “handle”, Type to “Numeric”, Data type to “Signed 32-bit Integer”, and Pass to “Value”.

- Compare your Function Prototype to the following:

```
int32_t MCL_TriggerWaveformAcquisition(uint32_t axis, uint32_t DataPoints,
                    double *waveform, int32_t handle);
```

- Left click OK to accept the changes and exit the Call Library Function configuration menu.

6.2.10 *Initialize the Trigger Waveform Function"s Array Output*

- Create an Initialize Array function above the left-most Case Structure. The Initialize Array function can be found within the Array sub-category of the Programming category.

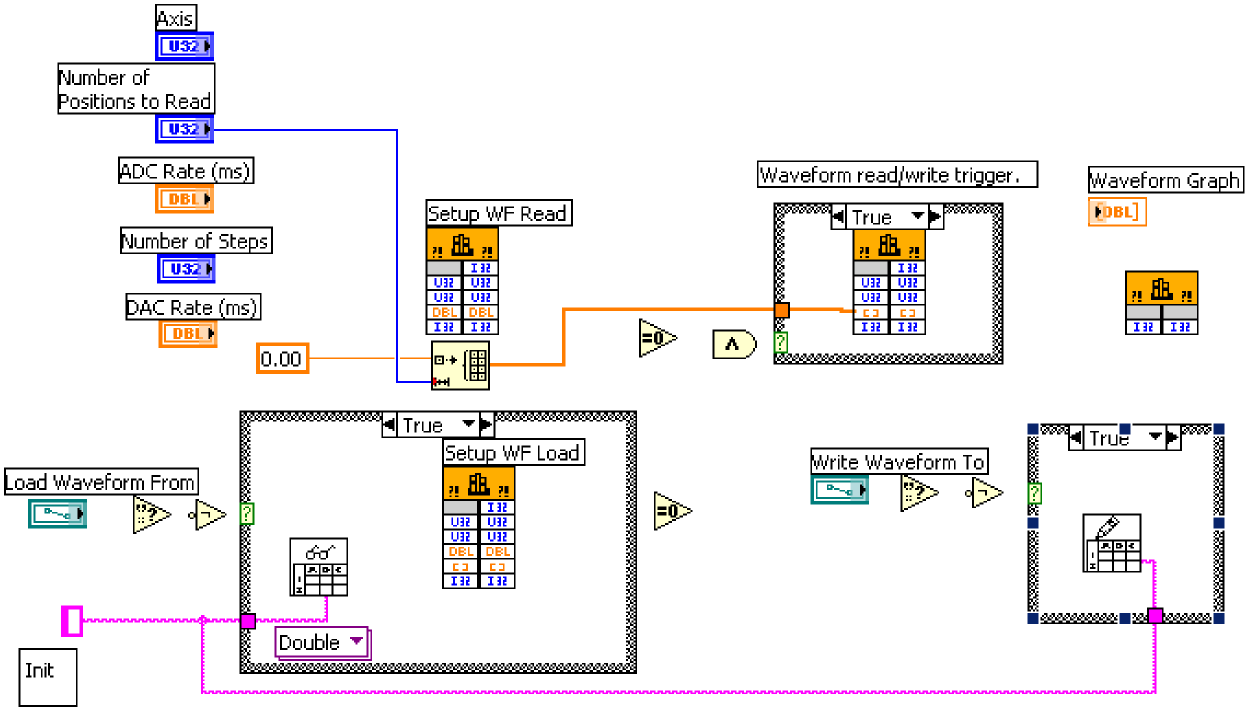
- Create a Numeric Constant with a value of "0" at the left of the Initialize Array function.

- Change the Representation of the Numeric Constant to DBL.

- Create a wire to connect the output of the Numeric Constant to the Initialize Array function's top input, "element".

- Create a wire to connect the output of the Number of Positions to Read control to the Initialize Array function's bottom input, "dimension size".

- Create a wire to connect the output of the Initialize Array function to the "waveform" input of the MCL_TriggerWaveformAcquisition Call Library Function Node. The wire will tunnel into the center Case Structure to make this connection.



6.2.11 Prepare the Front Panel

- Switch back to the Front Panel of your VI.

- If you haven't done so already, arrange the components of your Front Panel to match the appearance of the finished Front Panel as seen in the image presented at the beginning of this section (you are about to create the green LEDs so disregard them for now.)

6.2.12 Create the Square LEDs

- Right click an empty area of the Front Panel. The Controls Palette appears.

- Select the LEDs category of the Express Palette.

- Select the Square LED icon to create a square LED.

- Follow this process again to create a second square LED.

- Name the LEDs: "Loaded", and "Saved".

- Place the LED labeled "Loaded" near the "Load Waveform From" File Path Control.

- Place the LED labeled "Saved" near the "Write Waveform To" File Path Control.

6.2.13 Prepare the Saved/Loaded Alerts

- Switch back to the Block Diagram of your VI.

- Drag the "Loaded" indicator to the right of the left-most Case Structure

- Drag the "Saved" indicator below the right-most Case Structure.

- Right click an empty area of the Block Diagram to access the Functions Palette.

- Locate and place a *True* constant. This can be found within the Boolean sub-category of the Express>>Arithmetic & Comparison category.

- Drag the True constant into the True case of the left-most Case Structure. Place it near the right edge.

- Create a wire to connect the output of the True constant to the input of the "Loaded" indicator. The wire will tunnel out of the Case Structure to make this connection. However, the tunnel appears hollowed.

- Switch to the False case of this Case Structure. The False case must also pass data to the recently created tunnel.

- Create a *False* constant. The False constant can be found within the Express>>Arithmetic & Comparison>>Boolean category of the Functions Palette.

- Place the False constant inside the False case of the left-most Case Structure, near the right edge.

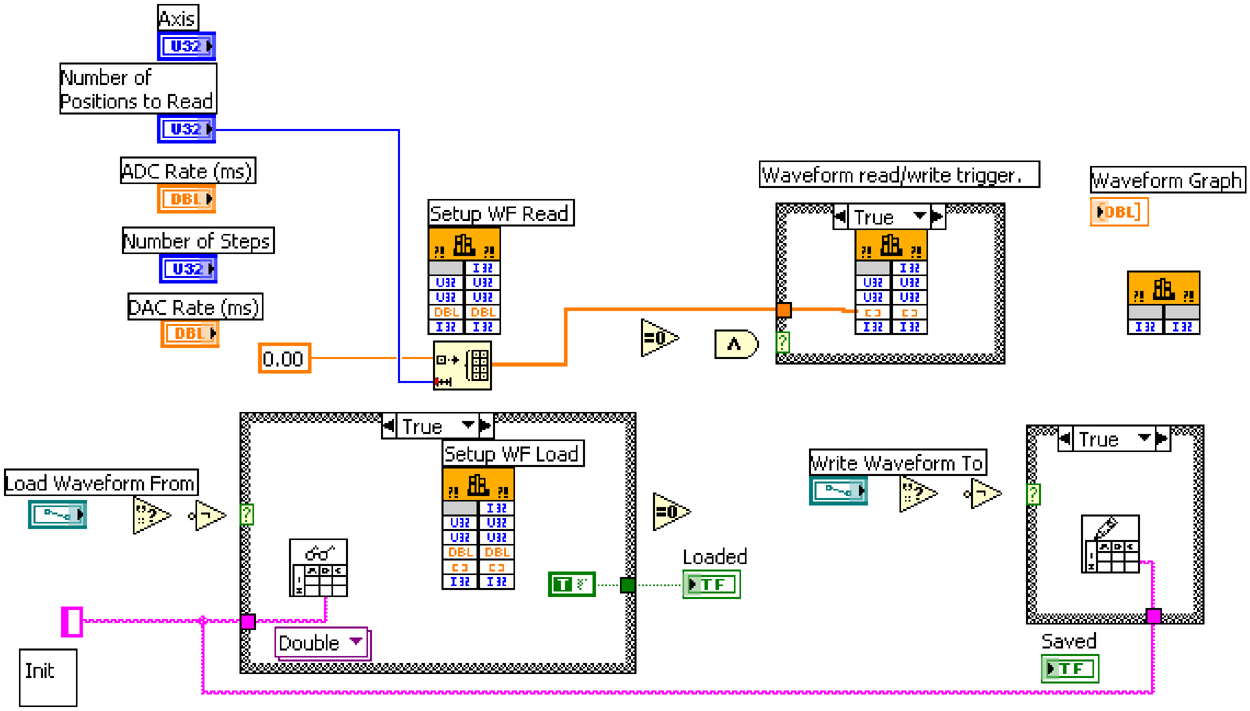
- Create a wire to connect the output of the False constant to the input of the recently created tunnel. The tunnel should now be a solid color. The False constant now passes data to the "Loaded" indicator via this tunnel.

6.2.14 Arrange the Components in Preparation for Wiring

Your VI should now contain all the components seen in the image below. If this is not the case, browse the previous set of instructions for the step(s) you've overlooked or try creating the components according to the image.

- Switch each Case Structure to the True case. Disregard the False case of these Case Structures for now.

- Arrange the components of your Block Diagram to closely match the positioning of the components as seen in the image below.



- Verify that the Data Type Representations of all your controls and indicators are correct before continuing.

6.2.15 Make the Wire Connections (True Case of the Case Structures)

- Create a wire to connect the output of the “Axis” control to the “axis” input of the “Setup WF Read” Call Library Function Node.

- Create a wire to connect the output of the “Axis” control to the “axis” input of the “Setup WF Load” Call Library Function Node.

- Create a wire to connect the output of the “Number of Positions to Read” control to the “Datapoints” input of the “Setup WF Read” Call Library Function Node.

- Create a wire to connect the output of the “ADC Rate (ms)” control to the “milliseconds” input of the “Setup WF Read” Call Library Function Node.

- Create a wire to connect the output of the “Number of Steps” control to the “DataPoints” input of the “Setup WF Load” Call Library Function Node. The wire will tunnel into the left-most Case Structure to make this connection.

- Create a wire to connect the output of the “DAC Rate (ms)” control to the “milliseconds” input of the “Setup WF Load” Call Library Function Node. The wire will tunnel into the left-most Case Structure to make this connection.

- Create a wire to connect the output of the “Load Waveform From” control to the input of the left-most Empty String/Path? function.

- Create a wire to connect the output of the left-most Empty String/Path? function to the input of the left-most Not function.

- Create a wire to connect the output of the left-most Not function to the input of the left-most Case Structure”s Selector Terminal.

- Create a wire to connect the output of the “Load Waveform From” control to the “file path” input of the “Read From Spreadsheet File.vi” SubVI.

- Create a wire to connect the “first row” output of the “Read From Spreadsheet file.vi” SubVI to the “waveform” input of the “Setup WF Load” Call Library Function Node.

- Create a wire to connect the output of the “Init” SubVI to the “handle” input of all three Call Library Function Nodes. The wire will tunnel into two separate Case Structures to make two of these connections.

- Create a wire to connect the “error code” output of the “Setup WF Read” Call Library Function Node to the upper Equal To 0? function.

- Create a wire to connect the “axis” output of the “Setup WF Read” Call Library Function Node to the “axis” input of the “Waveform read/write trigger” Call Library Function Node. The wire will tunnel into a Case Structure to make this connection.

- Create a wire to connect the “DataPoints” output of the “Setup WF Read” Call Library Function Node to the “DataPoints” input of the “Waveform read/write trigger” Call Library Function Node The wire will tunnel into a Case Structure to make this connection.

- Create a wire to connect the “error code” output of the “Setup WF Load” Call Library Function Node to the input of the lower Equal to 0? function. The wire must tunnel out of the Case Structure to make this connection.

- Create a wire to connect the output of the upper Equal to 0? function to the top input of the And function.

- Create a wire to connect the output of the lower Equal to 0? function to the bottom input of the And function.

- Create a wire to connect the output of the And function to the input of the center Case Structure”s Selector Terminal.

- Create a wire to connect the output of the “Write Waveform To” control to the input of the right-most Empty String/Path? function.

- Create a wire to connect the output of the right-most Empty String/Path function to the input of the right-most Not function.

- Create a wire to connect the output of the right-most Not function to the input of the right-most Case Structure’s Selector Terminal.

- Create a wire to connect the output of the “Write Waveform To” control to the “file path” input of the “Write to Spreadsheet File.vi” SubVI. The wire will tunnel into the Case Structure to make this connection.

- Create a wire to connect the output of the right-most Empty String/Path? function to the “append to file?” input of the “Write to Spreadsheet File.vi” SubVI. The wire will tunnel into the Case Structure to make this connection.

- Create a wire to connect the output of the right-most Not function to the input of the “Saved” indicator.

- Create a wire to connect the “waveform” output of the “Waveform read/write trigger” Call Library Function Node to the “Waveform Graph” indicator. The wire will tunnel outside the Case Structure to make this connection.

- Create a wire to connect the “waveform” output of the “Waveform read/write trigger” Call Library Function Node to the “1D data” input of the “Write to Spreadsheet File.vi” SubVI. Avoid creating a second tunnel out of the Case Structure by branching this wire off this output’s existing wire. The wire will tunnel into the right-most Case Structure to make this connection.

- Create a wire to connect the “handle” output of the “Waveform read/write trigger” Call Library Function Node to the “handle” input of the “MCL_ReleaseHandle” Call Library Function Node. The wire will tunnel outside the Case Structure to make this connection.

- The Wire connections for the True cases of the three Case Structures are complete. Compare your Block Diagram to the image of the Finished Block Diagram presented at the beginning of this section.

6.2.16 Complete the Wire Connections for the False Case of the Left-Most Case Structure

- Switch the left-most Case Structure to its False case.

- Create a Numeric Constant in the False case of the left-most Case Structure.

- Set the Representation of the Numeric Constant to I32.

- Set the Value of the Numeric Constant to “-1”.

- Create a wire to connect the output of the “-1” constant to the existing tunnel which passes data to the lower Equal to 0? function.

6.2.17 Complete the Wire Connections for the False Case of the Center Case Structure

- Switch the center Case Structure to its False case.

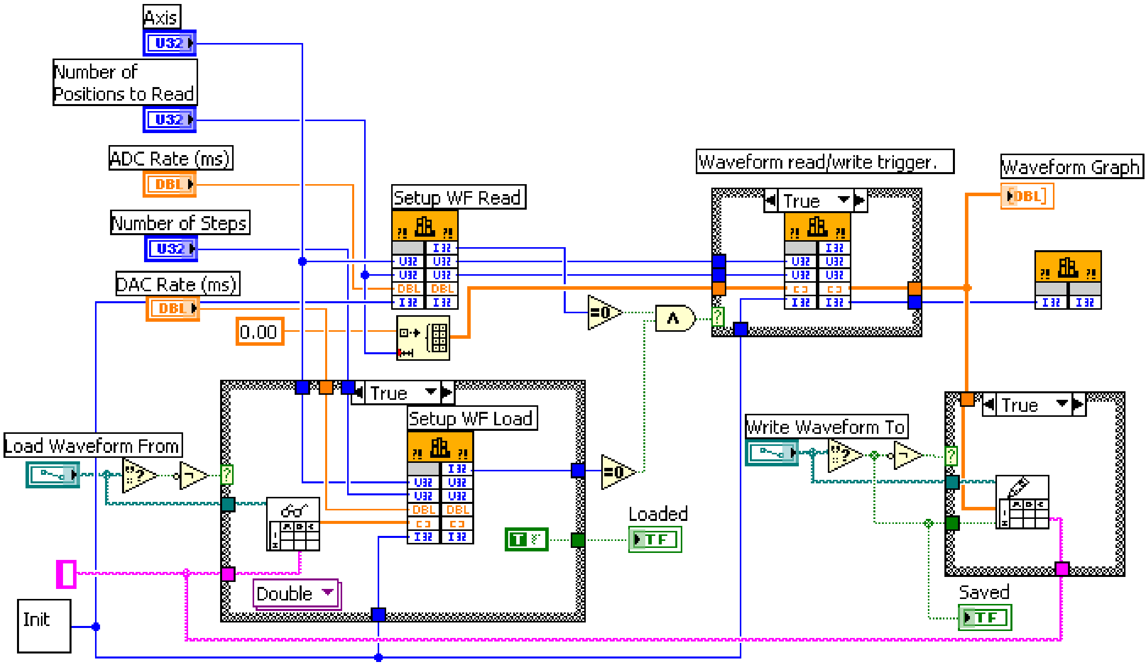
- Create a wire to connect the output of the tunnel passing data from the Initialize Array function to the input of the tunnel which passes data to the “Waveform Graph” indicator and “1D data” input of the “Write to Spreadsheet File.vi” SubVI.

- Create a wire to connect the output of the tunnel passing data from the “Init” SubVI to the input of the tunnel passing data to the “MCL_ReleaseHandle” Call Library function Node.

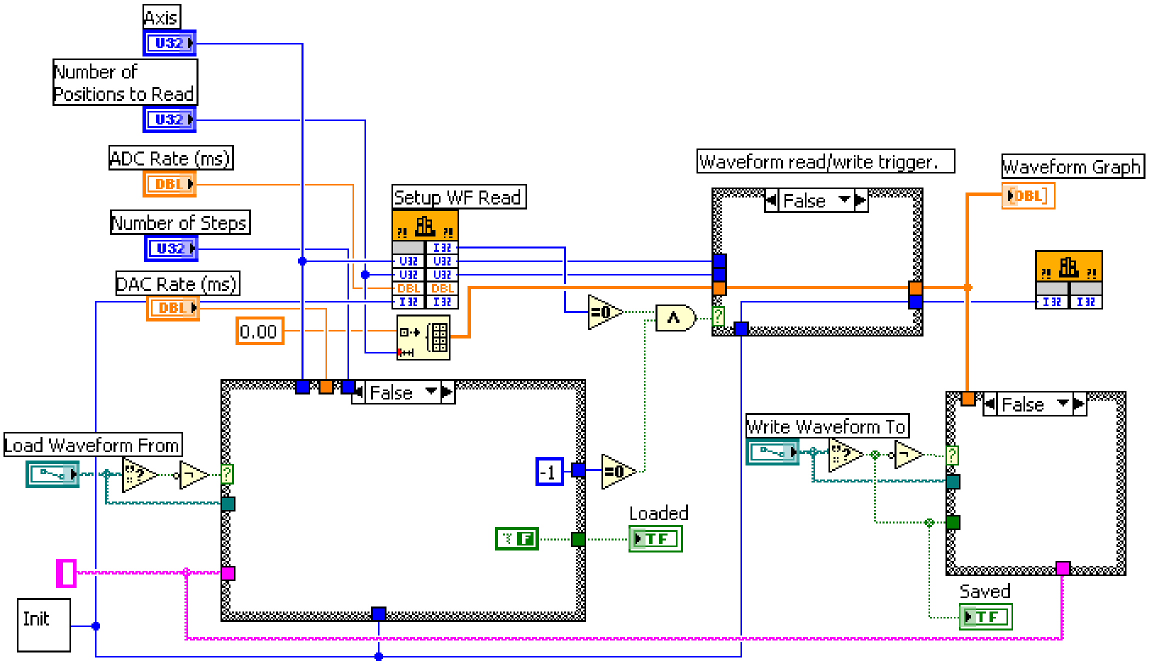
6.2.18 Check for Errors

Below are two separate images of the completed Block Diagram; one depicting the Case Structures in their True case, the other depicting the Case Structures in their False case.

- Compare your Block Diagram to the images below. As can be seen, the False case of the right-most Case Structure should be empty at this point.



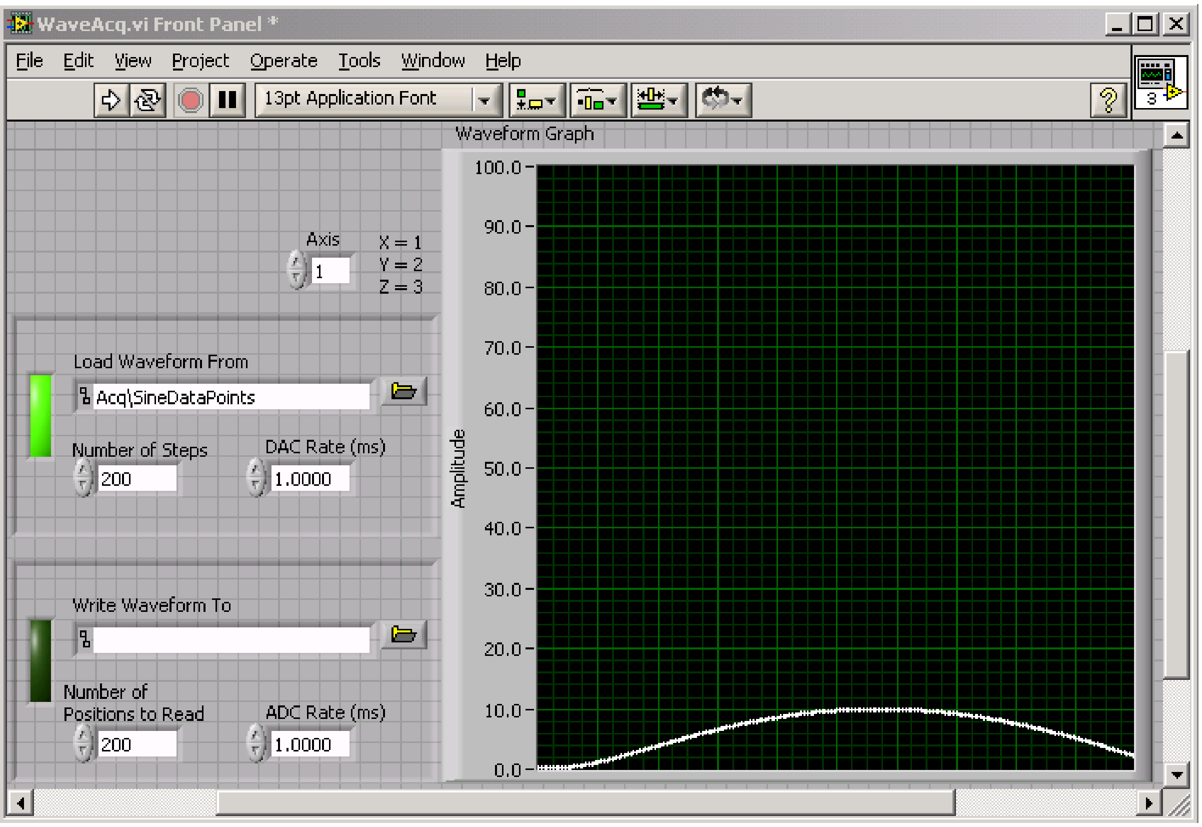
(Image of the Chapter 6 – Section 2 Block Diagram with Case Structures in the True Case)



(Image of the Chapter 6 – Section 2 Block Diagram with Case Structures in the False Case)

Section 6.3 – Prepare the VI to Acquire a Waveform

The VI below shows a graph of actual ADC data read over a span of 200 milliseconds.



Axis: 1

X = 1
Y = 2
Z = 3

Load Waveform From: Acq\SineDataPoints

Number of Steps: 200

DAC Rate (ms): 1.0000

Write Waveform To:

Number of Positions to Read: 200

ADC Rate (ms): 1.0000

(The VI has written the previously generated data points to the DAC)

6.3.1 Set Default Values for the Front Panel Components

- Switch to the Front Panel of your exWaveAcq.vi
- Enter the Value “200” into the “Number of Steps” control.
- Enter the Value “200” into the “Number of Positions to Read” control.
- Set the “DAC Rate (ms)” control to “1”.
- Set the “ADC Rate (ms) control to “1”.
- Left click the *browse* button at right of the “Load Waveform From” control.

- Locate and select the file created by your sine wave generating VI. This file should contain the 200 individual datapoints depicting a sine wave with amplitude of ten, and offset of zero.

- Set the “Axis” control to “1”.

6.3.2 Run the VI

- Verify that your Nano-Drive® is on and properly connected to the computer with a USB cable.

- Run the VI. The Nano-Drive® moves the designated axis ten microns, then returns. On the Waveform Graph you will see a plot of 200 positions read from the Nano-Drive® ADC.

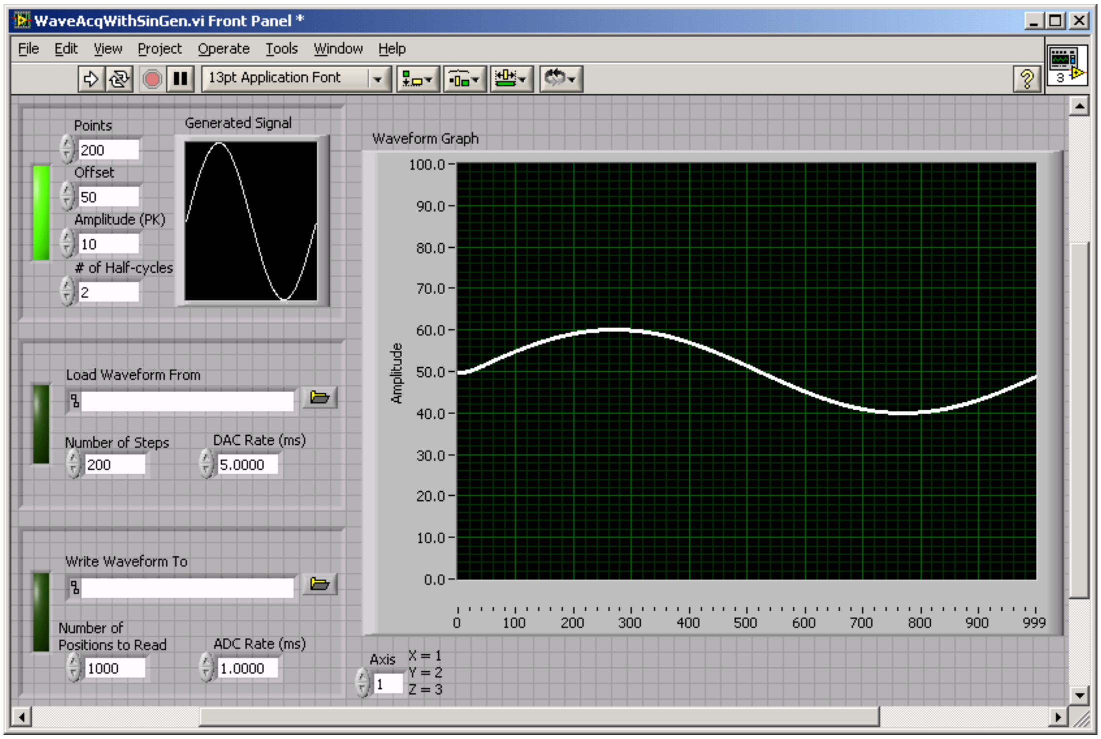
- All other controls can remain at their initial default value, “0”.

- The VI is finished. Save this VI before proceeding with this tutorial.

Note
Contoured areas, and text, as well as other aesthetic effects can be added to the Front Panel. Many of the effects used within this tutorial can be found within the *Modern>>Decorations* category of the Controls palette (don’t forget you can expand the Controls palette to access hidden categories.) Double-left click an empty area to begin adding text.

Section 6.4 – Add the Sine Generator to the VI

The VI shown below will be created in Section 4 of Chapter 6.



Points 200

Offset 50

Amplitude (PK) 10

# of Half-cycles 2

Number of Steps 200

DAC Rate (ms) 5.0000

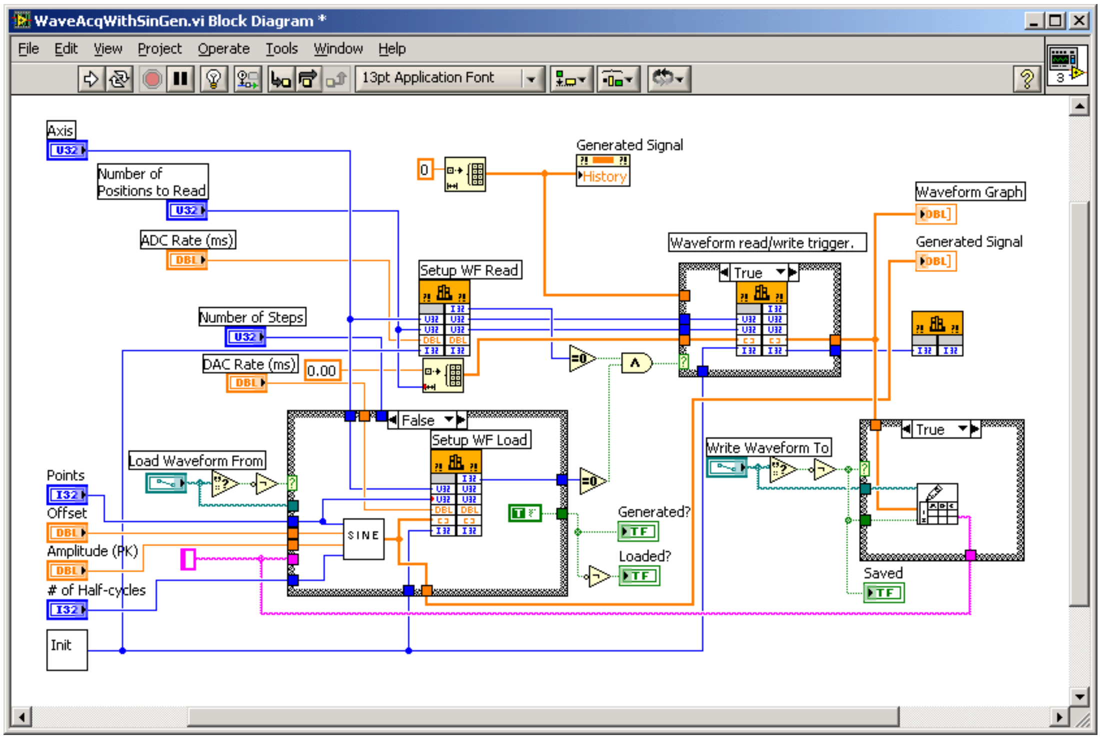
Number of Positions to Read 1000

ADC Rate (ms) 1.0000

Axis 1

X = 1
Y = 2
Z = 3

(Image of the finished Chapter 6 – Section 4 Front Panel after one run.)



(Image of the finished Chapter 6 – Section 4 Block Diagram.)

6.4.1 *Copy the Existing Waveform Acquisition VI*

- Open your Waveform Acquisition VI (built during Chapter 6 – Section 2 of this tutorial.)

- Save the VI as a separate file, named: exWaveAcqWithSinGen

- Compare your Block Diagram to the image above of the finished Block Diagram. Notice the SINE SubVI is used in the previously empty False case of the left-most Case Structure.

6.4.2 *Create the Required Front Panel Components*

- From the Front Panel, create four Numeric Controls.

- Name the controls: Points, Offset, Amplitude (PK), # of Half-cycles.

- Change the Representation of the “Points” control to I32.

- Change the Representation of the “Offset” control to DBL.

- Change the Representation of the “Amplitude (PK)” control to DBL.

- Change the Representation of the “# of Half-cycles” control to I32.

- Create a Waveform Chart (make sure to create a Waveform *Chart*, not a Waveform Graph.)

- Name the Chart: Generated Signal

- Create a green LED.

- Name the LED: Generated

6.4.3 *Prepare the Sine Generation Option*

- From the Block Diagram, switch the left-most Case Structure to its True case.

- Select and copy the “Setup WF Load” Call Library Function Node.

- Switch to the False case of this Case Structure.

- Paste the copy of the “Setup WF Load” Call Library Function Node into the False case of the left-most Case Structure.

- Select and delete the “-1” Numeric Constant which exists inside the False case of this Case Structure.

6.4.4 Add the SINE SubVI

- Right Click an empty area of your Block Diagram to access the Functions Palette.

- Left click the *Select a VI…* item which appears on the Functions Palette.

- Locate and select the exSineGenerator.vi SubVI you created during Chapter 6 – Section 1 of this tutorial

- Place the SubVI within the False case of the left-most Case Structure.

6.4.5 Empty the “Generated Signal” Chart History Before Each Run

- Right-click the “Generated Signal” chart, and select *Create>>Property Node* from the shortcut menu that appears.

- Select *History Data* from the bottom of the *Property Node* menu.

- Place the *History Data* property node in an empty area of the Block Diagram, above all other components.

Editing Property Nodes
Most graphical LabVIEW controls/indicators can be graphically manipulated according to code on the Block Diagram. For instance, the label of an LED could change according to its status or the visibility of a button could alter while the program runs a particular section of code. In this VI you will force the History Data of the Waveform Chart to empty before each run. This will guarantee that the graph is showing only the most recent and relevant data.

- Right click the History Data property node. A properties pop-up menu appears

- Left click the top item: *Change All To Write*. This property node is now configured to be the recipient of data.

- Create an Initialize Array function beside the History Data property node.

- Create a Numeric Constant with the value “0”. Place this at the left of the Initialize Array function.

- Connect the output of the Numeric Constant to the “element” input of the Initialize Array function.

- Connect the output of the Initialize Array function to the “History” input of the “Generated Signal” History Data property node.

6.4.5 *Prepare the LED Indicators*

- Create a Not function, and place it at right of the left-most Case Structure, below the “Loaded” indicator.

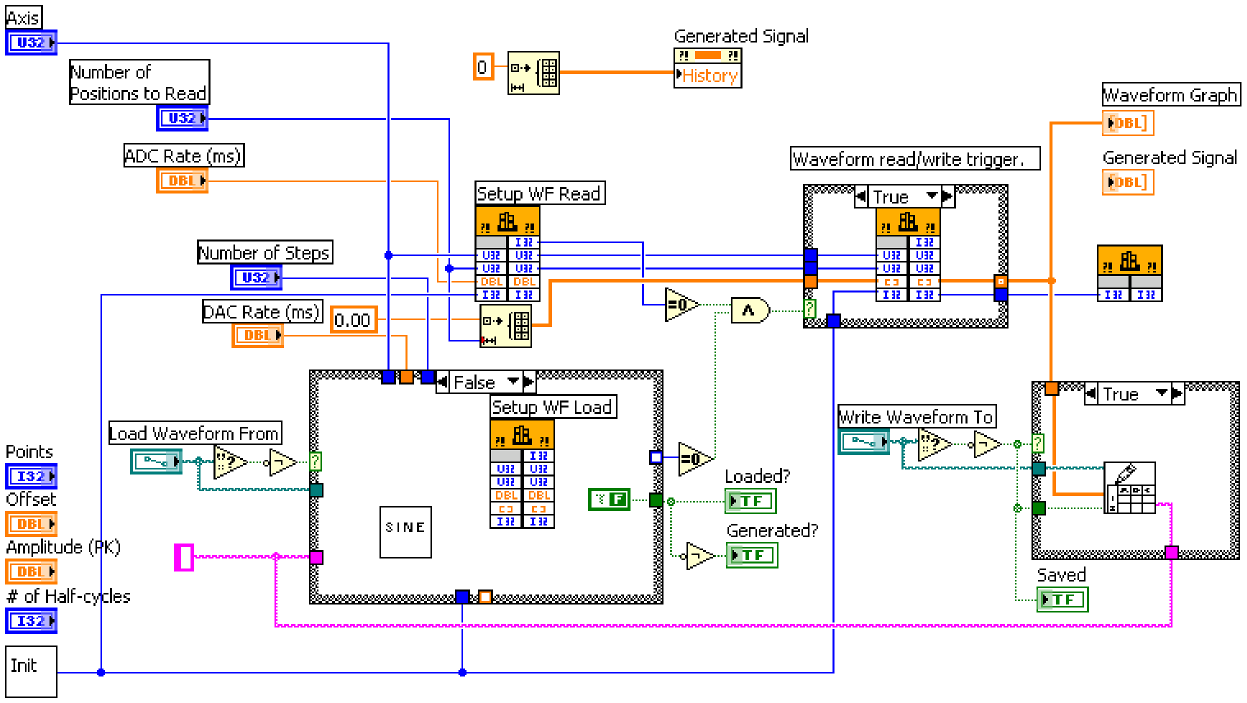
- Drag the “Generated” indicator to the right side of the Not function.

- Create a wire to connect the Not function with the “Loaded” indicator in a parallel connection. In other words, a Boolean Constant passes data through a tunnel to the “Loaded” indicator and you must branch a second wire out of the tunnel’s output and connect this wire to the “Generated” indicator’s input. Be careful to begin the new wire outside the Case Structure; *you should not have to create another tunnel to complete this connection*.

- Connect the output of the Not function to the input of the “Generated” indicator.

6.4.6 *Prepare the Block Diagram Components for Wiring*

- Arrange the newly added components to match your Block Diagram to the one seen in the image below.



6.4.7 Make the Remaining Wire Connections

- Connect the output of the “Points” control to the “Points” input of the SINE SubVI. The wire will tunnel into the Case Structure to make this connection.

- Connect the output of the “Offset” control to the “Offset” input of the SINE SubVI. The wire will tunnel into the Case Structure to make this connection.

- Connect the output of the “Amplitude (PK)” control to the “Amplitude (PK)” input of the SINE SubVI. The wire will tunnel into the Case Structure to make this connection.

- Connect the output of the “# of Half-cycles” control to the “# of Half-cycles” input of the SINE SubVI. The wire will tunnel into the Case Structure to make this connection.

- Connect the “Load Waveform From” control to the “Write points to file:” input of the SINE SubVI. Use the existing tunnel coming from the “Load Waveform From” control to make this connection.

- Connect the output of the SINE SubVI to the “waveform” input of the “Setup WF Load” Call Library Function Node.

- Connect the output of the “Axis” control to the “axis” input of the “Setup WF Load” Call Library Function Node. Use the existing tunnel coming from the “Axis” control to make this connection.

- Connect the output of the “DAC Rate (ms)” control to the “milliseconds” input of the “Setup WF Load” Call Library Function Node. Use the existing tunnel coming from the “DAC Rate (ms)” control to make this connection.

- Connect the output of the “Number of Steps” control to the “Datapoints” input of the “Setup WF Load” Call Library Function Node. Use the existing tunnel coming from the “Number of Steps” control to make this connection.

- Connect the output of the Init SubVI to the “handle” input of the “Setup WF Load” Call Library Function Node. Use the existing tunnel coming from the Init SubVI to make this connection.

- Connect the “error code” output of the “Setup WF Load” Call Library Function Node to the input of the lower *Equal to Zero?* function. Use the existing tunnel connecting to the Equal to Zero? function to make this connection.

- Connect the output of the SINE SubVI to the input of the “Generated Signal” chart. The wire will tunnel out of the Case Structure to make this connection.

- Switch the left-most Case Structure to its True case.

- Create a wire to connect the output of the “Read from Spreadsheet File.vi” SubVI to the input of the “Generated Signal” graph. You should not create a new tunnel out of the Case Structure to make this connection. Instead, use the existing tunnel passing data from the “Sine” SubVI to the “Generated Signal” graph.

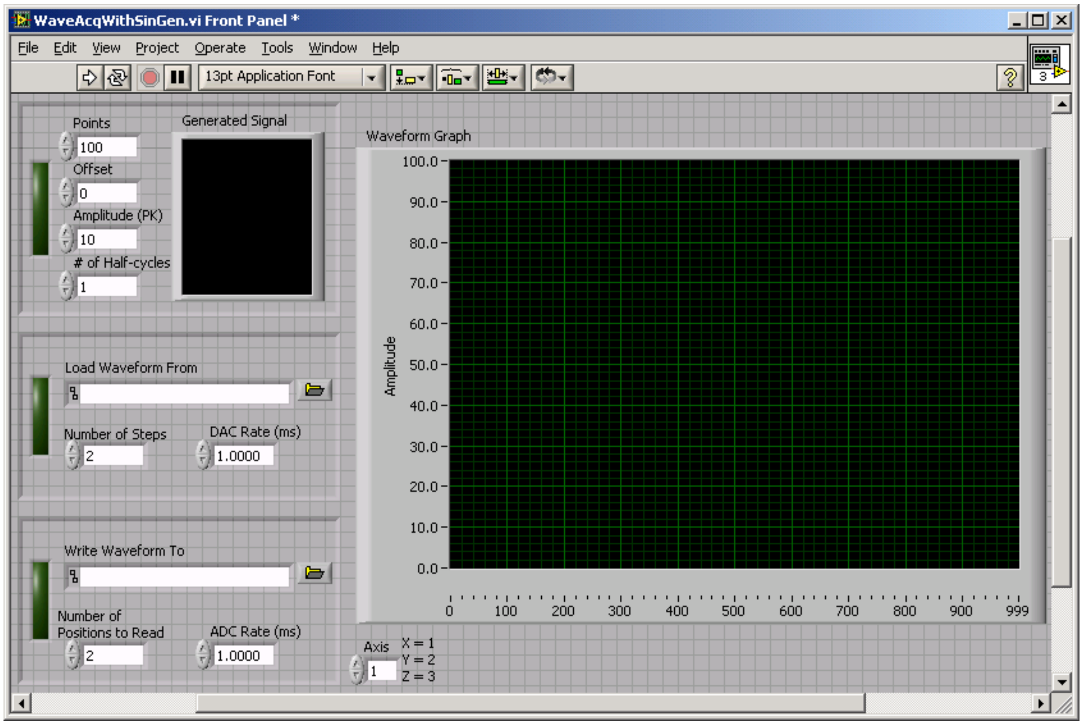
Note
Because the “Sine” SubVI and “Read from Spreadsheet File.vi” SubVI exist in different cases of the Case Structure, they can share a tunnel out of the Case Structure. In fact, tunnels which pass data out of a case structure *must* receive an input from each case of the Case Structure.

- Switch the center Case Structure to its False case.

- Create a wire to connect the output of the upper Initialize Array function to the input of the “Waveform Graph” graph. However, make the wire pass through the False case of the center Case Structure. The new wire must tunnel into the Case Structure, and should use the existing output tunnel which passes data out to the “Waveform Graph” graph, and the “Write Waveform to file” control.

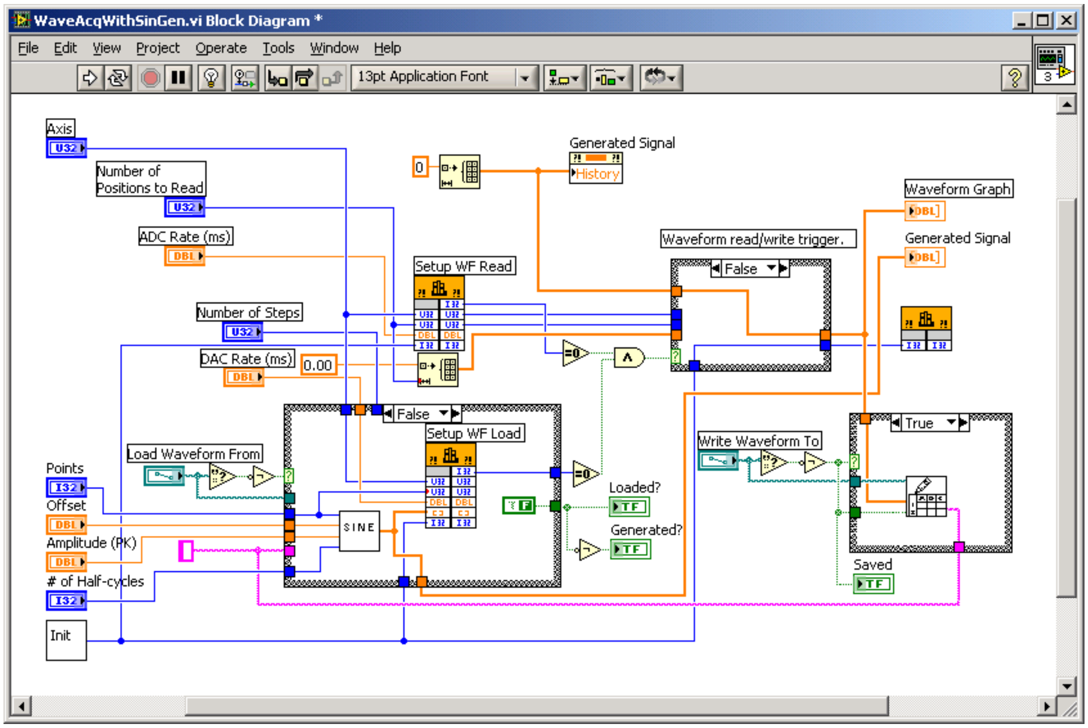
6.4.8 *Prepare to Run the VI*

The VI is finished. Before continuing, compare your VI to the images below.



Points 100
Offset 0
Amplitude (PK) 10
# of Half-cycles 1
Load Waveform From
Number of Steps 2
DAC Rate (ms) 1.0000
Write Waveform To
Number of Positions to Read 2
ADC Rate (ms) 1.0000
Axis 1

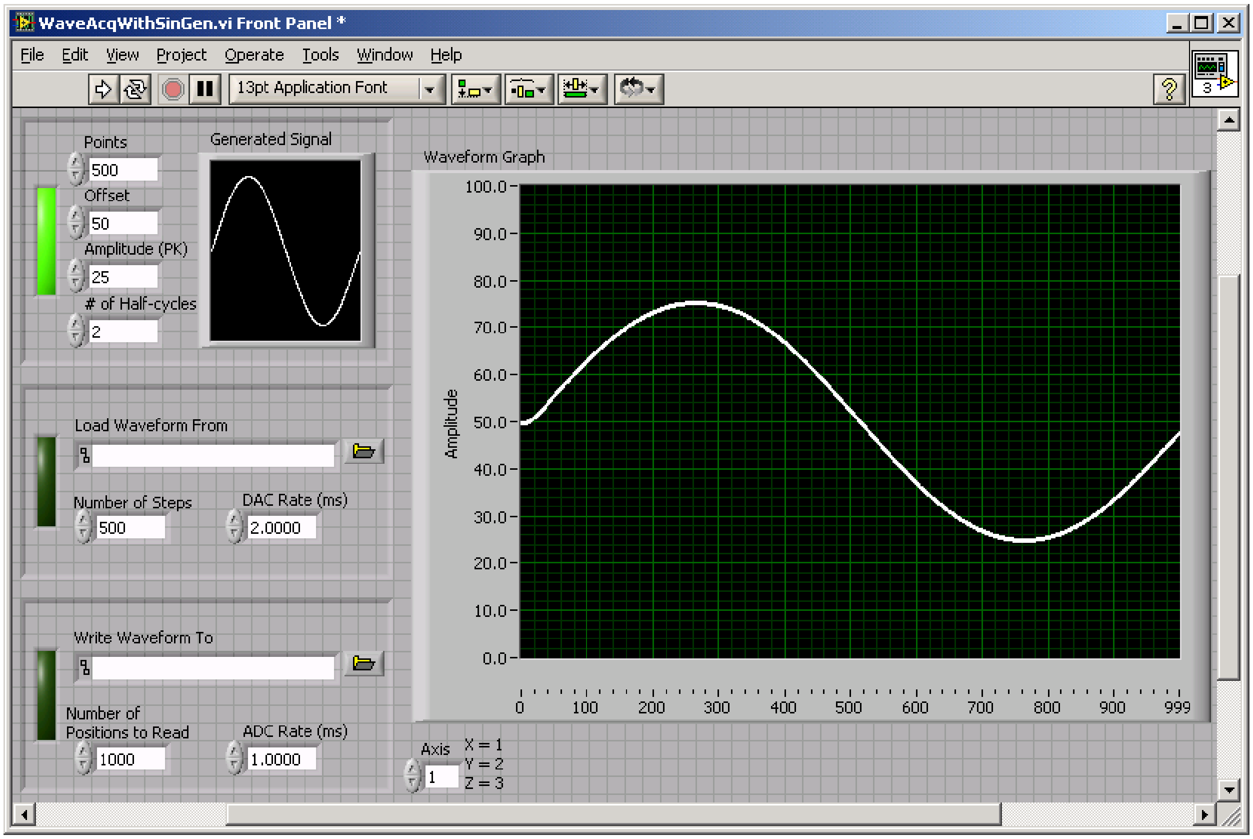
(Image of the finished Chapter 6 - Section 4 Front Panel.)



(Image of the finished Chapter 6 - Section 4Block Diagram.)

Section 6.5 –Generate, Write, and Acquire, a Waveform

The parameters for your VI will be set equal to those appearing in the image below within Section 5 of Chapter 6.



Points 500

Offset 50

Amplitude (PK) 25

# of Half-cycles 2

Number of Steps 500

DAC Rate (ms) 2.0000

Number of Positions to Read 1000

ADC Rate (ms) 1.0000

Axis 1

(Image of the finished VI after one run)

6.5.1 Set Valid Values for the Components

- Switch to the Front Panel of your exWaveAcq.vi
- Clear any text from the “Load Waveform From” and “Write Waveform To” controls.
- Enter the Value “500” into the “Points” control.
- Enter the Value “500” into the “Number of Steps” control.
- Enter the Value “1000” into the “Number of Positions to Read” control.
- Enter the Value “50” into the “Offset” control.

- Enter the Value “25” into the “Amplitude (PK)” control.
- Enter the Value “2” into the “# of Half-cycles” control.
- Set the “DAC Rate (ms)” control to “2”.
- Set the “ADC Rate (ms) control to “1”.
- Set the “Axis” control to “1”.

6.5.2 Run the VI

- Verify your Nano-Drive® is on and properly connected to the computer.
- Run the VI. The Nano-Drive® moves the designated axis, while simultaneously testing its position. On the Waveform Graph you can now see a plot of 1000 positions read from the Nano-Drive® ADC.
- The VI is finished. Save this VI before proceeding with this tutorial.

Appendix A: (Madlib_*_*.doc)

Below is documentation for the Mad City Labs libriary (MadLib.dll)   The documentation provides a quick reference to function prototypes for interfacing MadLib.dll to LabVIEW and C/C++ programs. This document is available as a seperate MS Word file (Madlib_*_*.doc) on your installation disk, and is repeated here as a quick reference for the LabVIEW tutorial.

Index / Quick Reference

Handle Management

```
MADLIB_API   int      MCL_InitHandle();
MADLIB_API   int      MCL_GrabHandle(short device);
MADLIB_API   int      MCL_InitHandleOrGetExisting();
MADLIB_API   int      MCL_GrabHandleOrGetExisting(short device);
MADLIB_API   int      MCL_GrabAllHandles();
MADLIB_API   int      MCL_GetAllHandles(int *handles, int size);
MADLIB_API   int      MCL_NumberOfCurrentHandles();
MADLIB_API   int      MCL_GetHandleBySerial(short serial);
MADLIB_API   void     MCL_ReleaseHandle(int handle);
MADLIB_API   void     MCL_ReleaseAllHandles();
```

Standard Device Movement

```
MADLIB_API   double   MCL_SingleReadZ(int handle);
MADLIB_API   double   MCL_SingleReadN(unsigned int axis, int handle);
MADLIB_API   int      MCL_SingleWriteZ(double zposition, int handle);
MADLIB_API   int      MCL_SingleWriteN(double nposition, unsigned int axis, int handle);
MADLIB_API   double   MCL_MonitorZ(double zposition, int handle);
MADLIB_API   double   MCL_MonitorN(double nposition, unsigned int axis, int handle);
```

Device Information

```
MADLIB_API   bool     MCL_DeviceAttached(unsigned int milliseconds, int handle);
MADLIB_API   double   MCL_GetCalibration(unsigned int axis, int handle);
MADLIB_API   int      MCL_GetFirmwareVersion(short *version, short *profile, int handle);
MADLIB_API   void     MCL_PrintDeviceInfo(int handle);
MADLIB_API   int      MCL_GetSerialNumber(int handle);
MADLIB_API   int      MCL_GetProductInfo(struct ProductInformation *pi, int handle);
MADLIB_API   void     MCL_DLLVersion(short *version, short *revision);
```

Clock Functionality

```
MADLIB_API   int      MCL_ChangeClock(double milliseconds, short clock, int handle);
MADLIB_API   int      MCL_GetClockFrequency(double *adcfreq, double *dacfreq, int handle);
```

Waveform Acquisition

```
MADLIB_API  int   MCL_ReadWaveFormN(
                        unsigned int axis,
                        unsigned int DataPoints,
                        double milliseconds,
                        double* waveform,
                        int handle);
MADLIB_API  int   MCL_Setup_ReadWaveFormN(
                        unsigned int axis,
                        unsigned int DataPoints,
                        double milliseconds,
                        int handle);
MADLIB_API  int   MCL_Trigger_ReadWaveFormN(
                        unsigned int axis,
                        unsigned int DataPoints,
                        double *waveform,
                        int handle);
MADLIB_API  int   MCL_LoadWaveFormN(
                        unsigned int axis,
                        unsigned int DataPoints,
                        double milliseconds,
                        double* waveform,
                         int handle);
MADLIB_API  int   MCL_Setup_LoadWaveFormN(
                        unsigned int axis,
                        unsigned int DataPoints,
                        double milliseconds,
                        double *waveform,
                        int handle);
MADLIB_API  int   MCL_Trigger_LoadWaveFormN(unsigned int axis, int handle);
MADLIB_API  int   MCL_TriggerWaveformAcquisition(
                        unsigned int axis,
                        unsigned int DataPoints,
                              double* waveform,
                         int handle);
```

ISS Option

```
MADLIB_API  int   MCL_PixelClock(int handle);
MADLIB_API  int   MCL_LineClock(int handle);
MADLIB_API  int   MCL_FrameClock(int handle);
MADLIB_API  int   MCL_AuxClock(int handle);
MADLIB_API  int   MCL_IssSetClock(int clock, int mode, int handle);
MADLIB_API  int   MCL_IssResetDefaults(int handle);
MADLIB_API  int   MCL_IssBindClockToAxis(int clock, int mode, int axis, int handle);
MADLIB_API  int   MCL_IssConfigurePolarity(int clock, int mode, int handle);
```

Encoder

```
MADLIB_API  double  MCL_ReadEncoderZ(int handle);
MADLIB_API  int     MCL_ResetEncoderZ(int handle);
```

Tip Tilt Z

```
MADLIB_API   int      MCL_ThetaX(double milliradians, double *actual, int handle);
MADLIB_API   int      MCL_ThetaY(double milliradians, double *actual, int handle);
MADLIB_API   int      MCL_MoveZCenter(double position, double *actual, int handle);
MADLIB_API   int      MCL_LevelZ(double position, int handle);
MADLIB_API   double   MCL_TipTiltHeight(int handle);
MADLIB_API   double   MCL_TipTiltWidth(int handle);
MADLIB_API   int      MCL_MinMaxThetaX(double *min, double *max, int handle);
MADLIB_API   int      MCL_MinMaxThetaY(double *min, double *max, int handle);
MADLIB_API   double   MCL_GetTipTiltThetaX(int handle);
MADLIB_API   double   MCL_GetTipTiltThetaY(int handle);
MADLIB_API   double   MCL_GetTipTiltCenter(int handle);
MADLIB_API   int      MCL_CurrentMinMaxThetaX(double *min, double *max, int handle);
MADLIB_API   int      MCL_CurrentMinMaxThetaY(double *min, double *max, int handle);
MADLIB_API   int      MCL_CurrentMinMaxCenter(double *min, double *max, int handle);
```

Error Codes & Structures

MCL_SUCCESS	0
MCL_GENERAL_ERROR	-1
MCL_DEV_ERROR	-2
MCL_DEV_NOT_ATTACHED	-3
MCL_USAGE_ERROR	-4
MCL_DEV_NOT_READY	-5
MCL_ARGUMENT_ERROR	-6
MCL_INVALID_AXIS	-7
/MCL_INVALID_HANDLE	-8

```
struct ProductInformation {
        unsigned char  axis_bitmap;
        short ADC_resolution;
        short DAC_resolution;
        short Product_id;
        short FirmwareVersion;
        short FirmwareProfile;
}
```

Handle Management

In order to communicate with a Mad City Labs Nano-Drive® a handle must be obtained using one of the following functions. The handle obtained will be passed as an argument to every function that needs to communicate with the Nano-Drive®. At the conclusion of your program all handles should be released.

Function Prototype: int MCL_InitHandle()

Request control of a single Mad City Labs Nano-Drive.

Notes:

-If multiple Mad City Labs Nano-Drives are attached it is indeterminate which Nano-Drive this function will gain control of. (See MCL_GrabHandle, or the combination of MCL_GrabAllHandles and MCL_GetHandleBySerial for ways of dealing with multiple Nano-Drives.)

Return Value:

Returns a valid handle or returns 0 to indicate failure.

Function Prototype: int MCL_GrabHandle(short device)

Request control of a specific type of Mad City Labs device.

Parameters:

device [in] Specifies the type of Nano-Drive. See list below for valid types.
(Values in hexadecimal).

USB 1.1

0x1000 Nano-Drive Single Axis
0x1030 Nano-Drive Three Axis
0x1230 Nano-Drive 20 bit Three Axis
0x1253 Nano-Drive 20 bit Tip/Tilt Z
0x2000 Nano-Gauge

USB 2.0

0x2001 Nano-Drive Single Axis
0x2003 Nano-Drive Three Axis
0x2004 Nano-Drive Four Axis
0x2053 Nano-Drive 16 bit Tip/Tilt Z
0x2201 Nano-Drive 20 bit Single Axis
0x2203 Nano-Drive 20 bit Three Axis
0x2253 Nano-Drive 20 bit Tip/Tilt Z
0x2100 Nano-Gauge
0x2401 C-Focus

Notes:

-If multiple Mad City Labs Nano-Drives of the same type are attached it is indeterminate which of those Nano-Drives this function will gain control of. (Use the combination of MCL_GrabAllHandles and MCL_GetHandleBySerial to differentiate between multiple Nano-Drives of the same type.)

Return Value:

Returns a valid handle or returns 0 to indicate failure.

Function Prototype: int MCL_InitHandleOrGetExisting()

Request control of a single Mad City Labs Nano-Drive. If all attached Nano-Drives are controlled this function will return a handle to one of the Nano-Drives controlled by the DLL.

Notes:

-If multiple Mad City Labs Nano-Drives are attached it is indeterminate which Nano-Drive this function will gain control of. (See MCL_GrabHandle, or the combination of MCL_GrabAllHandles and MCL_GetHandleBySerial for ways of dealing with multiple Nano-Drives.)

-If multiple MadCity Labs Nano-Drives are controlled by the DLL it is indeterminate which will be returned by this function. However, subsequent calls to this function will cycle through all of the controlled Nano-Drives.

Return Value:
Returns a valid handle or returns 0 to indicate failure.

Function Prototype: int MCL_GrabHandleOrGetExisting(short device)
Request control of a specific type of Mad City Labs device. If all attached Nano-Drives of a certain type are controlled this function will return a handle to one of the Nano-Drives of that type controlled by the DLL instance.

Parameters:
device [in] Specifies the type of Nano-Drive. See list below for valid types.
(Values in hexadecimal).

USB 1.1

0x1000 Nano-Drive Single Axis
0x1030 Nano-Drive Three Axis
0x1230 Nano-Drive 20 bit Three Axis
0x1253 Nano-Drive 20 bit Tip/Tilt Z
0x2000 Nano-Gauge

USB 2.0

0x2001 Nano-Drive Single Axis
0x2003 Nano-Drive Three Axis
0x2004 Nano-Drive Four Axis
0x2053 Nano-Drive 16 bit Tip/Tilt Z
0x2201 Nano-Drive 20 bit Single Axis
0x2203 Nano-Drive 20 bit Three Axis
0x2253 Nano-Drive 20 bit Tip/Tilt Z
0x2100 Nano-Gauge
0x2401 C-Focus

Notes:
-If multiple Mad City Labs Nano-Drives of the same type are attached it is indeterminate which of those Nano-Drives this function will gain control of. (Use the combination of MCL_GrabAllHandles and MCL_GetHandleBySerial to differentiate between multiple Nano-Drives of the same type.)

-If multiple Mad City Labs Nano-Drives of the specified type are controlled by the DLL instance it is indeterminate which will be returned by this function. However, subsequent calls to this function will cycle through all of the controlled Nano-Drives of a given type.

Return Value:
Returns a valid handle or returns 0 to indicate failure.

Function Prototype: int MCL_GrabAllHandles()
Requests control of all of the attached Mad City Labs Nano-Drives that are not yet under control.

Return Value:
Returns the number of Nano-Drives currently controlled by this instance of the DLL.

Function Prototype: int MCL_GetAllHandles(int *handles, int size)
Fills a list with valid handles to the Nano-Drives currently under the control of this instance of the DLL.

Parameters
handles [in/out] Pointer to an array of 'size' integers.

size[in]	Size of the 'handles' array

Return Value:
Returns the number of valid handles put into the 'handles' array

---

Function Prototype: int MCL_NumberOfCurrentHandles()

Returns the number of Nano-Drives currently controlled by this instance of the DLL.

Function Prototype: int MCL_GetHandleBySerial(short serial)
Searches Nano-Drives currently controlled for a Nano-Drive whose serial number matches 'serial'.

Parameters:
serial [in]	Serial # of the Nano-Drive whose handle you want to lookup.

Notes:
Since this function only searches through Nano-Drives which the DLL is controlling, MCL_GrabAllHandles() or multiple calls to MCL_(Init/Grab)Handle should be called before using this function.

Return Value:
Returns a valid handle or returns 0 to indicate failure.

Function Prototype: void MCL_ReleaseHandle(int handle)
Performs some cleanup operations and releases control of the specified Nano-Drive.

Function Prototype: void MCL_ReleaseAllHandles()
Performs some cleanup operations as it releases control of all Nano-Drives controlled by this instance of the DLL.

Standard Device Movement

These functions provide a simple interface for changing and reading the position of an axis.

Function Prototype: double MCL_SingleReadZ(int handle)
Function Prototype: double MCL_SingleReadN(unsigned int axis, int handle)

Read the current position of the specified axis.

Requirements:	Firmware version > 0.

Parameters:
Axis [in]	Which axis to move. (X=1,Y=2,Z=3,AUX=4)
Handle [in]	Specifies which Nano-Drive to communicate with.

Return Value:
Returns a position value or the appropriate error code.

Function Prototype: int MCL_SingleWriteZ(double position, int handle)
Function Prototype: int MCL_SingleWriteN(double position, unsigned int axis, int handle)

Commands the Nano-Drive to move the specified axis to a position.

Requirements: Firmware version > 0.

Parameters:

position [in]	Commanded position.
Axis [in]	Which axis to move. (X=1,Y=2,Z=3,AUX=4)
Handle [in]	Specifies which Nano-Drive to communicate with.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: double MCL_MonitorZ(double position, int handle)
Function Prototype: double MCL_MonitorN(double position, unsigned int axis, int handle)

Commands the Nano-Drive to move the specified axis to a position and then reads the current position of the axis.

Requirements: Firmware version > 0.

Parameters:

position [in]	Commanded position.
Axis [in]	Which axis to move. (X=1,Y=2,Z=3,AUX=4)
Handle [in]	Specifies which Nano-Drive to communicate with.

Return Value:
Returns a position value or the appropriate error code.

Device Information

These functions provide information about the Nano-Drive, the Nano-Drive's firmware, and the DLL in use. This information is useful for determining which DLL functions your Nano-Drive supports and if using multiple devices which device you are currently controlling.

Function Prototype: bool MCL_DeviceAttached(unsigned int milliseconds, int handle)
Function waits for a specified number of milliseconds then reports whether or not the Nano-Drive is attached.

Parameters:

milliseconds [in]	Indicates how long to wait.
handle [in]	Specifies which Nano-Drive to communicate with.

Return Value:
Returns true if the specified Nano-Drive is attached and false if it is not.

Function Prototype: double MCL_GetCalibration(unsigned int axis, int handle)
Returns the range of motion of the specified axis.

Requirements: Firmware version > 0.
Parameters:
axis [in] Which axis to get the calibration of. (X=1,Y=2,Z=3,AUX=4)
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns the range of motion or the appropriate error code.

Function Prototype: int MCL_GetFirmwareVersion(short *version, short *profile, int handle)
Gives access to the Firmware version and profile information.

Parameters:
version [in/out] Pointer to the address to be filled with the firmware version number.
profile [in/out] Pointer to the address to be filled with the firmware profile number.
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: void MCL_PrintDeviceInfo(int handle)
Prints to the console product name, product id, DLL version, firmware version, and some product specific information.

Function Prototype: int MCL_GetSerialNumber(int handle)
Returns the serial number of the Nano-Drive. This information can be useful if you need support for your device or if you are attempting to tell the difference between two similar Nano-Drives.

Parameters:
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns the serial number of the specified Nano-Drive.

Function Prototype: int MCL_GetProductInfo(struct ProductInformation *pi, int handle)
Provides some useful device information that is otherwise unavailable (ADC/DAC resolution, available axes, and product id).

Parameters:
pi [in/out] Address of Product information structure to be filled by the function
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: void MCL_DLLVersion(short *version, short *revision)
Gives access to the DLL version information. This information is useful if you need support.

Parameters:
version [in/out] Valid addresses filled with DLL version #
revision [in/out] Valid addresses filled with DLL revision #

Clock Functionality

Most users will never have to use this functionality. They provide control over the hardware clocks used to determine how often ADC data is read and DAC data is written.

On 20-bit systems selecting a slower ADC rate decreases the amount of noise in position reads.

On 16-bit systems the last 30 position reads are averaged by the MCL_SingleRead functions to reduce noise. Modifying the rate at which positions are read increases the period of time over which those data points are collected.

Function Prototype: int MCL_ChangeClock(double milliseconds, short clock, int handle)

Changes the frequency the Nano-Drive reads or writes data

Requirements: Firmware version > 0.

Parameters:

milliseconds [in]

ADC frequency (16 bit)

Range: 5ms – 1/30ms	Nano-Drive Single Axis(USB 1.1, USB 2.0)
Range: 5ms – 1/30ms	C-Focus(USB 2.0)
Range: 5ms – 1/10ms	Nano-Drive Three Axis(USB 1.1 , USB 2.0)
Range: 5ms – 1/10ms	Nano-Drive Tip/Tilt Z(USB 2.0)
Range: 5ms – 1/10ms	Nano-Drive Four Axis(USB 2.0)

ADC frequency (20 bit)

The ADC frequency argument for 20 bit systems is an index into a table of acceptable values. The following are the valid indexes with their associated time periods. (index -> time period): 3 -> 267us , 4 -> 500us , 5 -> 1ms , 6 -> 2ms, 7 -> 10ms , 8 -> 17ms , 9 -> 20ms

DAC frequency:

Range: 5ms – 1/30ms	Nano-Drive Single Axis (USB 1.1, USB 2.0)
Range: 5ms – 1/30ms	C-Focus(USB 2.0)
Range: 5ms – 1/10ms	Nano-Drive Three Axis(USB 1.1 , USB 2.0)
Range: 5ms – 1/10ms	Nano-Drive Tip/Tilt Z(USB 2.0)
Range: 5ms – 1/10ms	Nano-Drive Four Axis(USB 2.0)
Range: 5ms – 1/6ms	Nano-Drive 20 bit Three Axis (USB 1.1, USB 2.0)
Range: 5ms – 1/6ms	Nano-Drive 20 bit Tip/Tilt Z (USB 1.1, USB 2.0)

clock [in]

0 - ADC(reading data)
1 - DAC(writing data)

handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: int MCL_GetClockFrequency(double *adcfreq, double *dacfreq, int handle)

Allows user to see the Nano-Drive's clock frequencies (in milliseconds)

Requirements: Firmware version > 0.

Parameters:
adcfreq [in/out] If successful adcfreq will be set equal to the Nano-Drive's ADC frequency in milliseconds.

Dacfreq [in/out] If successful adcfreq will be set equal to the Nano-Drive's ADC frequency in milliseconds.

handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Waveform Acquisition

Waveform acquisition can be used to overcome the Windows polling interval (~1ms) and achieve timing resolution down to 30kHz depending on the system. This increased time resolution allows for more complex motions to be generated.

Function Prototype: int MCL_ReadWaveFormN(unsigned int axis,unsigned int DataPoints,
double milliseconds,double* waveform, int handle);
This function sets up and triggers a waveform on the specified axis

Requirements: Firmware version > 0.
Firmware profile bit 4 equal to 1.

Parameters:
axis [in] Which axis to move. (X=1,Y=2,Z=3,AUX=4)
DataPoints [in] Number of data points to read.

Range: 1 – 1000	All USB 1.1 products.
Range: 1 – 10000	Nano-Drive Single Axis (USB 2.0)
	C-Focus (USB 2.0)
	Nano-Drive Three Axis (USB 2.0)
	Nano-Drive 16 bit Tip/Tilt Z (USB 2.0)
	Nano-Drive Four Axis (USB 2.0)
Range: 1 – 6666	Nano-Drive 20 bit Single Axis (USB 2.0)
	Nano-Drive 20 bit Three Axis (USB 2.0)
	Nano-Drive 20 bit Tip/Tilt Z (USB 2.0)

milliseconds [in] Rate at which to read data:
16 bit:

Range = 5ms - 1/30ms	Nano-Drive Single Axis (USB 1.1, USB 2.0)
Range = 5ms - 1/30ms	C-Focus (USB 2.0)
Range = 5ms - 1/10ms	Nano-Drive Three Axis (USB 1.1)
Range = 5ms - 1/30ms	Nano-Drive Three Axis (USB 2.0)
Range = 5ms - 1/30ms	Nano-Drive 16 bit Tip/Tilt Z(USB 2.0)
Range = 5ms - 1/30ms	Nano-Drive Four Axis (USB 2.0)

20 bit:
The ADC frequency argument for 20 bit systems is an index into a table of acceptable values. The following are the valid indexes with their associated time periods. (index -> time period): 3 -> 267us, 4 -> 500us , 5 -> 1ms , 6 -> 2ms, 7 -> 10ms , 8 -> 17ms , 9 -> 20ms

waveform [in/out]   Pointer to an array to be filled by this function.
handle [in]   Specifies which Nano-Drive to communicate with.

Notes:
-The ADC frequency reverts to the default ADC frequency after calling this function.

-During a waveform read only the the specified axis records data.
-(16 bit only) A normal read on the specified axis directly following a waveform read will have stale data. The data will be stale for a few milliseconds (1 ms on single axis systems, 3ms on three axis systems).

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: int   MCL_Setup_ReadWaveFormN(unsigned int axis, unsigned int DataPoints, double milliseconds, int handle);
Using this functions makes a waveform read a two step process. Setup then Trigger.

Requirements: Firmware version > 0.
Firmware profile bit 4 equal to 1.

Parameters:
axis [in]   Which axis to move. (X=1,Y=2,Z=3,AUX=4)
DataPoints [in]   Number of data points to read.

Range: 1 – 1000	All USB 1.1 products.
Range: 1 – 10000	Nano-Drive Single Axis (USB 2.0) C-Focus (USB 2.0) Nano-Drive Three Axis (USB 2.0) Nano-Drive 16 bit Tip/Tilt Z (USB 2.0) Nano-Drive Four Axis (USB 2.0)
Range: 1 – 6666	Nano-Drive 20 bit Single Axis (USB 2.0) Nano-Drive 20 bit Three Axis (USB 2.0) Nano-Drive 20 bit Tip/Tilt Z (USB 2.0)

milliseconds [in]   Rate at which to read data:
16 bit:

Range = 5ms - 1/30ms	Nano-Drive Single Axis (USB 1.1, USB 2.0)
Range = 5ms - 1/30ms	C-Focus (USB 2.0)
Range = 5ms - 1/10ms	Nano-Drive Three Axis (USB 1.1)
Range = 5ms - 1/30ms	Nano-Drive Three Axis (USB 2.0)
Range = 5ms - 1/30ms	Nano-Drive 16 bit Tip/Tilt Z(USB 2.0)
Range = 5ms - 1/30ms	Nano-Drive Four Axis (USB 2.0)

20 bit:
The ADC frequency argument for 20 bit systems is an index into a table of acceptable values. The following are the valid indexes with their associated time periods. (index -> time period): 3 -> 267us, 4 -> 500us, 5 -> 1ms , 6 -> 2ms, 7 -> 10ms , 8 -> 17ms , 9 -> 20ms
handle [in]   Specifies which Nano-Drive to communicate with.

Notes:
-The Nano-Drive's ADC frequency will not not change until the waveform is triggered.

-Performing a setup operation on an already setup axis will overwrite the previous setup.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: int MCL_Trigger_ReadWaveFormN(unsigned int axis, unsigned int DataPoints, double *waveform, int handle);

This function triggers a waveform read (previously setup) on the specified axis

Requirements: Firmware version > 0.
Firmware profile bit 4 equal to 1.

Parameters:

axis [in]	Which axis to move. (X=1,Y=2,Z=3,AUX=4)	
DataPoints [in]	Number of data points to read.	
	Range: 1 – 1000	All USB 1.1 products.
	Range: 1 – 10000	Nano-Drive Single Axis (USB 2.0) C-Focus (USB 2.0) Nano-Drive Three Axis (USB 2.0) Nano-Drive 16 bit Tip/Tilt Z (USB 2.0) Nano-Drive Four Axis (USB 2.0)
	Range: 1 – 6666	Nano-Drive 20 bit Single Axis (USB 2.0) Nano-Drive 20 bit Three Axis (USB 2.0) Nano-Drive 20 bit Tip/Tilt Z (USB 2.0)
waveform [in/out]	Pointer to an array to be filled by this function.	
handle [in]	Specifies which Nano-Drive to communicate with.	

Notes:
-The axis must have been setup prior to calling this function.

-Data Points and axis much match their setup values.

-The Nano-Drive's ADC frequency changes prior to data acquisition and reverts to the Nano-Drive's default ADC frequency after calling this function.

-During a waveform read only the the specified axis records data.

-(16 bit only) A normal read on the specified axis directly following a waveform read will have stale data. The data will be stale for a few milliseconds (1 ms on single axis systems, 3ms on three axis systems).

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: int MCL_LoadWaveFormN(
unsigned int axis,
unsigned int DataPoints,
double milliseconds,
double* waveform,
int handle);

This function sets up and triggers a waveform load on the specified axis

Requirements: Firmware version > 0.
Firmware profile bit 4 equal to 1.

Parameters:

axis [in]	Which axis to move. (X=1,Y=2,Z=3,AUX=4)	
DataPoints [in]	Number of data points to read.	
	Range: 1 – 1000	All USB 1.1 products.
	Range: 1 – 10000	Nano-Drive Single Axis (USB 2.0) C-Focus (USB 2.0)

Nano-Drive Three Axis (USB 2.0)
Nano-Drive Tip/Tilt Z(USB 2.0)
Nano-Drive Four Axis (USB 2.0)
Range: 1 – 6666 Nano-Drive 20 bit Single Axis (USB 2.0)
Nano-Drive 20 bit Three Axis (USB 2.0)
Nano-Drive 20 bit Tip/Tilt Z (USB 2.0)

milliseconds [in] Rate at which to write data.
Range: 5ms – 1/30ms Nano-Drive Single Axis (USB 1.1, USB 2.0)
Range: 5ms – 1/30ms C-Focus (USB 2.0)
Range: 5ms – 1/10ms Nano-Drive Three Axis (USB 1.1)
Range: 5ms – 1/30ms Nano-Drive Three Axis (USB 2.0)
Range: 5ms – 1/30ms Nano-Drive 16 bit Tip/Tilt Z (USB 2.0)
Range: 5ms – 1/30ms Nano-Drive Four Axis (USB 2.0)
Range: 5ms – 1/6ms Nano-Drive 20 bit Three Axis (USB 1.1, USB 2.0)
Range: 5ms – 1/6ms Nano-Drive 20 bit Tip/Tilt Z (USB 1.1, USB 2.0)

waveform [in] Pointer to an array of commanded positions.
handle [in] Specifies which Nano-Drive to communicate with.

Notes:
-The Nano-Driver's DAC frequency reverts to the Nano-Drive's default DAC frequency after calling this function.

-Only the specified axis will have control positions written to it while there is waveform data to process.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: int MCL_Setup_LoadWaveFormN(unsigned int axis, unsigned int DataPoints, double milliseconds, double *waveform, int handle);

This function sets up a waveform load on the specified axis

Requirements: Firmware version > 0.
Firmware profile bit 4 equal to 1.

Parameters:
axis [in] Which axis to move. (X=1,Y=2,Z=3,AUX=4)
DataPoints [in] Number of data points to read.
Range: 1 – 1000 All USB 1.1 products.
Range: 1 – 10000 Nano-Drive Single Axis (USB 2.0)
C-Focus (USB 2.0)
Nano-Drive Three Axis (USB 2.0)
Nano-Drive Tip/Tilt Z(USB 2.0)
Nano-Drive Four Axis (USB 2.0)
Range: 1 – 6666 Nano-Drive 20 bit Single Axis (USB 2.0)
Nano-Drive 20 bit Three Axis (USB 2.0)
Nano-Drive 20 bit Tip/Tilt Z (USB 2.0)

milliseconds [in] Rate at which to write data.
Range: 5ms – 1/30ms Nano-Drive Single Axis (USB 1.1, USB 2.0)
Range: 5ms – 1/30ms C-Focus (USB 2.0)
Range: 5ms – 1/10ms Nano-Drive Three Axis (USB 1.1)
Range: 5ms – 1/30ms Nano-Drive Three Axis (USB 2.0)
Range: 5ms – 1/30ms Nano-Drive 16 bit Tip/Tilt Z (USB 2.0)
Range: 5ms – 1/30ms Nano-Drive Four Axis (USB 2.0)
Range: 5ms – 1/6ms Nano-Drive 20 bit Three Axis (USB 1.1, USB 2.0)
Range: 5ms – 1/6ms Nano-Drive 20 bit Tip/Tilt Z (USB 1.1, USB 2.0)

waveform [in] Pointer to an array of commanded positions.
handle [in] Specifies which Nano-Drive to communicate with.

Notes:
-Nano-Drive's DAC frequency does not change until the waveform is triggered.

-Performing a setup operation on an already setup axis will overwrite the previous setup

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: int MCL_Trigger_LoadWaveFormN(unsigned int axis, int handle);
This function triggers a waveform load on the specified axis

Requirements: Firmware version > 0.
Firmware profile bit 4 equal to 1.

Parameters:
axis [in] Which axis to move. (X=1,Y=2,Z=3,AUX=4)
handle [in] Specifies which Nano-Drive to communicate with.

Notes:
-The axis must have been setup prior to calling this function.

-The Nano-Drive's DAC frequency reverts to the Nano-Drive's default DAC frequency after calling this function.

-Only the specified axis will have control positions written to it while there is waveform data to process.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: int MCL_TriggerWaveformAcquisition(unsigned int axis,unsigned int DataPoints,
double* waveform, int handle);
Triggers a waveform read and a waveform load on the specified axis synchronously

Requirements: Firmware version > 0.
Firmware profile bit 4 equal to 1.

Parameters:
axis [in] Which axis to move. (X=1,Y=2,Z=3,AUX=4)
DataPoints [in] Number of data points to read.
Range: 1 – 1000 All USB 1.1 products.
Range: 1 – 10000 Nano-Drive Single Axis (USB 2.0)
C-Focus (USB 2.0)
Nano-Drive Three Axis (USB 2.0)
Nano-Drive 16 bit Tip/Tilt Z (USB 2.0)
Nano-Drive Four Axis (USB 2.0)
Range: 1 – 6666 Nano-Drive 20 bit Single Axis (USB 2.0)
Nano-Drive 20 bit Three Axis (USB 2.0)
Nano-Drive 20 bit Tip/Tilt Z (USB 2.0)
waveform [in] Pointer to an array of commanded positions.
handle [in] Specifies which Nano-Drive to communicate with.

Notes:
-The axis must have a read & a load waveform setup.

-DataPoints and axis must match the values used to setup the read waveform.

-Only the specified axis will have control positions written to it while there is waveform data to process

-During a waveform read only the the specified axis records data.

-(16 bit only) A normal read on the specified axis directly following a waveform read will have stale data. The data will be stale for a few milliseconds (1 ms on single axis systems, 3ms on three axis systems).

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

ISS Option

The ISS option allows programmatic control over four external clocks. The external clocks provide TTL pulses as input to other equipment. The clocks can be pulsed, set high, set low, or bound to NanoDrive events. For example you could signal an external device to start recording data prior to running a waveform and signal it to stop when the waveform completes.

Function Prototype: int MCL_PixelClock(int handle)
Function Prototype: int MCL_LineClock(int handle)
Function Prototype: int MCL_FrameClock(int handle)
Function Prototype: int MCL_AuxClock(int handle)

These functions allow programmatic control of the four external clocks that make up the ISS option. Calling these functions will generate a 250 ns pulse on the corresponding clock. The polarity of these pulses can be configured with MCL_IssConfigurePolarity. By default all pulses will be low to high.

Requirements: Firmware version > 0.
Image Scan Sync (ISS) option.

Parameters:
handle [in] Specifies which Nano-Drive to communicate with.

Notes:
-Using MCL_IssSetClock to set a clock high or low will have an affect on these functions. For example if a clock is set high and has its default polarity the "pulse" will be seen as a falling edge and the clock will remain low after this function.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: int MCL_IssSetClock(int clock, int mode, int handle)
Sets an external clock high or low.

Requirements: Firmware version > 0.
Firmware profile bit 1 equal to 1.

Image Scan Sync (ISS) option.

Parameters:

clock [in]	Which clock to set. (1=Pixel, 2=Line, 3=Frame, 4=Aux)
mode [in]	Determines whether to set the clock high or low. 0 = Sets clock low. 1 = Sets clock high.
handle [in]	Specifies which Nano-Drive to communicate with.

Notes:
-Does not affect the polarity of clocks bound to axes or events by MCL_IssBindClockToAxis.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: int MCL_IssResetDefaults(int handle)
Resets the Iss option to its default values. No axis is bound. All polarities are low to high. The Pixel clock is bound to the waveform read event. The line clock is bound to the waveform write event.

Requirements: Firmware version > 0.
Firmware profile bit 1 equal to 1.
Image Scan Sync (ISS) option.

Parameters:
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: int MCL_IssBindClockToAxis(int clock, int mode, int axis, int handle)
Allows an external clock pulse to be bound to the read of a particular axis. Clocks may also be bound to portions of the waveform functionality. A clock can be bound to the data acquisition of a read waveform so that every time a point is recorded a clock pulse is generated. Additionally, a clock can be pulsed prior to the first point and pulsed after the last point of a position command waveform.

Requirements: Firmware version > 0.
Firmware profile bit 1equal to 1.
Image Scan Sync (ISS) option.

Parameters:

clock [in]	Which clock to bind. (1=Pixel, 2=Line, 3=Frame, 4=Aux)
mode [in]	Selects polarity of the clock to be bound or selects to unbind the axis. 2 = low to high pulse. 3 = high to low pulse. 4 = unbind the axis.

axis [in] Axis or event to bind a clock to.
1 = X axis
2 = Y axis
3 = Z axis
4 = Aux axis
5 = Waveform Read.
6 = Waveform Write.

handle [in] Specifies which Nano-Drive to communicate with.

Notes:
-Each axis read and waveform event can only be bound to one external clock. Attempting to bind a second clock to an axis will simply replace the existing bind with the newer bind.

-Using MCL_IssSetClock will unbind the specified clock from all axes; the specified clock will not be unbound from waveform read/write events. If you have the Pixel clock bound to the X axis, Y axis, and waveform read and then use MCL_IssSetClock to set the pixel clock high, the Pixel clock will no longer be bound to the X axis or Y axis but will remain bound to the waveform read event.

-Setting the polarity of a particular clock using MCL_IssConfigurePolarity does not change the polarity of the particular clock's pulses generated through binding that clock to an axis or event. If you set the polarity of the Pixel clock to be high to low using MCL_IssConfigurePolarity and bind the Pixel clock to the X axis as a low to high pulse then calling MCL_PixelClock will generate a high to low pulse and reading the X axis will generate a low to high pulse.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Function Prototype: int MCL_IssConfigurePolarity(int clock, int mode, int handle)
Configures the polarity of the external clock pulses generated by MCL_(Pixel/Line/Frame/Aux)Clock().

Requirements: Firmware version > 0.
Firmware profile bit 1 equal to 1.
Image Scan Sync (ISS) option.

Parameters:
clock [in] Which clock to configure. (1=Pixel, 2=Line, 3=Frame, 4=Aux)
mode [in] Selects polarity of the clock.
2 = low to high pulse.
3 = high to low pulse.
handle [in] Specifies which Nano-Drive to communicate with.

Notes:
-Does not affect the polarity of clocks bound to axes or events by MCL_IssBindClockToAxis.

Return Value:
Returns MCL_SUCCESS or the appropriate error code.

Encoder

Function Prototype: double MCL_ReadEncoderZ(int handle)
Reads the current value of the Z encoder.
Requirements:
This function is specific to product ids 0x2000, 0x2100, 0x2401
Parameters:
handle [in] Specifies which device to communicate with.
Return Value:
Returns the current value of the Z encoder or the appropriate error code.

Function Prototype: int MCL_ResetEncoderZ(int handle)

Resets the Z encoder to 0.

Requirements:

This function is specific to product ids 0x2000, 0x2100, 0x2401

Parameters:

handle	[in]	Specifies which device to communicate with.

Return Value:

Returns MCL_SUCCESS or the appropriate error code.

Tip Tilt Z

Θ_C

Θ_B Θ_A

The standard device movement functions (MCL_SingleWriteN, MCL_SingleWriteN) may be used to control each actuator individually. The axis argument decides which actuator to move. $\Theta_A = 1$, $\Theta_B = 2$, $\Theta_C = 3$.

The actuators Θ_A, Θ_B , and Θ_C form an isosceles triangle. The width of the actuator triangle is the distance from Θ_B to Θ_A. The height of the actuator triangle is the minimum distance from Θ_C to the line segment formed by Θ_B and Θ_A.

A pure motion in Θ_X is achieved by moving Θ_A and Θ_B an equal distance and moving Θ_C the same distance in the opposite direction. A positive change in Θ_X is achieved by increasing Θ_C and decreasing Θ_A and Θ_B.

$$\Theta_A = \Theta_A - x$$
$$\Theta_B = \Theta_B - x$$
$$\Theta_C = \Theta_C + x$$

A pure motion in Θ_Y is achieved by keeping Θ_C constant and moving Θ_A and Θ_B equal distances in opposite directions. A positive change in Θ_Y is achieved by increasing Θ_B and decreasing Θ_A.

$$\Theta_A = \Theta_A - x$$
$$\Theta_B = \Theta_B + x$$
$$\Theta_C = \Theta_C$$

Function Prototype: int MCL_ThetaX(double milliradians, double *actual, int handle)

Calculates the required positioning of Θ_A, Θ_B, and Θ_C to form a Θ_X of 'milliradians' based on the dimensions of the actuator triangle. If achieving a Θ_X of 'milliradians' is not possible given ' the current position of the actuators the function will coerce 'milliradians' to the maximum possible(or minimum if 'milliradians' is negative) currently achievable Θ_X. 'actual' will be filled with the angle in milliradians produced by this function.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:

milliradians [in]	Desired Θ_X in milliradians.
actual [in/out]	Achieved Θ_X in milliradians.
handle [in]	Specifies which Nano-Drive to communicate with.

Return Value:

Returns 0 on success or the appropriate error code.

Function Prototype: int MCL_ThetaY(double milliradians, double *actual, int handle)
Calculates the required positioning of Θ_A and Θ_B to form a Θ_Y of 'milliradians' based on the dimensions of the actuator triangle. If achieving a Θ_Y of 'milliradians' is not possible given the current position of the actuators the function will coerce 'milliradians' to the maximum possible(or minimum if 'milliradians' is negative) currently achievable Θ_Y. 'actual' will be filled with the angle in milliradians produced by this function.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:

milliradians [in]	Desired Θ_Y in milliradians.
actual [in/out]	Achieved Θ_Y in milliradians.
handle [in]	Specifies which Nano-Drive to communicate with.

Return Value:
Returns 0 on success or the appropriate error code.

Function Prototype: int MCL_MoveZCenter(double position, double *actual, int handle);
Moves the center point (the midpoint of the perpendicular line segment from Θ_C to line segment $\Theta_A\Theta_B$) to a height of 'position' maintaining the current Θ_X and Θ_Y of the plane. If 'position' is not achievable without changing Θ_X and/or Θ_Y the function will move the center point to the maximum(or minimum) possible position such that Θ_X and Θ_Y remain unchanged. 'actual' will be filled with height of the center point produced by this function.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:

position [in]	Desired center point in microns.
actual [in/out]	Achieved center point in microns.
handle [in]	Specifies which Nano-Drive to communicate with.

Return Value:
Returns 0 on success or the appropriate error code.

Function Prototype: int MCL_LevelZ(double position, int handle)
Moves all the actuators to 'position' achieving a level plane.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:

position [in]	Z height to move to.
handle [in]	Specifies which Nano-Drive to communicate with.

Return Value:
Returns 0 on success or the appropriate error code.

Function Prototype: double MCL_TipTiltHeight(int handle)
Returns the height of the actuator triangle in millimeters.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:

handle [in]	Specifies which Nano-Drive to communicate with.

Return Value:
Returns height of the actuator triangle or the appropriate error code.

Function Prototype: double MCL_TipTiltWidth(int handle)
Returns the width of the actuator triangle in millimeters.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns width of the actuator triangle or the appropriate error code.

Function Prototype: int MCL_MinMaxThetaX(double *min, double *max, int handle)
Calculates the maximum and minimum achievable Θ_X angles in milliradians.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:
min [in/out] Set to the minimum Θ_X in milliradians.
max [in/out] Set to the maximum Θ_X in milliradians.
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns 0 or the appropriate error code.

Function Prototype: int MCL_MinMaxThetaY(double *min, double *max, int handle)
Calculates the maximum and minimum achievable Θ_Y angles in milliradians.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:
min [in/out] Set to the minimum Θ_Y in milliradians.
max [in/out] Set to the maximum Θ_Y in milliradians.
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns 0 or the appropriate error code.

Function Prototype: double MCL_GetTipTiltThetaX(int handle)
Returns the current Θ_X of the plane.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns the current Θ_X of the plane or the appropriate error code.

Function Prototype: double MCL_GetTipTiltThetaY(int handle)
Returns the current Θ_Y of the plane.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:
handl [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns the current Θ_Y of the plane or the appropriate error code.

Function Prototype: double MCL_GetTipTiltCenter(int handle)
Returns the current height of the center point (the midpoint of the perpendicular line segment from Θ_C to line segment $\Theta_A\Theta_B$).

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns the height of the center point or the appropriate error code.

Function Prototype: int MCL_CurrentMinMaxThetaX(double *min, double *max, int handle)
Given the position of aΘ_A, Θ_B, and Θ_Cs calculates the minimum and maximum achievable Θ_X.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:
min [in/out] Set to the minimum achievable Θ_X in milliradians.
max [in/out] Set to the maximum achievable Θ_X in milliradians.
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns 0 or the appropriate error code.

Function Prototype: int MCL_CurrentMinMaxThetaY(double *min, double *max, int handle)
Given the position of aΘ_A, Θ_B, and Θ_Cs calculates the minimum and maximum achievable Θ_Y.

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:
min [in/out] Set to the minimum achievable Θ_Y in milliradians.
max [in/out] Set to the maximum achievable Θ_Y in milliradians.
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns 0 or the appropriate error code.

Function Prototype: int MCL_CurrentMinMaxCenter(double *min, double *max, int handle)

Given the position of aΘ_A, Θ_B, and Θ_Cs calculates the minimum and maximum achievable center point (the midpoint of the perpendicular line segment from Θ_C to line segment $\Theta_A\Theta_B$).

Requirements: This function is specific to product ids 0x1253 and 0x2253.

Parameters:
min [in/out] Set to the minimum achievable center height in microns.
max [in/out] Set to the minimum achievable center height in microns.
handle [in] Specifies which Nano-Drive to communicate with.

Return Value:
Returns 0 or the appropriate error code.

Error Codes & Structures

MCL_SUCCESS 0
Task has been completed successfully.

MCL_GENERAL_ERROR -1
These errors generally occur due to an internal sanity check failing.

MCL_DEV_ERROR -2
A problem occurred when transferring data to the Nano-Drive. It is likely that the Nano-Drive will have to be power cycled to correct these errors.

MCL_DEV_NOT_ATTACHED -3
The Nano-Drive cannot complete the task because it is not attached.

MCL_USAGE_ERROR -4
Using a function from the library which the Nano-Drive does not support causes these errors.

MCL_DEV_NOT_READY -5
The Nano-Drive is currently completing or waiting to complete another task.

MCL_ARGUMENT_ERROR -6
An argument is out of range or a required pointer is equal to NULL.

MCL_INVALID_AXIS -7
Attempting an operation on an axis that does not exist in the Nano-Drive.

MCL_INVALID_HANDLE -8
The handle is not valid. Or at least is not valid in this instance of the DLL.

```
struct ProductInformation {
        unsigned char  axis_bitmap;
                xx65  4321
                6:      D axes
                5:      Z encoder
                4:      Aux axis
                3:      Z axis
                2:      Y axis
                1:      X axis
                1 indicates a valid axis, 0 an invalid axis
        short ADC_resolution;
                ADC resolution in # of bits
        short DAC_resolution;
                DAC resolution in # of bits
        short Product_id;
                USB 1.1
                0x1000 Nano-Drive Single Axis
                0x1030 Nano-Drive Three Axis
                0x1230 Nano-Drive 20 bit Three Axis
                0x2000 Nano-Gauge
                0x1253 Nano-Drive 20 bit Tip/Tilt Z
                USB 2.0
                0x2001 Nano-Drive Single Axis
                0x2003 Nano-Drive Three Axis
                0x2053 Nano-Drive 16 bit Tip/Tilt Z
                0x2004 Nano-Drive Four Axis
                0x2201 Nano-Drive 20 bit Single Axis
                0x2203 Nano-Drive 20 bit Three Axis
                0x2253 Nano-Drive 20 bit Tip/Tilt Z
                0x2100 Nano-Gauge
                0x2401 C-Focus
        short FirmwareVersion;
        short FirmwareProfile;
}
```