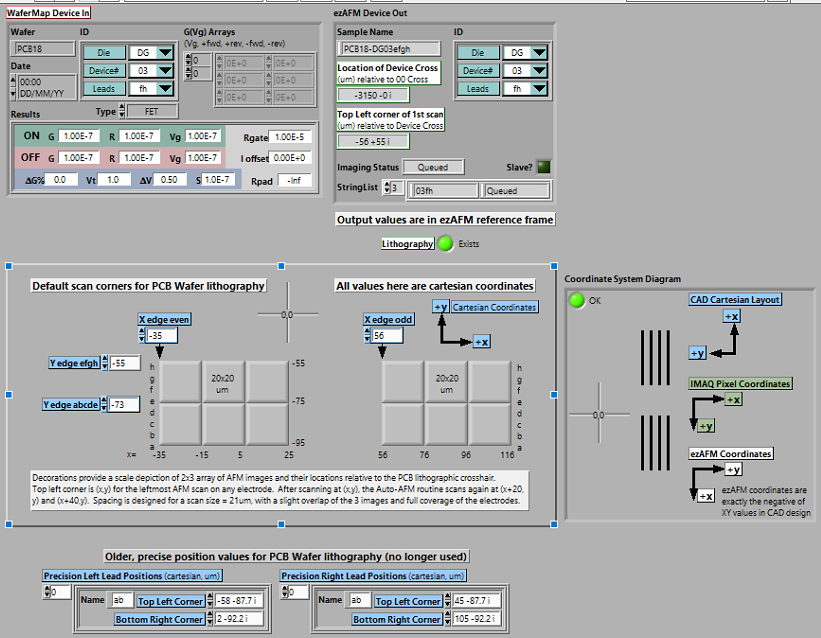
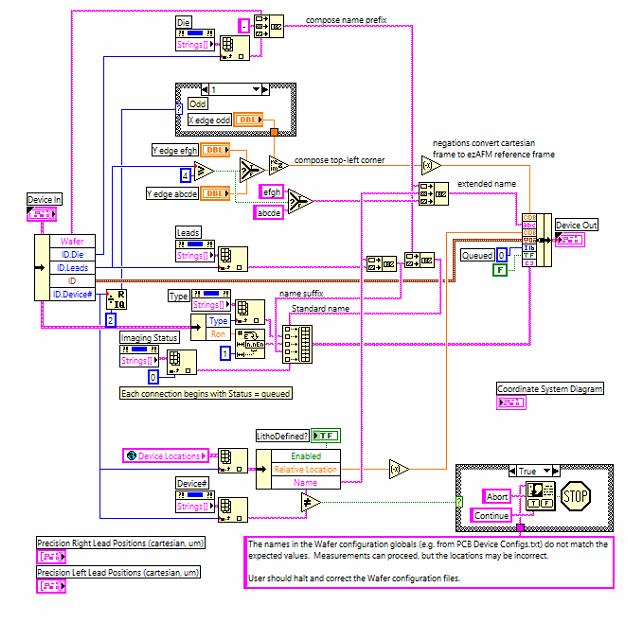
### ezAFM Scan Settings from WaferDevice VI

* This VI is actually quite simple; it essentially takes in the details of a device and outputs a summary of those details along with a little extra such as the locations of the device cross and the top left of the 1st scan

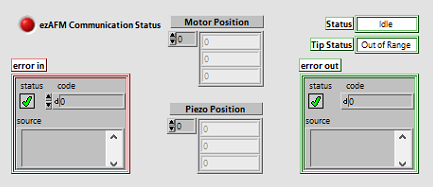


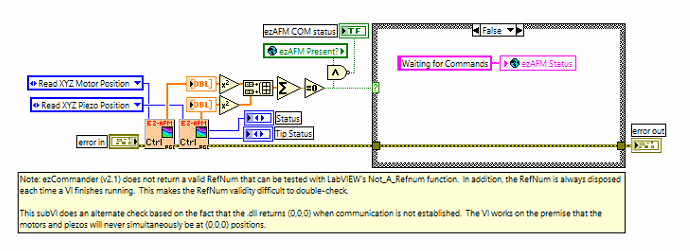


* Adopting typical Cartesian coordinates with the regular orientation of the die numbers right-side-up
* In the block diagram above, we see that the Device In cluster is unbundled and the gist of the program is to handle each component
  + Note that Die, Leads, and Device# are all values within a ring control
    - When we unbundle them, we obtain the index value which we then use to index the Property Node containing the array Strings[] to obtain the actual string itself
    - This is a little LabView tip, whenever we want to obtain the item name of a given Ring control as a string, we need to use the Property Node set to ‘Strings[]’ and index the correct string using the value of the Ring control
  + ‘Wafer’ and ‘ID.Die’ are both used to generate a name for the device using string concatenation
  + ‘ID.Leads’ is used to determine the y-coordinate of the scan spot
    - Using an inequality operator, if ID.Leads >= 4 then we can assume that all the possible leads belong to Spot B1 or B2
    - If ID.Leads < 4, then the leads belong to Spot A1 or A2
  + ‘ID’ is a cluster and is simply passed through to the output Device Out immediately
    - ‘ID’ is just a cluster of the Device#, Leads, and Die
    - It makes sense to just pass this value immediately through since the purpose of this VI is to not modify information but to just summarize and output the desired information
  + ‘ID.Device#’ is used to determine the x-coordinate of the scan spot
    - Using a case structure for even/odd indices since the ID.Device# comes from a ring control
      * if the index is even, then the device is even and the scan spot is A1 or B1
      * If the index is odd, then the device is odd and the scan spot is A2 or B2
  + In summary, the program judges whether the leads and device# belong to Spot A1, B1, A2, or B2 and outputs the coordinates
    - The coordinates for the scan spot come simply from the front panel controls
    - The coordinates of the crosshair with respect to the 00 crosshair comes from a global variable which is an array of clusters containing the name and coordinates of each device#

### ezAFM Communication Test

* This VI checks if the ezCommander DLL is connected properly using the given methods from the ezCommander.dll
  + NOTE: DLL stands for dynamic linked library, which provides a way for a program to access external code. In this case, it allows LabView to access the same commands that the ezAFM software (ezCommander) uses



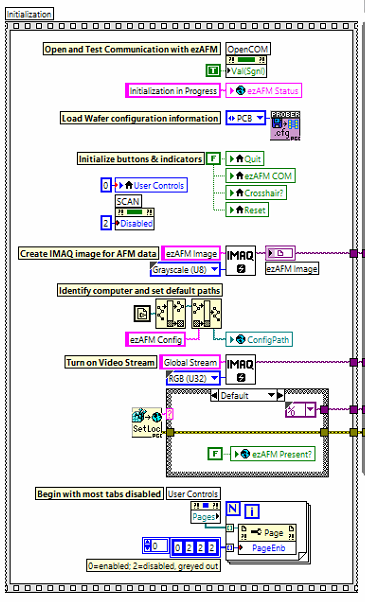


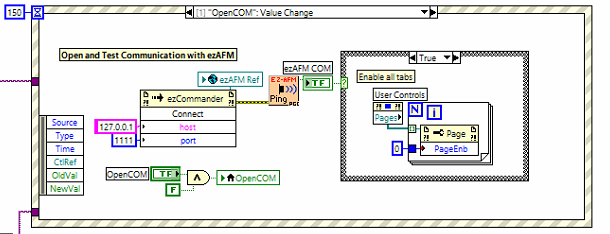
* In the block diagram, we see two calls to the ezAFM Control VI, one to read the XYZ Motor and the other to read the XYZ Piezo positions
  + The values are outputted from the ezAFM Control VI as 1D numerical array ‘buffers’ with 3 items; ‘buffers’ are essentially raw data from the ezAFM
* I’m not sure why but the outputs are squared and then combined into one large array and then we test to see if the summation is equal to zero
  + Why can’t we just test to see if the sum of the outputs are both zero?

I think it’s just a matter of style

* The VI takes advantage of the fact that (0, 0, 0) is outputted for both positions if the ezAFM is NOT connected
  + If the numbers are both zero, then an error statement pops up on the front panel, asking the user to check the port 1111 connection
  + Else, we set the global variable ‘ezAFM Status’ to ‘Waiting for Commands’

### ezAFM Automated Imaging 3.3 (Initialization)

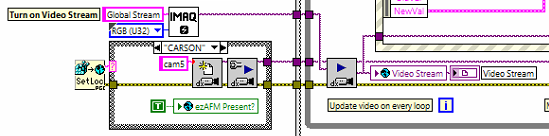


* The block diagram for this VI contains a flat sequence dedicated for initialization of certain variables
  + Firstly, we forcibly change (write) the value of the OpenCOM Button to be true using a Property Node set on ‘Value(Signalling)’; this operation essentially simulates the user pressing the button as the Signalling aspect registers as an event (value change) in LabView
    - The value change causes the event to run, which essentially just goes through with the whole connecting the ezCommander and enabling all the tabs
  + Another thing is that during the flat-sequence, we also write the global variable ‘ezAFM Status’ to be ‘Initialization in Progress’
  + Next, we load the wafer configuration information into a ‘Read Config Info’ VI using an Enum constant
    - Looking into this ‘Read Config Info’ VI, there are some file location errors and I’m not sure what it does
  + Next, we initialize all the buttons to be false by writing a false constant to their local variables
  + Next, we write the value of 0 into the local variable for the Tab Control named ‘User Controls’
    - I think this makes it so that the first tab is opened upon running the VI
  + Next, we also set the Scan button to be disabled using a Property Node with the Disabled feature where we write in the index 2 for ‘disabled and grayed out’
  + Next, we initialize an IMAQ image for AFM data which uses the ‘IMAQ Create’ VI, which takes in the image name and the image type, in this case: Grayscale(U8)
  + Next, we obtain the ezAFM Automated Imaging 3.3 file path, obtain only the file path up to the last component and then pass that stripped file path into a ‘build path’ item where we add on a string constant ‘ezAFM Config’ and wire the result into a global variable ‘ConfigPath’
    - I think the point of this is to just get the file/folder directory in which the ezAFM Automated Imaging VI is in and create a new folder named ‘ezAFM Config’ to do stuff
  + Next, we again initialize an IMAQ image this time for the camera with the name ‘Global Stream’ and image type: RGB(U32)
    - We pass the IMAQ image outputs into the while loop
  + Next, we use a ‘Set Datalog Location’ VI which outputs the name of the computer, in this case ‘CARSON’
    - We wire this name into the case structure and condition ‘CARSON’ so that we begin initializing the camera video
      * In this case, we write a string constant ‘cam5’ into the what seems to be an NI\_VISION\_ACQUISITION Open Camera VI, this involves a Session In and Session Out data, which is some IMAQ object
      * Then we use another vision acquisition VI named ‘Configure Grab’ which takes in the Session Out from earlier and outputs another Session Out which goes into the while loop
      * We also write the global variable ‘ezAFM Present?’ to true
    - If not ‘CARSON’, then end the session and write the global variable ‘ezAFM Present?’ to false
  + Finally, we initialization the Tab Control so that only the first tab is available and the rest are disabled and grayed out
    - We do so by using a series of 2 Property Nodes, one with ‘Pages’ which returns an array of references to each page. We wire this array into the reference input for another Property Node with ‘PageEnb’ which takes in a numerical 1D array of states for the tabs (0 = enabled, 1 = disabled, and 2 = disabled and grayed out)
    - We write the ‘Pages’ output into a for-loop since we want to iterate through the array of references
      * Moreover, we also wire a numerical constant 1D array [0, 2, 2, 2] whose elements determine the state of the tabs
  + That is all for the initialization

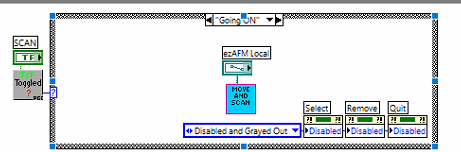
### 

### ezAFM Automated Imaging 3.3 (Body)

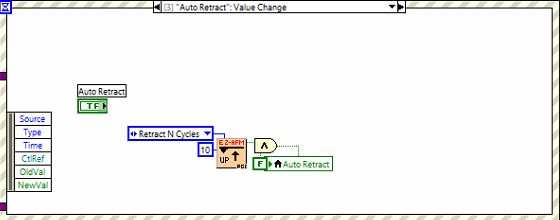
* Now looking at the main while loop of the VI



* We first pass in the Session Out from the ‘Configure Grab’ VI into another sort of ‘Grab’ VI for Image Acquisition; It takes in both the Session Out and the IMAQ Image object named ‘Global Stream’
  + I think the logic might be something like create an IMAQ object to write data on and initialize a session for writing data
* We then pass the Session Out into the global variable and indicator both named ‘Video Stream’
  + Looking at how we initialized the AFM image and the Video Stream image as IMAQ Objects, we also use Tab Control named ‘Image Panel’ to switch between viewing the two on the front panel
* Also within the while-loop, we’re constantly updating the status indicator with the global variable ‘ezAFM Status’
  + This makes sense so that we are tracking the status in real-time
  + Moreover, the string constant ‘The VI is not running’ is also wired outside the while-loop into the status indicator when the while-loop ends
* At the top of the while-loop we have a Scan button conditioned into a case structure
  + This aspect of the block diagram doesn’t seem to work



* Now looking at the event structure within the while-loop where everything goes down
* **[0] Timeout**
  + Nothing happens in this event
* **[1] ‘OpenCOM’: Value Change**
  + This references the button ‘OpenCOM’, which is automatically set to true at the beginning of initialization
  + Simply call the Invoke Node with the ezCommander.dll invoking the method Connect which takes in a string constant for the IP and a number for the port (1111)
    - We pass in the global variable ‘ezAFM Ref’ for the reference in the Invoke Node
  + Then we call the ezAFM Communication Test VI and wire its output to a boolean case structure and indicator
    - If true, then we enable all tabs on the front panel
      * Using a Property Node set on Pages which outputs the array of references for each tab
      * Then we pass in this array into a for-loop. Within the for-loop, we have another Property Node set on PageEnb which takes in the number indicating the state (0 = enabled), so we pass in zero and write it into the Property Node for each element within the Pages array
    - If false, then we do nothing
* **[2] ‘XY Motor Controls’: Value Change**
  + Here, XY Motor Controls represents a cluster of all the navigation buttons plus the step approach/retract buttons
    - The cluster control item is conveniently located on the left aside from the rest of the code for the user to double-click on and identify on the front panel
  + We use the ezAFM Control VI to send commands to the XY Motor depending on what button’s value has changed
    - Using the event’s data node, the NewVal terminal is wired into the ezAFM Control VI
      * NewVal is a cluster of the XY Motor Controls, except this time one of the buttons is set to True
      * Within the ezAFM Control, the cluster will be searched and the true value will trigger some sort of operation
    - The step size and speed numeric controls are also wired into the VI, along with the Enum constant set on ‘XYZ Motions’
  + Also within this event, the Image Out object coming from the ‘IMAQdx Grab’ VI is wired in and connected to the ezAFM Pattern Matching VI
    - This VI outputs the ‘Found?’ cluster which involves 6 booleans
      * ‘Crosshair Found?’, ‘Cantilever Found?’, ‘Both Found?’, ‘00 Found?’, ‘03 Found?’, and ‘00 or 03?’
    - This VI also outputs a ‘Both Found?’ Boolean, which is then wired into an indicator and into the boolean case structure conditional terminal
      * Here “Both” involves the crosshair lithography and the cantilever
    - We unbundle the ‘Found?’ cluster into ‘00 or 03?’ and ‘03 Found?’
      * Wiring the ‘00 or 03?’ into an indicator and the ‘03 Found?’ into the same case structure
    - Based on the ‘Both Found?’ Boolean value, we prompt the user via writing a string constant into the global variable ‘ezAFM Status’
      * If true, then we prompt the user based on the ‘03 Found?’ Boolean value
        + If true, then we write the string constant ‘hit Set Position button to confirm the position at 03 electrodes’
        + If false, then we write the string constant ‘hit Set Position button to confirm the position at 00 electrodes’
      * If false, then this means the crosshairs were not found and we write ‘Continue moving until 00 or 03 lithography comes into view’ into the global variable ‘ezAFM Status’
        + Essentially do nothing when the crosshair is not in view
  + In summary, this event takes in the XY Motor Controls cluster and performs whatever operation corresponding to the value change of the button. Moreover, it calls the ezAFM Pattern Matching VI to detect whether the crosshair, cantilever, and 00 / 03 lithography is within view
    - If so, then we prompt the user to click the Set Position button to confirm that either 00 or 03 are definitely in view
* **[3] ‘Auto Retract’: Value Change**
  + Located conveniently on the left aside from the other items is the Auto Retract button control itself
  + In this event, we just call the Auto-Retract with Pattern Recognition VI
    - Takes in a Mode (Enum constant) and N number of cycles to retract
    - Outputs a Done? Boolean value
  + Looking into the Auto-Retract with Pattern Recognition VI
    - Everything performs within a while-loop conditioned on the Done? Control, which is true when either the # of iterations is equal to N cycles or when the pattern recognition condition is achieved
      * The condition being when the cantilever/crosshair is NOT found, so stops when the crosshair is too blurry basically
    - We decide to wait for either N cycles or based on the pattern recognition depending on the Mode (Enum constant)
      * The mode is wired into a string case structure
        + If ‘Retract N Cycles’ then wire a false Boolean constant outwards since we’re not using pattern recognition
        + If ‘Retract to Out-of-Focus’ then we call the ezAFM Pattern recognition VI and wire the negation of the ‘Both Found?’ Boolean output outwards
        + If ‘Retract to In-Focus’ then we again cal the ezAFM Pattern Recognition VI and directly wire the ‘Both Found?’ Boolean output outwards
    - Within the while-loop, we continuously call the ezAFM Control VI
      * Wiring the mode ‘XYZ Motions’ and an input boolean array where only the second element is true. The function is just set up this way so that the second element corresponds to the retract move
    - Outside the while-loop, we write the string constant ‘Auto-retracting ezAFM Tip’ to the global variable ‘ezAFM Status’



* + There’s a bit of LabView programming intuition going on here
  + The output of the Auto-Retract with Pattern Recognition VI is wired into an And operator alongside a boolean False constant, whose result is then written into a local variable for the Auto Retract button
  + Interestingly, the output of the Auto-Retract VI is ‘Done?’, so true if we’ve finished retracting and false otherwise
    - However, the And operator will always render its result to be False due to the False constant
    - Why would we do this? For the purpose of taking advantage of the data flow
      * This is Prof. Collins way of using the Not operator like how I would to indicate the Auto-Retract button to switch back to False
  + My way of approaching the event cases involved placing the button controls outside of the event case, and then using local variables for whatever references necessary
    - This way somewhat automatically resets the Auto-Retract button to false after the event finishes, however I think Prof. Collin’s method is better
      * **[4] ‘Search’: Value Change**
  + The ‘Search’ button is the same as the ‘Set Position’ button
  + Calls the ezAFM Pattern Matching VI, wiring the ‘Found?’ cluster output into an Unbundle by Name operator and wiring the ‘Both Found?’ Boolean output into the conditional terminal for a case structure
    - Unbundles the ‘00 or 03?’ and ‘03 found?’ items and passes them into the case structure
  + Within the case structure (terminal condition is from ‘Both Found?’)
    - If true, then we wire the ‘00 or 03?’ Boolean item into the conditional terminal of another case structure
      * Within this case structure
        + We wire the ‘03 found?’ Boolean item into a Select operator

If true, then we wire a string constant ‘03’

If false, then we wire a string constant ‘00’

* + - * + We then concatenate these string constants to form Two Button Dialog for the user

‘Everything looks great! (03) / (00) electrodes, or choose *Cancel* to reposition the sample

* + - * + The two buttons are either *Confirm* or *Cancel*

If *Confirm*, then we wire the ‘03 found?’ Boolean item into a case structure

If true, then write the Alignment at 03 button control to be true via Property Node set on Value(Signalling)

If false, then write the Alignment at 00 button control to be true via Property Node… etc

In both cases, we also wire in some numeric data from a helper VI named ‘Convert ezAFM pixels to absolute position’ into a global variable named either ‘00 crosshair’ or ‘03 crosshair’

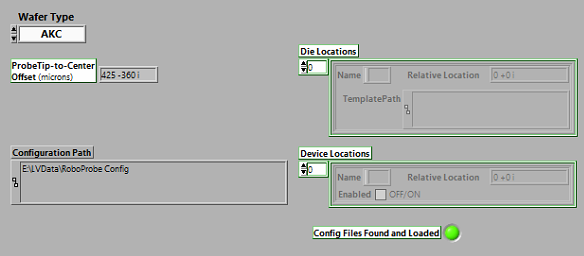
If *Cancel*, then do nothing

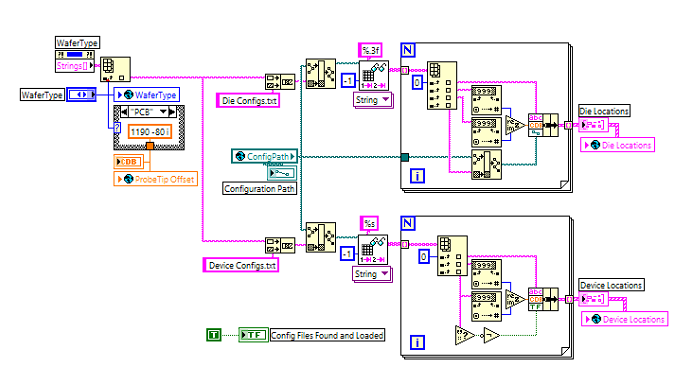
* + - If false, then we wire in the ‘Found?’ cluster and unbundle the ‘Crosshair?’ Boolean item, wiring the output into the conditional terminal of a case structure
      * If true, then we write a string constant indicating that the cantilever cannot be located due to several reasons into a One Button Dialog
      * If false, then we write a string constant indicating that the crosshair lithography was not found and to refocus / reposition the sample into a One Button Dialog
  + In summary, this event takes in the ‘Image Out’ object from the ‘IMAQdx Grab’ VI and wires it into the ezAFM Pattern Matching VI
    - Then it takes the results of the image recognition and based on whether something was found properly, it prompts the user to either confirm the finding or to refocus / reposition
      * Prompts the user via One or Two Button Dialog
      * If the user does confirm the location then the value of ‘00 Aligned’ or ‘03 Aligned’ changes and the global variables ‘03 Crosshair’ or ‘00 Crosshair’ are set to true
* **[5] ‘03 Aligned’, ‘00 Aligned’: Value Change**
  + This event is not like an event triggered directly by an item from the front panel
  + When we click the ‘Search’ or ‘Set Position’ button, we trigger an event case that may change the value of ‘03 Aligned’ and ‘00 Aligned’ which would then trigger this event
  + This event checks if both ‘00 Aligned’ and ‘03 Aligned’ are true
    - If true, then we reference the global variables ‘00 Crosshair’ and ‘03 Crosshair’ and compute the angle between them.
      * We write the angle as a complex vector into the global variable ‘SampleAngle’ and as a number in degrees into the Rotation indicator
    - Secondly, we reference the Connections array to see if it’s empty
      * If it’s empty, then update the global variable ‘ezAFM Status’ to be ‘Waiting for user to select connections
      * If it’s not empty, then update the global variable ‘ezAFM Status’ to be ‘Ready to begin AFM Scanning’
    - Thirdly, we enable the Scan tab via a Property Node set on Disabled and we write 0 into it for 0 = enabled
  + In summary, this event is a neat trigger for when the values of ‘00 Aligned’ and ‘03 Aligned’ are changed
    - If they are changed then we immediately check if both are true which then enables the Scan tab on the main menu

**[6] ‘Reset’: Value Change**

* + Interestingly this event involves writing the Reset button control into an And operator alongside a boolean false constant, wiring the result into the Reset button’s local variable
    - I’m not sure what the benefit of doing this is…Does wiring the control data make it so that we reset the button to false after the control’s done doing whatever it’s doing?
  + Also write the boolean false constant into the ‘00 Aligned’ and ‘03 Aligned’ indicators (original, not local variable)
  + Also write zero complex vector (0 + 0i) into global variables ‘SampleAngle’, ‘00 Crosshair’, and ‘03 Crosshair’ and write 0 into the local variable for Rotation
  + In summary, this VI simply sets all the sample parameters to zero and boolean indicators to false
* **[7] ‘Auto Land’: Value Change**
  + The Auto Land button control is conveniently located to the left aside from the rest
  + We wire the NewVal from the event’s data node into the conditional terminal for a case structure
    - If true, then call the ezAFM Control VI with the mode set on ‘Begin AutoLand’
    - If false, then call the Auto-Retract with Pattern Recognition VI with the mode set on ‘Retract N Cycles’ and wiring 1 for N
  + In summary, this VI uses the event’s data node to wire the NewVal into a case structure. If true, then we begin auto-land, if false, then we retract once.
* **[8] ‘Select’, ‘Remove’: Value Change**
  + In this event, we wire the Select control into the conditional terminal of a case structure
    - If true, then we call the SimpleMap VI and wire the Connections output into an indicator (Connections is a 1D array of a large cluster of clusters)
    - If false, then do nothing
  + In this event, we wire the Remove control into the conditional terminal of a case structure
    - If true, then we prompt the user to confirm if they want to remove the connections via Two Button Dialog wired into a case structure
      * If *Confirm*, then we wire the connections into the case structure and use the Delete From Array operator and wire the output into the Connections local variable
        + We wire the DeviceList (multicolumn ListBox object) into the case structure and use it as the indices we’d like to remove from Connections
        + For some reason we also include the ezAFM Image indicator as well but we don’t wire anything to it
      * If *Cancel*, then we do nothing
    - If false, then we do nothing
  + Also in this event, we include the Scanning Device# control but we don’t wire anything to it
  + Also, we wire a boolean true constant into a Property Node set at Value(Signalling) for the Update List control
    - The Update List control shows up as an indicator labeled as ‘Device List needs updating’
* **[9] ‘Update List’: Value Change**
  + The Update List indicator is conveniently located to the left aside from the rest
    - We trigger this event within the previous event: [8] ‘Select’, ‘Remove’
    - This is rather intuitive because rather than having a unique case for updating the list when it comes to selecting/removing connections, the program simply updates the list every time either action occurs
  + In this event, we use a local variable for Connections wired into the ‘ezAFM Build ScanList from Connections’ VI
    - This auxiliary VI outputs the item names as a 2D string array which we wire into a Property Node for the DeviceList set on ItemNames
      * The ItemNames takes in the 2D string array and displays them on the multicolumn table on the front panel
    - Using the same Property Node, we set another property on NumRows which takes in the number of visible rows for the ListBox on the front panel
      * We wire this input from the ‘minimum’ output of the Max & Min operator which just takes in two numbers and outputs the maximum and minimum
        + The two inputs are the number constant 18 and the length of the Connections list
        + Interestingly, the Property Node also has VScrollVis property which if we wire a true value, enables the vertical scrollbar to be visible. The true value comes from the comparison operator if the number of rows is greater than 18
* **[10] ‘ezAFM Scan Settings’: Value Change**
  + This event uses the ezAFM Scan Settings, which is a cluster of controls and wires them into the global variable ‘ezAFM Scan Settings’
  + Any change to any of the control parameters triggers this event and updates the global variable
    - The global variable is called each time we initialize a scan
* **[11] ‘Image Pane’, ‘ezAFM Image to Load’: Value Change**
  + The IMAQ Image object we created and named for the ezAFM Image is wired into this event
    - This object is wired into a IMAQ ReadFile operator which takes in the image and a file path
      * The file path comes from a control on the front panel; if it is empty, then we just wire an empty file path into the operator, else we wire in the path itself
      * The operator outputs an image which we write into ezAFM Image object via a local variable
* **[12] ‘Format’: Value Change**
  + Format is a ring control on the front panel, where we select a file type (.nmi, .jpg, .bmp, and .png)
  + We wire the Format control into a case structure for each choice and then we wire a string constant of the file type outside the case structure into a Property Node for the ezAFM file path control set on BrowseOptions:MatchPattern
* **[13] ‘WaferType’: Value Change**
  + **This event has the WaferType enum control inside except it’s wired to nothing**
    - **The enum has items ‘AKC’ and ‘PCB’**
  + **This event wires the data node NewVal item into the Read Config Info VI which takes in the WaferType as input**

### Read Config Info VI





* This VI takes in a Wafer Type via enum control
* First we use a Property Node set on Strings[] for the WaferType enum to index and output the string constant of the wafer type ie ‘PCB’ or ‘AKC’
  + We use this string constant to build a file path which would then be read into a Read Delimited Spreadsheet VI which then takes the text file and outputs a 2D array of data (strings, numbers, etc.)
  + We then wire the 2D array into a for-loop. Now we’re working with 1D arrays (of strings)
    - We index the elements of the arrays, converting the strings to numbers and then we bundle them into a cluster. The cluster is then wired outside the for-loop and into the Die Locations indicator and the global variable ‘Die Locations’
      * The cluster is composed of Name (string), Relative Location (complex vector), and TemplatePath (file path)
    - It appears that the 1D arrays consist of a Name, x-coordinate, y-coordinate, and file path, respectively
      * The file path is just the ‘relative path’ (the ending component) so we use the Build Path operator in conjunction with the global variable ‘ConfigPath’, which shows up as an indicator on the front panel
* Based on the wafer type, we wire a complex vector into the ProbeTip Offset indicator
  + 425 - 360i for AKC
  + 1190 - 80i for PCB
  + I’m not sure what these numbers mean
* Secondly, we perform a similar operation of building a file path using the wafer type string constant and wiring that file path into the Read Delimited Spreadsheet VI
  + The output 2D array is then similarly wired into a for-loop and indexed
    - The indices of this are Name (string), Relative Location (complex vector), and Enabled (boolean)
    - Again we convert the elements from strings to numbers before bundling them into a cluster
    - The ‘Enabled’ item is determined if the file path is empty or not. If empty, then Enabled is false; if not empty, then Enabled is true
* This VI currently does not run properly due to some file path errors

### 9/13 Notes

* Currently the scanning feature on ‘ezAFM Automated Imaging 3.3’ VI is outside the main event structure
  + The VI waits until the user presses ‘Launch Auto-AFM Scanning’
  + Once it’s pressed, the control is wired into some ‘Toggled’ VI which I’m not sure what it does. The outgoing wire of this ‘Toggled’ VI is then wired into a case structure.
  + Within the case structure
    - We wire the ‘ezAFM Local’ file path into the ‘Move and Scan Device’ VI
    - We wire an enum representing the ‘Enabled’, ‘Disabled’, and ‘Disabled and Greyed Out’ data values for the main Tab Control

### ‘Move and Scan Device’ VI

* Input Terminals
  + ‘Enable Pattern Recognition?’ Boolean control
  + ‘Rescan for Tubes?’ Boolean control
  + ‘Adjust for Angle?’ Boolean control
  + ‘ezAFM Image Desired Location’ File Path control
  + Error in
* Output Terminals
  + ‘Device with tubes’ array indicator
    - Array of clusters (each cluster is a subset of Roboprobe ID Control)
* The input controls are wired into a nested while-loop, in addition to 2 global variables: ‘ezAFM Scan Settings’ and ‘Device to Scan Array’
  + Everything takes place here

