A Report on Summer Work

The LabVIEW Program

Preliminary Steps

I began my work this summer by helping Ann Marie Carroll to move furniture, but soon we ran out of things to do and she suggested that I study LabVIEW. When Jim McIntyre returned from his vacation, he talked with us and assigned Ann Marie to fuse fibers and me to write a program that would read a voltage from a thermocouple, convert it to temperature, and control a heater and a pump.

I still had to learn more about LabVIEW before beginning to write the program, so I continued to study the language using Emily Briere's LabVIEW textbook. But at some point in July I felt that I was ready to begin my attempt.

Most of the files that I used or made in the program development are in a folder on Jim's computer titled "Ben Willis" (C:\Documents and Settings\Jim\My Documents\Ben Willis). Some of the early ones may not be there, such as Emily's preliminary program.

Emily Briere had made a preliminary program file for me to start with, and I added controls and functions to it. My modified version is available as "Copy of Heater Control.vi." She put in a comparison function wired to a case structure with Boolean indicators labeled "Heater On" and "Turn Off Power." I had been informed that AIn-1.vi would read the voltage from the DAQ card, so I copied its block diagram into my block diagram. I found a function called "Convert Thermocouple Reading.vi" that had six input terminals: temperature units, thermocouple voltage and thermocouple type, cold junction compensation voltage and cold junction compensation sensor, and type of excitation. Its output terminal was called linearized temperature. I wired it to AIn-1.vi because I suspected that it would output the temperature without requiring us to use the mathscript node.

Brendan Pratt has been a great help to me on this job. I consider him my professor of LabVIEW. He brought in an oscilloscope and wired it to the DAQ card so we could measure a voltage. He made a file I titled "Communicating with DAQ Card.vi" that displayed a voltage from an oscilloscope on a waveform chart, but it did not behave as he expected. Because I did not know how to use an oscilloscope, I preferred to focus on how the program would respond to a signal after measuring it.

Writing Strings to the netBooter

The first problem that I confronted was how to communicate with the netBooter that operates the switches and outlets that would supply or cut off electricity to the heater and pump. As shown in "VISA Session.vi," I searched for the netBooter using the function "VISA Find Resource." The find list reported two resources, ASRL1::INSTR and ASRL10::INSTR. At the time, I had not correctly identified the serial cable that carries signals between the computer and the netBooter. I was baffled until Brendan determined that the computer was communicating with the netBooter through the COM1 port. He found an example called "Basic Serial Write and Read.vi" that we could use to write the same strings to the netBooter as we would use in HyperTerminal. This file is in my folder and is titled "Bens_Serial.vi."

On the front panel of this VI is a "string to write" text box. I entered each of the strings that we used with HyperTerminal into this text box, and the netBooter responded correctly to almost all of them. I described the commands in a WordPad file called "netBooter Strings," in which I wrote directions for the program at the time.

"These are the commands that work with the String to Write function in LabVIEW. At the end of each command you must put a termination character by pressing the enter key. Otherwise the command will not work.

Turn on all power outlets: **ps 1**
Turn on outlet 1: **pset 1 1**
Turn on outlet 2: **pset 2 1**
Turn off outlet 1: **pset 1 0**
Turn off outlet 2: **pset 2 0**
Turn all outlets off: **ps 0**

"Below Ann Marie describes the set of commands used in HyperTerminal. We don't have our outlets assigned to a group, so I did not try the commands for a group. The help command in LabVIEW does show some commands, but it also generates a rapid series of error messages. The reboot command causes the netBooter and the DAQ card to switch on and off rapidly. I'm guessing that is not a good idea.

"'In HyperTerminal the commands are:
help shows a list of helpful commands.
pset n x - turns power outlet on/off. Where “n” is outlet #; “x” is either “0” (off) or “1” (on).
rb n - reboots outlet n.
grb n - reboots outlets assigned to group “n”
ps 1 - turns all power outlets on.
ps 0 - turns all power outlets off.
gps n 0 - turns off outlets assigned to the group n.
gps n 1 - turns on outlets assigned to the group n.'"

However, the program needed to use more than one string. It needed to turn on all outlets and then turn them off separately. When more than one string was in the "string to write" text box, the commands did not work. I wanted to find a way to select which strings to write from the various strings we needed. I copied the block diagram from Basic Serial Write and Read to another file, "Bending.vi," and put the functions I had wired together to obtain the temperature there also. I hoped that the case structure would be able to select which strings to write based on the water temperature, so I wired "Greater or Equal?" to the case structure. This was a mistaken idea of course; what is wired to the selector terminal determines which case executes. The vertical toggle switch labeled "write" wired to the case structure on the left selects the case for this case structure.

On the "Strings and Timing.vi" block diagram you can see other ideas that would not work. The pink boxes containing strings are string constants from the String subpalette under Programming on the Functions palette. One of my ideas was to wire several string constants to a "Build Array" function and use the "Line Number" numeric control to select which array element to send to the "Write Buffer" terminal of "VISA Write" in the case structure.

A more promising idea (I still wonder if it would work) was to use the "Pick Line" function to select a line from a multi-line string to send to the Write Buffer terminal. I tried to use an Enumerated Type control to test my idea. From reading the help, I thought that Pick Line would select a line based on a numeric input and append that line to an empty string, which I imagined as something like an envelope that holds any string that needs to be mailed. Perhaps that is not what an empty string is. In any case, I did not succeed at the time.

At this point, Brendan came in to save the day. He knew we could use Basic Serial Write and Read.vi as a subVI and duplicate it for each string we needed. He created a file called smiley.vi, which is on the desktop, that we would use to turn on the heater and pump. It is a subVI with an icon of a smiley face that can be put in another VI so that it doesn't clutter the block diagram. It has the same controls and functions on the front panel and block diagram as Basic Serial Write and Read. In the "string to write" text box is the string "ps 1" which turns all outlets on. Brendan also made a similar file, frowny.vi, which is for turning off outlets. I made one copy for each outlet, calling them "Outlet 1 Off.vi" and "Outlet 2 Off.vi." The strings in them do what the file names describe. I also made other copies of these files, but we don't need to use them because we don't plan to use all the possible commands.

For a few days I was annoyed that whenever I opened a program that contained smiley, Outlet 1 Off, or Outlet 2 Off, the strings they should have had in the "string to write" text boxes

were not there. Instead, the strings read, *IDN?\r\n. Finally I discovered that the shortcut menu for the text box has the submenu Data Operations, where there is an option to "Make Current Value Default." After that the strings were always right. "Make Current Value Default" is also available on the shortcut menus for numeric and Boolean controls, and there is a very helpful item called "Reinitialize Values to Default" in the Edit menu on the front panel and in the shortcut menus for controls.

Brendan made a program, which I saved as "Simple Heater Control.vi" and copied as "Pre-Heat Control SP1.vi" and "Bending Control SP1.vi." I made the copies because Jim wanted to include two heating situations, pre-heating and bending. (As I look at this program, I wonder if keeping the flat sequence structure with the frames that divide the process into steps would have enabled later versions of the program to turn off the outlets separately as they were supposed to do.) We used the program to measure a voltage with the oscilloscope and turn the outlets on if it was high and off if it was low.

The Button Challenge

But Jim wanted several buttons on the front panel. He wanted start buttons for pre-heating and bending, stop buttons to turn off the heater and the pump, and an "End Bending" button. One of the stop buttons should turn off the pump five minutes after the heater, another two minutes after the heater, and the emergency stop should shut off both at the same time. I wanted to add those buttons, and I thought it would be a simple matter.

First I wanted to figure out how to make buttons that would turn the outlets on and off directly, with a delay after Outlet 1 shut off before Outlet 2 shut off, and indicators to show that the strings had been written to the netBooter. Brendan accomplished this feat in "Adding On and Off Indicators.vi." He developed more complicated cases for turning on the heater and the pump so that the indicator lights on the front panel would respond to the true case along with the "smiley.vi," "Outlet 1 Off.vi," and "Outlet 2 Off.vi" files. We had also added a time delay using a flat sequence structure and a for loop in each case structure so the processes would execute only once per iteration of the while loop. We used local variables to read true or false constants wired to them and write to the indicator lights outside the case structures. You will notice that there is a false constant wired to local variables outside the while loop. I do not know why they are there, but Brendan put them there, and I have always kept them there in subsequent files because the program might not work without them. See Brendan for more information.

The Event Structure

In "Adding On and Off Indicators.vi," there is one button for each case structure. But I needed three stop buttons to control the lower case structure with different time delays for each. You can't just wire two or more controls directly to a case structure. I explored the functions palette, the LabVIEW help, and the National Instruments discussion forums. Brendan's programming skills had taught me to look for useful functions in the Structures subpalette, so I eventually settled on the event structure as a likely method of controlling the case structures with buttons in "Heater Control Using Event Structure.vi." I had six buttons with a case for each button that would occur with the "Mouse Up" event. To view the details about these events, you can select "Edit Events Handled by This Case" on the event structure shortcut menu and navigate through the cases using the "Events Handled for Case" dropdown menu.

In each case (except timeout, which is set to -1 for never time out) I had a true constant wired to one of the case structures or to the conditional terminal of the while loop. Thus, each event case would turn on the heater and pump or turn them off by supplying a true value to one of the case structures or it would supply a true value to the conditional terminal of the while loop, which would stop the program from running. (The conditional terminal is set to "Stop if True.") Also in the event cases are numeric constants wired to the "Milliseconds to Wait" function in the second frame of the flat sequence structure in the lower case structure. These constants determine how long to wait before turning off the pump after turning off the heater. I have a constant where it is not needed in one of the cases.

I do not think I needed a for loop around the event structure. It seems that the event structure executes for only one iteration of the while loop after an event and then waits for another event.

However, the buttons in the program we needed would not simply turn the heater on or off. The start button will have to cause the program to turn the heater and pump on or turn them off based on the water temperature. In "Respond Once to Sine Signal.vi," the scenario is more realistic. I wired a simulated sine signal to a "Less or Equal?" function in an attempt to turn the outlets on if the signal were less than or equal to a certain value and off if it were greater than that value. The wiring is quite complicated, but the behavior of the program is simple. First press the run button. Then press the OK button labeled "Start Heater and Pump." Immediately a curve appears on the waveform chart, but the heater and pump do not turn on, and the curve does not change. Press the same button again and the curve will step up to higher y values, then stop again. After a few clicks, the curve crosses the 0.5 mark on the y-axis as it is decreasing and the heater and the pump turn on.

What I wanted was a sine signal that would be continuously generated and displayed on each iteration of the while loop after you press OK. Instead, in this program a signal is generated only once each time you press the OK button.

What went wrong? The problem is with the event structure, which executes only once in response to an event. When you first press the OK button, the "Start Heater and Pump" case executes once. That means the case structure with the "Simulate Signal" function gets a true value on one iteration of the while loop from a true constant wired through the event structure to the case structure. After the event case has executed once, the true constant is no longer wired to the case structure. Only when you press OK again is the true constant wired to the case structure. So the "Simulate Signal" function simulates one value change, from 0 to approximately 0.58, when you first press OK. Because the new value is greater than 0.5, the pump and heater do not turn on. Each additional click on OK simulates one more value change, and when the curve passes the 0.5 mark, the new value is less than 0.5, so the heater and pump turn on. I tried moving the functions around, but as long as I used an event structure, the buttons would only work once after being pressed.

Held up by SubVIs

At last I decided that the buttons I needed would not work with an event structure. Then I remembered radio buttons; they are in plain sight on the Boolean subpalette on the Controls palette, but I had not thought of using them before. I put a "Radio Buttons" control on the front panel of "Heater Control with Radio Buttons.vi" and wired it to a case structure on the block diagram. When that worked, I added buttons with a case for each button and changed the appearance of the buttons to look like the buttons I had made before. I put my "Simulate Signal" and comparison functions in the while loop. They would no longer need to wait for an event to happen before executing. I wired the comparison functions through the "Start Heater and Pump" case to the case structures, and as I had done with the event event structure, so I did with the case structure, adding true constants wired to the bottom case structure on the right and numeric controls wired to the "Milliseconds to Wait" function to determine how long the delay should be.

But when I started the program and clicked "Start Heater and Pump," strange things happened. The heater and pump would turn on, and the curve on the graph would move up from zero through the y values rather quickly until it reached 0.5, when the indicator lights would turn off as they were supposed to. But several seconds would go by before the outlets in the netBooter turned off, and when they did go off, they went off at the same time rather than with a delay between them. Meanwhile, when the curve reached 0.5 it would slow to a crawl. It moved, stopped for a while, moved again, stopped for a while, and upset my expectations of success.

I decided that something was slowing the program down, so I added a button and labeled it "Dummy Button" (later renamed "Default Button") to see what would happen if the case structure wired to the radio buttons were empty. When I clicked on the dummy button, the graph went faster than it had even when the top right case structure had been true. After that I suspected that both of the case structures were slowing things down, but the bottom one was doing it the most. However, I was very puzzled that the indicator lights would respond just as they

should but the subVIs were not working even though they were in the same structures as the Boolean constants and local variables for the indicator lights.

I suspect that the subVIs, smiley.vi, Outlet 1 Off.vi, and Outlet 2 Off.vi, are what slowed down the program. I used case structures frequently, and only the ones with the subVIs slowed down the program. When I wired Boolean indicators through a case structure, for example, the graph on the waveform chart looked like it did with the Default Button on. Because the indicator lights always worked on time, we may conclude that the case structure and its contents were working right except for the subVIs.

There are a couple of other things that need explaining in this VI. The Milliseconds to Wait function with the numeric constant 5000 wired to it is a much later addition. After I had tried various ideas that did not work, Ann Marie suggested that I increase the time between iterations of the while loop from 1 ms to some longer period. With the long delay, the heater and pump turned off separately as desired, but the graph was quite slow. The Boolean indicator lights "High Temperature" and "Low Temperature" (later renamed "On Case" and "Off Case") were originally connected with another attempt to use an event structure, which I will explain shortly. My last versions of the program, which display the signal changing rapidly as well as turning the devices on and off with the delay between them, also depend on these indicator lights.

Back to the Event Structure

My next idea can be seen, though greatly changed from the original, in "Events for Heater Control.vi" and "Copy of Events for Heater Control.vi." I had been reading one of the National Instruments discussion threads; it was titled "Programmatically detect a value change event." This thread strongly influenced my strategy. By this time I realized that the case structures containing the subVIs, the Boolean constants, the time delay, and the local variables were executing on each iteration of the while loop and that they were slowing the program down. I decided to try to use an event structure so that these case structures would get a true constant only on one iteration of the while loop.

"Events for Heater Control.vi" most clearly shows how I tried to use the indicators and the event structure. Originally everything inside the two while loops was in one big while loop. The Boolean indicator lights High Temperature and Low Temperature respond to the same values, true and false, as the case structures on the right do. In the "Edit Events" dialog box (accessible throught the event structure shortcut menu if you click "Add Event Case" or "Edit Events Handled by this Case") the indicators High Temperature and Low Temperature appeared on the list of event sources, and on the list of events was one called "Value Change." Therefore it seemed that I could add events for when the indicators changed from true to false or false to true. When the indicator changed value, an event case would execute. I wired the event case so that the case structures would get a true or false value from the comparison functions or from true constants in the cases for the radio buttons. However, the event cases never executed as I had hoped. It is hard to remember what happened, but I think that when the program was running, if you clicked "Start Heater and Pump" the graph would display one movement of the sine signal and then stop until you clicked again. The outlets for the heater and pump would turn on only on the third click even though the signal had started at a value less than the benchmark for turning them on. I think the event structure was making all the code wait for an event before executing. I had hoped that everything would execute except what needed an output from the event structure, but that was not what happened.

As for the event cases associated with the High and Low Temperature indicators, I have many questions in mind. I don't know whether there is a value change when you first click the start button. What would the value of the indicator be before you click the start button? Somewhere online I read that the event structure requires direct user interaction with the front panel. I interpreted the statement to mean that you have to click on a control to produce an event. If that is true, the value change of an indicator light would not be a valid event. However, in the "Edit Events" dialog box, the indicators were listed as event sources! Why would they be there if they couldn't cause an event? The same article noted an exception to the requirement for interaction with the front panel: you can set up a User Event. I found the palette, but I was unable to wire the functions correctly to create a user event that I wanted.

In "Events for Heater Control.vi," the event cases for the indicator lights have true constants wired to the case structures no matter which way the value of the indicator lights changes. Thus, if they were true and become false, the event case would still deliver true constants to the case structures. I suppose I was assuming that the water would be at a low temperature and that when you press "Start Heater and Pump" the value of the "Low Temperature" indicator would change from false to true. Then the heater and pump would turn on. Afterwards, in the Start Heater and Pump case, the devices would never turn on unless the water crossed the low temperature, and they would never turn off unless it crossed the high temperature. And they would always turn on at the low crossing, whether the temperature was rising or falling, and off at the high crossing. So they would never turn on while going above the high temperature or off while going below the low temperature. But we don't want the heater and pump to turn on again after the water cools from the high temperature. I had thought at the time that we would keep the water between the low temperature and the high temperature for a while.

I considered alternatives to the value change events. One possibility was to use conditional disable structures to disable the case structures on the right after they had recieved a true or false value on one iteration until another click on a control or a value change. However, apparently the conditional disable structure has conditions for what operating system you are using and not for what happens with controls and functions in a program. I won't say that something is impossible though, only that I did not know how to do it. And the conditional disable structure sounds very much like what I wanted. I considered the Diagram Disable structure also. It might have been possible to use it somehow. There is one Enabled case and the rest are disabled. Let the enabled case be the case structure when the first true or false value arrives, and let the disabled case be the case structure if the next values are the same. But cases are easier said than done.

After working with event structures for some time without success, I tested the theory that the case structures with subVIs were slowing the program down. In "Separated.vi," I substituted Boolean indicator lights for the case structures wired to the radio buttons case structure. I had the indicators inside the while loop, and I wired Boolean push buttons to the case structures with the subVIs. The push buttons' operation was set to "Latch when released," meaning that they would give a true value once rather than on every iteration after a mouse click on the button. When I ran the program, the indicators lit up at once when the comparison functions were true and stayed lit, but the graph always went as fast as it did with the Dummy/Default radio button. I believed that proved that the subVIs in the case structures were to blame for the slow graph, reasoning that the indicators were getting a true value on every iteration but didn't slow anything down. When I clicked on the push buttons, the graph slowed or paused just for a moment, then resumed as fast as before. The outlets turned on and off as they were supposed to, and the time delay worked. So the fast repetitions of the true value were what caused the subVIs in the lower case structure to turn off the heater and pump at the same time after a pause. Later after I had read something online about parallel while loops, while trying to pass data from one while loop to another I put the Boolean indicators outside the while loop to see if they would light up when I ran the program. They did not, so the Boolean values were not making it out of the while loop.

Success with Parallel Loops and Shift Registers

There was a National Instruments LabVIEW tutorial that explains parallel loops, and I followed the directions, as you can see in "Parallel Loops.vi." The tutorial is in my folder (Ben Willis). It is named "Tutorial on Variables and Race Conditions." You can find it online; the title is "Tutorial: Local Variable, Global Variable, and Race Conditions." You should study it also and follow the directions to make the VI they describe. It will help you to understand parallel loops and local variables. The great thing about parallel loops is that one can execute without waiting for the other. The tutorial explains that you have to be careful to avoid race conditions, but otherwise parallel loops are a great idea. This tutorial was where I found out about the "Reinitialize Values to Default" item on the front panel Edit menu. It has been invaluable in my later VIs.

Now that I knew you can pass data between while loops using local variables, I did it a

lot. Local variables are associated with front panel controls. In "Local Variable Foray.vi," I wired Low and High Temperature Boolean indicators to the comparison functions and created local variables for them. I changed them from Write to Read with a shortcut menu item and put them in the other while loop. I also created a local variable for the radio buttons and wired it to a case structure. The local variables read the values from their controls and pass them on to the case selector of the case structure (Radio Buttons) or through the case "Start Heater and Pump" (Low and High Temperature).

This VI worked better than any of the previous ones. The graph went as fast as I could wish, no matter whether the temperature was high or low. At first the outlets turned off at the same time after a delay. But following Ann Marie's suggestion I increased the delay between iterations of the while loop on the right to 5000 ms, and then the outlets turned off with the delay between them. There was a still a delayed response to a value change in the left while loop, though, because the right one had to wait a few seconds before iterating. I'll leave it to someone else to decide if a few seconds would matter. This VI would keep the temperature between the minimum and maximum rather than heating it up once and letting it cool. It could probably be changed, however, to behave as desired.

In this file there is a numeric control wired to the Milliseconds to Wait function in the flat sequence structure. Sometimes I had different numeric controls in each stop case of the radio buttons case structure so that each case would have a unique delay. In my final VIs this is so. But in some of the VIs I had the time delay control or constant in other places for testing whether a different position would work or to save time. It takes a little while to put one in each case and wire it to Milliseconds to Wait.

For some time I had been wondering whether shift registers would enable the program to send only one true value to the case structures with the subVIs for turning the outlets on and off. Shift registers can pass data from one iteration of a while loop to the next iteration. I tried to do this in "Shift Registers.vi" and "Radio Buttons with Shift Registers.vi," but neither attempt succeeded. Finally I found another NI tutorial, "Tutorial: Timing, Shift Registers, and Case Structures." I followed the directions from the tutorial in "Shift Register Tutorial.vi." I strongly recommend this tutorial for learning about shift registers. I have saved the file by the same name, except for the colon, in my folder.

Where to Put the Local Variables.vi

After studying the shift register tutorial, I tried shift registers again in "Where to Put the Local Variables.vi." I used two local variables for each of the High and Low Temperature indicators. Two of the local variables are in the false cases of the case structures wired to the "Equal?" comparison functions in the while loop on the right. The two others are not in any structure but the while loop. I will begin my explanation with reference to the High Temperature and Low Temperature local variables outside the case structures but inside the right while loop.

The program should not be run, and it usually will not run either, without clicking "Reinitialize Values to Default" on the front panel Edit menu. Once that is done, the radio buttons return to their default state, with the Default button on, and the High and Low Temperature indicators are set to their default state, which is false. Then when you click the run button the program runs with the Default button on, and in the case structure for the radio buttons in the left while loop, the Default Button case is selected. Because there are no wires through the Default Button case, the High and Low Temperature indicators are still false.

When you click "Start Heater and Pump," however, the case changes to the Start Heater and Pump case. In this case the indicators are wired to the comparison functions that tell whether the simulated signal (which we might as well imagine as the temperature of the water tank) is less than or equal to one value or greater than another value. Supposing that the signal is less than or equal to the smaller value, the Low Temperature indicator becomes true.

In the other while loop on the right, a local variable for Low Temperature reads the new true value from the indicator and outputs it to the right terminal of a shift register (a small green box with a green up arrow on the right border of the while loop). On the next iteration, the left terminal of the shift register (with a small green down arrow) outputs the true value from the first iteration to the "Equal?" comparison function.

On the first iteration with the true value, the Low Temperature variable also outputs the true value to the "Equal?" function, where it is compared with the value from the previous iteration supplied by the shift register. Because the first true value does not equal the value before it, the "Equal?" function outputs a false value to the case structure to which it is wired. Then the false case executes. In it, another Low Temperature local variable reads the true value of the indicator and outputs the true value through a wire to a case structure that should be familiar by now, turning on the outlets for the heater and the pump and the indicator lights "Heater On" and "Pump On."

On the second iteration after the Low Temperature indicator becomes true, the Low Temperature variable reads the true value and outputs it to the shift register and the "Equal?" function. But the shift register outputs the true value from the first true iteration to the "Equal?" function so that it is comparing equal values and becomes true. The "Equal?" function outputs a true value to the case structure, and the empty true case executes. Therefore the case structure with the subVI for turning on all outlets does not get another true value. On subsequent iterations, as long as the Low Temperature indicator is true, the "Equal?" function remains true, and nothing happens to the outlets. When it becomes false, nothing happens either, because the false case is empty.

Suppose that after a while the High Temperature indicator becomes true. In the while loop on the right, a High Temperature variable reads the true value and outputs it to a second shift register and to an "Equal?" function. The shift register outputs the value from the previous iteration to the "Equal?" function, and because the values are not equal, the "Equal?" function outputs a false value to a case structure. The false case executes. Another local variable for High Temperature reads the true value from the indicator and outputs it to the case structure with the subVIs for turning off the outlets.

If the High Temperature variable were to read a second true value, it would output it to the shift register and the "Equal?" function. At the "Equal?" function it would be the same as the true value from the previous iteration, the empty true case of the case structure would execute, and nothing more would happen.

However, in this VI the High Temperature variable never reads a second true value. This is because the High Temperature variable in the false case of the case structure is also wired to the conditional terminal of the while loop (a small reddish orange stop sign shape in a yellow square), which is set to "Stop if True." So when that variable becomes true, the right while loop stops, leaving the outlets off. I made this wiring because I realized that we need to raise the water to a certain temperature, then let it cool rather than keeping it between the two temperatures.

Whenever the High Temperature indicator becomes true, the right while loop stops and won't run again until the program is stopped and run another time. When you press any of the stop buttons, High Temperature becomes true (so High Temperature is not the best name for it, and I changed it in another version of the program) and the right while loop stops. You can start it again only by clicking "End Bending" to stop the program (or Abort Execution, but you should avoid clicking Abort) and running it after clicking "Reinitialize Values to Default."

I needed a time delay in turning off the outlets. I made a local variable for Radio Buttons and wired it to a case structure in the right while loop. Depending on which radio button is on, one of four numeric controls with certain default values is selected in the case for the button and wired to the Milliseconds to Wait function in the flat sequence structure. Thus the delay is accomplished.

Local Variables and Stop Buttons that Work Right.vi

In "Local Variables and Stop Buttons that Work Right.vi," I made some improvements, but the VI does not work as I had wanted it to. It does seem to work as well as "Where to Put the Local Variables.vi." One improvement that succeeded was the label change of the "Low Temperature" and "High Temperature" indicators to "On Case" and "Off Case." As we have seen, the High Temperature indicator did not always indicate high temperature; it sometimes meant that a stop button was on. The "Start Heater and Pump" button is also mislabeled because it starts them only if the water is less than or equal to a certain temperature, but I couldn't think of a better name for it.

What I really wanted to do in this file was to enable the stop buttons to be operated without stopping the right while loop while retaining the feature of stopping this while loop if the water temperature exceeded the maximum value. The "End Bending" button would stop both while loops. (There is a problem with having "End Bending" stop both while loops. If the outlets were on when the while loops stopped, they would remain on and the heater and pump would keep running after the program was shut down. Therefore, if the heater and pump are on and you want to stop the program, you need to click one of the stop buttons first, then "End Bending.")

I kept the wiring for the "On Case" the same as in "Where to Put the Local Variables.vi," but I changed the wiring for the "Off Case" or Low Temperature local variable. Between the "Equal?" function and the case structure I added an "And" function that is true if two conditions are true. I wired a Radio Buttons variable to one terminal of a shift register and another "Equal?" function and wired the other terminal of the shift register to the other input of the "Equal?" function, then wired the output terminal of the "Equal?" function to an "And" input terminal.

If both the radio buttons and the Off Case indicator were the same as on the previous iteration, the "And" function would output "true" to the case structure and the empty true case would execute. But if either the radio buttons or the Off Case indicator value changed, "And" would output "false" to the case structure and the false case with the local variable "Off Case" would execute and turn off the outlets if the variable read a true value from the "Off Case" indicator. If this false case executed while the "Start Heater and Pump" button was on, the "Off Case" variable would be wired through two case structures to the conditional terminal of the while loop. So if the water exceeds the maximum temperature, the heater and pump will go off and not come on again during the same run of the program. But if any other radio button is on, the "Off Case" variable would not be wired to the conditional terminal.

I kept the case structure with the Time Delay controls and put the wire to the conditional terminal through the "Start Heater and Pump" case. I made another case structure and wired it to a local variable for the "End Bending" button (which is not one of the radio buttons). If you click "End Bending," it becomes true and both while loops will stop. In the right loop the true case with a true constant wired to the conditional terminal will execute. If you run the program while "End Bending" is true, it will go for one iteration and stop. To run the program properly, you need to "Reinitialize Values to Default."

As long as "End Bending" is false the case structure will be false, and a wire is connected to the conditional terminal through the case structure. This wire is to stop the while loop if the "Start Heater and Pump" button is on and the temperature exceeds the maximum value.

Button Peculiarities

The VI "Local Variables and Stop Buttons that Work Right.vi" may be good enough for our purposes, but it could be improved. When you run the program with the simulated signal, the signal value begins at zero and increases. If you click "Start Heater and Pump" before the signal value is greater than 0.2 (or whatever value is selected by the "Minimum Temperature" numeric control), the outlets for the heater and pump turn on. Then if you click on any of the stop buttons, the outlets turn off with a time delay between them. If you click again on "Start Heater and Pump," the outlets do not turn on again even though the signal value may still be less than or equal to 0.2.

Rewired Local Variables and Stop Buttons.vi

Actually, they aren't rewired. I thought I had a new idea, but then I changed my mind. If someone else wants to try something different, go ahead and rewire them.

Thermocouple Signal Attempt.vi

This VI is the same as "Local Variables and Stop Buttons that Work Right.vi" except that I

replaced the "Simulate Signal" function with the functions that I had intended to measure the thermocouple voltage and convert it to temperature.

Work on Waveform Display.vi

This VI is the same as "Local Variables and Stop Buttons that Work Right.vi" except for the waveform chart and the waveform graph. My previous VIs had a waveform chart wired to the "Simulate Signal" function that would display the signal value over the past tenth of a second. Jim wanted a graph of the signal over the entire program run time, and he wanted to have the time and temperature data automatically saved in a file as the program was running.

After my success with shift registers, I began to work on the graph we wanted, but I did not have time to do much. I added a waveform graph labeled "Temperature During Entire Run Time" and wired it to the "Simulate Signal" function. I read an article in the NI LabVIEW Knowledge Base titled "What Is the Difference Between Graphs and Charts in LabVIEW?" The article says that a waveform graph accepts arrays of data and displays them after it has received the entire array but that a waveform chart "remembers and displays a certain number of points by storing them in a buffer." But when I first ran the program with both the graph and the chart wired to "Simulate Signal," they looked the same to me.

I made some kind of change to the waveform chart labeled "Temperature," and now it shows straight diagonal lines moving across the display from left to right when you run the program. I don't remember what caused that unhelpful change. Perhaps the best solution would be to delete the chart, replace it with another by the same name, and change the time display on the x-axis as I describe below.

When I first added the waveform chart, it did not show the time on the x-axis. Sometime later I unchecked the item "Ignore Time Stamp" on the chart's shortcut menu, and it showed the time after that. However, it only showed the time in seconds. It would be better to show the time in hours, minutes, and seconds. In the "Format and Precision" tab of the Graph Properties dialog box I found the option HH:MM:SS for displaying the time and selected it. The time became more comprehensible. We won't have to figure out how long 7000 seconds is.

I wanted to make the Waveform Graph display the temperature over the entire run time. I thought that if I put the graph outside the while loop it might be forced to do that, so I tried it. But now the graph waited till the program was stopped to display data, and it still displayed only the values over the past tenth of a second.

I did my work on this file on a Wednesday. The next day we had our research group meeting, and Dr. Jones said I should write this report before I leave for classes in the fall semester. Someone else must carry on and make an adequate graph and file saving capability.

Final Experiments

I wrote my report, but I wanted to stay until 5:00, so I decided to experiment with a waveform chart until then. I copied "Work on Waveform Display.vi" to a new folder (C:\Documents and Settings\Jim\My Documents\Ben Willis\Final Experiments) and looked for example VIs (available from the Getting Started window) that would show how you can use graphs and charts. None of them had what we need. Then I did some searches online. I found a Graphs wiki that I saved in the new folder and a thread on the NI discussion forum called "How to view the curve of the past on Strip Chart?"

I decided to see if a strip chart would display all the data since the program began running, so on the shortcut menu for the waveform chart I changed the update mode to strip chart (Advanced>Update Mode>Strip Chart). However, the chart shows the data only over a five-second interval. I tried multiplying the chart history length by 100, but that didn't change the display either. The update modes may be worth investigating, though. I renamed my last file "Strip Chart.vi."

Perplexed by Thermocouples

We were planning to use a type J thermocouple to measure the water temperature in the fish tank for bending the fibers. The type J thermocouple is a junction of two metals: iron and constantan (an alloy of copper and nickel). Dr. Jones asked me to study information on thermocouples and report my findings at a research group meeting. I read a number of web pages and found out some things. Some of the articles are saved as files on Jim's computer in C:\Documents and Settings\Jim\My Documents\Ben Willis\Thermocouples. I read the document "Making Temperature Measurements using Measurement Computing DAQ Products," which was published by the manufacturer of the DAQ card we have. This article briefly explains the Seebeck voltage and several options for cold junction compensation.

Another article ("Criteria for Temperature Sensor Selection of T/C and RTD Sensor Types: The Basics of Temperature Measurement Using Thermocouples, Part 1 of 3," published by Acromag, Inc.) corrected my understanding of thermocouples by emphasizing that the Seebeck voltage is caused by a temperature difference between junctions, not merely by a junction of two metals.

One of the most helpful articles I read was an Application Note from the Dataforth Corporation titled "Introduction to Thermocouples." It explains that if there is a temperature difference between the ends of any wire made of a single metal, some electrons move from the hot end to the cold end. At the group meeting, Dr. Jones said the electrons move this way according to the Second Law of Thermodynamics.

Another article ("Thermocouple Theory and Practice," published by the British company Labfacility) showed a similar simple situation, with two wires of different metals joined together at either end. A current would flow in these wires.

One of the difficulties of using thermocouples is that when the wires are different from the metal of the measuring device, they form junctions. If there is a junction at each terminal and the junctions are at the same temperature, they cancel each other with equal and opposite voltages ("Making Temperature Measurements"). If you don't want these junctions at the measuring instrument, you can run wires from the instrument to an isothermal block and have the junctions there.

One of my main difficulties in studying thermocouples was to understand a voltage. Wikipedia gave a clear definition (but said there are other definitions): "A voltage is the energy required to move a charge from one point to another." I wanted to apply that definition to the thermocouple, so I wondered what the two points were. They could be any two points in the circuit. However, Wikipedia also said that the path between the points did not matter in determining the voltage, only the initial position and the final position, so some of the points in the circuit must not matter. In the end I concluded that the points that define the thermocouple voltage were at the terminals of the measuring instrument. But what if they were two other points, perhaps both inside the instrument or one in one wire and one in the other? Then you would measure different voltages in the same circuit.

In a loop made entirely of a single metal, no current will flow even though electrons may move from the hot end to the cold end. Dr. Jones said that this is because the charges are pushing on each other with equal force. At least, that's what I thought he said, and I took the idea and envisioned an answer to why there is a current in a loop of two metals with the junctions at different temperatures. From my reading I knew that the magnitude of the difference between the charges at either end of a wire with a temperature gradient depends on what metal the wire is. In a loop of two metal wires, a different number of electrons will move from the hot end to the cold end of each wire. That means that at each junction one metal will be more positive than the other metal. Electrons will flow from the less positive metal to the more positive metal. In other words, there will be a current in the wire.

In stating this argument, I see a difficulty in it. At the hot junction, the electrons will move away from the junction in each wire, producing a positive charge in both metals at the junction. At the cold junction, they will move toward the junction in each wire. There will be a negative charge in each metal at the cold junction. But like charges repel each other, so wouldn't these charges

push each other apart? In a wire, protons can't move, but electrons can. The negative charges, even though they are unequal, should drive each other apart. The electrons should move back up each wire toward the hot junction. What a conundrum! I shall have to take a physics course. Maybe that will help.