9 - Analog and Digital I/O

Log in or register to post comments

Analog and Digital I/O Overview

9.1 Low Speed Acquisition (< 100hz)

9.1.1 Introduction

Because Windows needs to occasionally use the CPU in your computer to redraw the screen, process the mouse, and perform other tasks, you generally cannot read inputs or control outputs faster than 100hz using the computer as the timer. Of course with the new multicore CPU's, you may be able to tweak extra speed out DAQFactory's polling loops, essentially putting the acquisition on one core, and letting Windows use the other core for display. Doing this sort of thing requires more advanced techniques described in the last chapter. This section describes how to do software (i.e. DAQFactory) polled reading and writing of analog and digital inputs and outputs. The techniques are essentially the same for both analog and digital inputs and outputs.

9.1.2 The Easy Way - With Channels

The easiest way to read analog and digital inputs at low speeds (i.e. < 100hz) or set analog and digital outputs is to use channels as we did in the Basic I/O chapter. No scripting is required, you can easily convert the readings using Conversions, and log it using Logging sets. For the U3, which allows the specification of the differential channel, you can simply put the channel number for the negative side in the Quick Note / Special / OPC column of the channel. If you need more advanced configuration, you can still use channels, combined with some basic scripting to send the configuration commands to the LabJack. For example, to set the UE9 resolution to 14 bit and the range on channels 2 and 3 to +/-5V:

```c
using("device.labjack.")
include("c:\program files\labjack\drivers\labjackud.h")
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 14, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 2, LJ_rgBIP5V, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 3, LJ_rgBIP5V, 0, 0)
GoOne(0)
```
This was explained earlier in the chapter on scripting.

### 9.1.3 Basic Scripting using eGet

Channels, however, aren't always the best choice, and sometimes you need script. The next step up from channels is to use basic scripting and the eGet() function to retrieve a single value, or ePut() to set a single value. For example, to read channel 1 and then put the result into a channel called MyChannel we would do:

```plaintext
private err
private val
private string message
while(1)
    err = eGet(0, LJ_ioGET_AIN, 1, @val, 0)
    if (err)
        ErrorToString(err, @message)
    else
        MyChannel.AddValue(val)
    endif
    delay(1)
endwhile
```

Now truthfully, this next code snippet does the exact same thing, provided MyChannel is setup to read channel 1:

```plaintext
while(1)
    read(MyChannel)
    delayendwhile
```

But, this code does not allow for any direct error handling, and of course doesn't demonstrate the eGet function! eGet is also more useful when you don't know your channel numbers at runtime. In the first example, we used scalar values in our eGet() function call, but there is no reason why you couldn't use variables that could then be changed from elsewhere:

```plaintext
err = eGet(ID, LJ_ioGET_AIN, chan, @val, 0)
```

Another thing the first example shows is the AddValue() function of the channel MyChannel. This function essentially stuffs a value into a channel. This allows you to utilize the benefits of a channel (history, easy logging, etc), without actually using the channel Timing to trigger the reads. In this case, we are putting the result of the eGet call into MyChannel. MyChannel does not have to have the same channel number, I/O type, or even be Device Type "LabJack".

**Note:** if using the U3, you will need to configure the pins as analog inputs. With Channels, this is done automatically for you, but when scripting you have to do it yourself. It only takes one line to
enable a channel:

ePut(ID, LJ_ioPUT_ANALOG_ENABLE_BIT, 1, 1, 0)

Sample file: LJGuideSamples\eGet.ctl

9.1.4 More Advanced Using Add / Go / Get

eGet can be handy for very simple scripts, but when doing a number of different requests, it is much better to use the AddRequest() / GoOne() / GetResult() functions. So, if we wanted to read channels 1 through 3 we could do:

```c
eGet(0, LJ_ioGET_AIN, 1, @val1, 0)
eGet(0, LJ_ioGET_AIN, 2, @val2, 0)
eGet(0, LJ_ioGET_AIN, 3, @val3, 0)
```
or we could do:

```c
AddRequest(0, LJ_ioGET_AIN, 1, 0, 0, 0)
AddRequest(0, LJ_ioGET_AIN, 2, 0, 0, 0)
AddRequest(0, LJ_ioGET_AIN, 3, 0, 0, 0)
GoOne(0)
GetResult(0, LJ_ioGET_AIN, 1, @val1)
GetResult(0, LJ_ioGET_AIN, 2, @val2)
GetResult(0, LJ_ioGET_AIN, 3, @val3)
```

OK, you may think that eGet looks much easier since its only three lines of code vs. seven, and it is easier on a basic level, but with ease you lose flexibility and efficiency. Using the second method is more efficient internally. The second method also allows you to do error handling easier using the GetNextError() function. With error handling, the eGet code ends up looking like this:

```c
err = eGet(0, LJ_ioGET_AIN, 1, @val1, 0)
if (err)
    ... error!
endif
err = eGet(0, LJ_ioGET_AIN, 2, @val2, 0)
if (err)
    ... error!
endif
err = eGet(0, LJ_ioGET_AIN, 3, @val3, 0)
if (err)
    ... error!
endif
```
But the Add / Go / Get looks like this:

```
AddRequest(0, LJ_ioGET_AIN, 1, 0, 0, 0)
AddRequest(0, LJ_ioGET_AIN, 2, 0, 0, 0)
AddRequest(0, LJ_ioGET_AIN, 3, 0, 0, 0)
GoOne(0)
GetResult(0, LJ_ioGET_AIN, 1, @val1)
GetResult(0, LJ_ioGET_AIN, 2, @val2)
GetResult(0, LJ_ioGET_AIN, 3, @val3)
while (GetNextError(1,@io,@ch,@err)
    ... error!
endwhile
```

As you can see the second method is much cleaner and easier to read. The eGet() version would get worse and worse as you added more function calls. Using the second method also allows you to create a single error handler for the entire block, or as shown in the section on error handling, you can create a single function to do all your error handling for all your scripts.

Note: variable declarations are not shown in the above examples, but would be required. Likewise, ... error! would need to be replaced with script to actually do something in the case of an error.

### 9.1.5 Controlling outputs

We've talked a little about setting an output from a screen control in section 4.6, but this requires user input. We also talked about setting an output based on an input in section 7.1. In this section we saw that you can easily set an output channel by simply assigning a value to it. So, to set a DAC channel named ValvePosition to 3 volts (assuming no Conversion exists on the channel), we would simply do:

```
ValvePosition = 3
```

You can also apply conversions to outputs like we did with inputs in section 4.3, however, the conversions work in reverse. For inputs, the conversion takes the raw voltage (counts or other units) from the LabJack and converts it into engineering units like temperature or pressure. For outputs, the conversion takes the engineering units and converts into voltage. So, for if we had a proportional valve that takes a 0 to 5V signal and we want to be able to specify the percentage open, where 0% = 0V and 100% = 5V, the conversion would be:

```
Value / 20
```

Then, to open the valve to 60% we could just do:

```
ValvePosition = 60
```

and 3 volts would be outputted from the DAC of the LabJack.

Now as a further example, lets say we'd like to ramp the valve position from 0 to 100% over 60
seconds in steps of 1%. The script is quite simple:

```c
// initialize valve position to 0
ValvePosition = 0
while (ValvePosition < 100)
  ValvePosition = ValvePosition + 1
  delay(0.6)
endwhile
```

Doing a ramp and soak is not much harder, just split out the while loops. Lets say we want to ramp to 40% in 20 seconds, soak for 10 seconds, ramp to 80% in 40 seconds, soak for 5 seconds, then ramp back down to 0 in 60 seconds:

```c
// initialize valve position to 0
ValvePosition = 0
// ramp to 40
while (ValvePosition < 40)
  ValvePosition = ValvePosition + 1
  delay(0.5)
endwhile
delay(10) // soak
// ramp to 80
ValvePosition = 40 // this is to make the graph look good
while (ValvePosition < 80)
  ValvePosition++ // this is the same as VP = VP + 1
  delay(1)
endwhile
delay(5) // soak again
ValvePosition = 80
// ramp down to 0
while (ValvePosition > 0)
  ValvePosition--
  delay(0.75)
endwhile
```

One important note: these sequences won't work if you try and ramp past the highest value of the output. If you try and set an output to an invalid value you will get an Alert and the channel WILL NOT update, so the while() will never exit because ValvePosition would never reach the desired point.

Sample file: LJGuideSamples\RampSample.ctl

### 9.2 High Speed Acquisition - Streaming

#### 9.2.1 Introduction

When you want to do things faster than 100hz, its usually best to let the LabJack perform the timing. The LabJack can then send data back in blocks and relieve your CPU from constantly having to do things. The LabJacks support streaming of both digital and analog inputs as well as timers and counters. This section explains how to setup and performed streamed data
9.2.2 Basic Streaming

When you want to read inputs at rates faster than 100hz, or at very precise intervals, it is usually best to use the LabJack's stream mode instead of having DAQFactory and the PC set the read rates. Anything above 100hz is difficult for a PC to perform since it has so many other tasks to do as well. To setup streaming in DAQFactory you will need to use a combination of Channels and sequence script. Streaming in DAQFactory is different from how the LabJack User Manual describes, as DAQFactory handles all the callback and data collection, putting the data into the appropriate channels. In this sample we'll stream 2 channels.

1) Start DAQFactory up with a new document.

2) Click on CHANNELS: in the Workspace to go to the channel table.

3) Add two new channels, ChannelA and ChannelB, both Device Type = LabJack, D# = 0, I/O Type = A to D, and Channel numbers 2 and 3. Set the Timing for both channels to 0, and the History: set to 36000.

3) Click on Apply to save your changes.

4) Right click on SEQUENCES: in the Workspace and select Add Sequence. Call the new sequence StartStream

5) In the sequence editor window that appears, enter the following script:

```plaintext
// standard initialization:
using("device.labjack")
include("c:\program files\LabJack\Drivers\labjackud.h")
// setup stream:
// set scan rate:
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chSTREAM_SCAN_FREQUENCY, 1000, 0, 0)
// setup channels to stream:
AddRequest(0, LJ_ioCLEAR_STREAM_CHANNELS, 0, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 2, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 3, 0, 0, 0)
GoOne(0)
// start the stream:
global scanrate = 0
eGet(0,LJ_ioSTART_STREAM, 0, @scanrate, 0)
// scanrate now has the actual scanrate, which you can display on the screen if you want.
```

6) Click on Apply and Compile to save your script.

7) Right click on SEQUENCES: in the Workspace and select Add Sequence. Call the new sequence StopStream and enter the following script:
8) Click on **Apply and Compile** to save your script.

9) Click on **Page_0** under **PAGES**: in the workspace to display a blank page.

10) On that page, right click and select **Graphs - 2D Graph**.

11) While holding down the Ctrl key, click and drag the graph to move it to the top left corner of the page, then click and drag the bottom right corner of the graph to expand it to take up about 3/4 of the screen.

12) Right click on the graph and select **Properties**.... For the **Y Expression** put **ChannelA**. Click on **New Trace** and for the **Y Expression** put **ChannelB**. Click **OK** to close the properties window.

13) Right click somewhere on the empty part of the page and select **Buttons - Button**. Right click on the new button and select **Properties**....

14) For **Caption**, put **Start**, then go to the **Action** tab and select **Start/Stop Sequence**, then select your **StartStream** sequence.

15) Repeat steps 13 and 14, but put **Stop** for the caption and **StopStream** for the sequence.

That is it. You should be able to click on the Start button and have streaming on channels 2 and 3 start up and be graphed. It is possible that the values will be off the scale of the graph, so you may want to click on the graph, then right click on the graph and select **AutoScale - AutoScale Y**.

One important point if you start tweaking this sample: the Channels that you created must have the same D# and channel number as the one you specified in the **LJ_ioADD_STREAM_CHANNEL** request. The I/O Type must be "A to D" as well, even if you are streaming digital inputs, timers or counters. If not then DAQFactory won't know where to put the data that is streaming in.

**Note**: make sure you configure your inputs before starting the stream. For the U3, this means you have to set the pins to analog input as shown in the example file.

**Note**: you should not change the **LJ_chSTREAM_WAIT_MODE**, as all waiting is handled internally. If you change this, you will most likely cause streaming to stop functioning

Sample file: **LJGuideSamples\BasicStream.ctl**

### 9.2.3 Streaming Other Inputs

You can stream other inputs besides the analog inputs of your LabJack. This is done by specifying special channel numbers when doing **LJ_ioADD_STREAM_CHANNEL**. The important part here is that even though the LabJack is actually streaming something other than an analog input, you MUST specify A to D for the I/O Type when creating your DAQFactory Channels to
receive the data.

The available channel numbers are slightly different for each LabJack and listed here for your reference:

**U3:**
- 193 EIO_FIO
- 200 Timer0
- 201 Timer1
- 210 Counter0
- 211 Counter1
- 224 TC_Capture0
- 225 TC_Capture1
- 226 TC_Capture2
- 227 TC_Capture3

**UE9:**
- 193 EIO_FIO
- 194 MIO_CIO
- 200 Timer0
- 201 Timer1
- 202 Timer2
- 203 Timer3
- 204 Timer4
- 205 Timer5
- 210 Counter0
- 224 TC_Capture0
- 225 TC_Capture1
You may notice that in DAQFactory, we have multiple TC_Capture channel numbers, where the LabJack documentation only lists one. This is to allow you to stream the high order word of multiple timers and counters and keep the data separate in separate channels. Internally, they are exactly the same, so you have to specify the appropriate TC_Capture immediately following its Timer or Counter channel #. In other words, to read the entire 32 bits of Timers 0 and 1, you'd do:

AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 200, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 224, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 201, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 225, 0, 0, 0)

**9.2.4 Triggered**

Triggered streaming is currently only supported by the UE9 and UE9-Pro.

Triggered streaming is similar to regular streaming except instead of using the internal LabJack clock to determine when a scan of the stream channels occurs, an external pulse triggers the scan. The interval between external pulses must be less than the maximum stream rate for the current input resolution. The external pulses do not need to occur at a constant interval. To enable external triggering, just add the following line of script before adding your stream channels using LJ_ioADD_STREAM_CHANNEL:

AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSTREAM_EXTERNAL_TRIGGER, 1, 0, 0)

The trigger input will be the first available FIO pin based on which timers and counters you have enabled.

The only problem with triggered streaming is that the time of each data point will be off. This is because the LabJack buffers the scans and DAQFactory doesn't actually get the data until a full packet occurs. DAQFactory doesn't realize this and assigns times based on an assumed interval. If you have the bandwidth, i.e. your pulses are slow enough that you aren't close to the stream interval limit, you can use the system timer mode of the timers to retrieve exact relative times of your scans. To do this, you need to setup a timer for system timer in, and then add the timer to the list of channels to stream. Depending on how long your experiment runs, you may be able to
get away with only SYSTIMERLOW. For the UE9 at 750khz, the low timer will roll over every 5726 seconds. Here's how to do it:

1) Enable two Timers:

AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 2, 0, 0)

2) Set the mode:

AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmSYSTIMERLOW, 0, 0)
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 1, LJ_tmSYSTIMERHIGH, 0, 0)
GoOne(ID)

3) Set up the stream to stream analog input 2 and 3 in external trigger mode:

AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSTREAM_EXTERNAL_TRIGGER, 1, 0, 0)
// setup channels to stream:
AddRequest(ID, LJ_ioCLEAR_STREAM_CHANNELS, 0, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 3, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 4, 0, 0, 0)

4) Now we need to add our timers to the list of channels to stream. Make sure you use the order indicated:

AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 200, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 224, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 201, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 225, 0, 0, 0)

5) Now finish up the stream setup:

GoOne(ID)
// start the stream:
global scanrate = 0
eGet(ID,LJ_ioSTART_STREAM, 0, @scanrate, 0)
// scanrate now has the actual scanrate, which you can display on the screen if you want.

6) Create 4 channels to receive this timing data in addition to the 2 you created to receive the analog input. All 6 channels are I/O type: A to D, **Timing** = 0. **Channel #**’s will be 3, 4, 200, 201, 224, and 225.

That completes it. When you run your script, the stream will start, streaming both analog inputs 2 and 3 as well as the system timer. A scan will occur every time a trigger is detected on FIO2.
FIO0 and FIO1 are used by the 2 timers. With each scan, your six channels will update. The time associated with these channels will be incorrect, but channels 200, 224, 201, and 225 will contain the 4 words that make up the 64 bit system timer value. While this is not absolute time, it will give you relative time between each triggered scan. Just use the difference in counts divided by the system clock speed of 750khz for the UE9 to determine the actual number of seconds between scans. The best way to do this is to create a calculated V channel:

7) Right click on CHANNELS: under V: in the Workspace. Note this is note the same CHANNELS: that we've been clicking before. Select Add V Channel

8) Call the new channel TheTime

9) Click on the new channel in the workspace. In the Expression window, put:

\[(\text{TimerLowLow} + \text{TimerLowHigh} \times 2^{16} + \text{TimerHighLow} \times 2^{32} + \text{TimerHighHigh} \times 2^{48})\]

This assumes you named channel 200 TimerLowLow, 224 TimerLowHigh, 201 TimerHighLow, and 225 TimerHighHigh. You also may want to put a divisor at the end to convert to seconds:

UE9:

\[(\text{TimerLowLow} + \text{TimerLowHigh} \times 2^{16} + \text{TimerHighLow} \times 2^{32} + \text{TimerHighHigh} \times 2^{48}) \div 750000\]

10) Click Apply.

At this point, you can reference this channel like you would any other, except putting V. in front of it. Instead of getting a channel reading, you'll get the result of the calculation. Since we didn't use any [0] notation, this is the entire array calculated from all the readings. If you want to graph your channels, you'd put:

V.TheTime

as the X Expression in place of Time. You'll need to change the bottom axis type to Lin, undo Use Time Width, and adjust the Scale From and Scale To:
All of this is shown, complete, in the sample file:

Sample file: \LJGuideSamples\TriggeredStream.ctl

9.2.5 Error Handling forStreaming

Streaming from a LabJack is what is called an asynchronous action. This means that the LabJack does its own thing and every so often it tells DAQFactory that there is new data or there is an error. For this reason, you cannot simply look at the error code returned by LJ_ioSTART_STREAM to catch all stream errors. This command may return a stream configuration type error, so you'll want to check for it using the same methods as the low speed acquisition, but will not handle errors in the actual stream. For this you have two choices:

1) You can create a simple sequence to retrieve the last stream error continuously and do something if it returns an error. The function GetLastStreamError() will return the code for the last stream error. This is reset when you start the stream. This, however, is not the best way to do this and will waste processor power.

2) You can use the DAQFactory OnAlert event, which only gets called when an actual error occurs. Using the event as an error handler is described in the previous section on error handling, and catching stream errors would be done the same way. One common error you might want to catch is if your LabJack accidentally gets unplugged while streaming. If you wanted to automatically restart streaming when it is reconnected you could do this in your OnAlert sequence:

```c
if ((left(strAlert,10) == "D0050:00:2001") && (Streaming)) // 2001 is LJE_RECONNECT
    StartStream()
endif
```

Now, this assumes that you have a sequence called StartStream that will reconfigure the LabJack and actually restart the stream. It also assumes that you've created a global variable called "Streaming" that you set to 1 in StartStream, and to 0 in StopStream. This is so an accident unplug when you aren't streaming doesn't spontaneously start the streaming process. Finally this assumes you are using device number 0 / first found. Please see the section on OnAlert if you
are using multiple LabJacks.

9.3 Thermocouples

Thermocouples are a very common way to measure temperature. They come in a wide variety of forms and have a large range. Do to their small voltage output and something called a cold junction, they are, however, slightly more challenging to read. Most applications need a temperature range of about -50 to +150°C, though, and for these applications you might want to consider using a silicon type sensor, as they are cheaper, more accurate and much easier to use. Examples of these are the EI-1022 and EI-1034 sold by LabJack.

Thermocouples are simply two different metals that meet at a single point. Due to a general property of metals, a small voltage is generated that is proportional to the temperature of this junction. There are many different types of thermocouples, identified by a letter such as J type or K type which determine the metals used which then determines the temperature range and voltage output scaling. In all cases, this voltage is between about -5 and 5 millivolts, a rather small voltage.

It gets more complicated though: since the terminals on your DAQ device (the LabJack) are made of yet another metal, there are two more points where different metals are touching which are generating a voltage proportional to the temperature of this terminal. This is called the cold junction. In order to get an accurate thermocouple reading, you have to adjust for the cold junction. This is called cold junction compensation.

Finally, we should mention that most thermocouples are only accurate to about 1 or 2 degrees C, though calibration can help a little with this.

Now that you have the basics down we can talk about how to read a thermocouple. There are several choices depending on what LabJack you have:

**U3:** the U3 has a resolution of 12 bit and a range of +/-2.44, the minimum voltage you can read is about 1.2 millivolts (4.88 volts / 4096 steps). Because a thermocouple typically outputs around 40 microvolts per degC, you only get a precision of about 30 degrees! Obviously that isn't going to work so you have to use an amplifier such as the LJTIA or EI1040, both available from LabJack, to amplify that millivolt signal before it gets to the converter.

**UE9:** the UE9 has a resolution of 16 bit so has 65536 steps instead of the 4096 of the U3. This helps some. The UE9 also has some built in gain. Typically you get about 76 microvolts minimum step noise free, and 15 microvolts RMS. This means you get about 1 or 2 degrees C of resolution. This isn't bad and might work. For the exact specs, please review Appendix B of the UE9 Users Guide. If you need better than this, then you will need to get an LJTIA or EI1040 amplifier from LabJack to amplify the thermocouple signal. If you aren't in a rush for your measurements and the temperature is changing slowly, you can also use DAQFactory to oversample by taking multiple measurements and averaging them. This is done by checking the Avg? column of your input channel in the channel table and entering the number of oversample points in the #Avg column.
**UE9 Pro**: the UE9 Pro gets you an additional 2 bits of resolution over the UE9 (27 microvolts noise free minimum step, 5 microvolts RMS) which is enough to read a thermocouple directly with a reasonable amount of precision. You can use an amplifier as well if you want higher speed readings.

Now that we have that covered lets go over using both methods: unamplified and amplified.

### 9.3.1 Unamplified Thermocouple Readings

This applies only to the UE9 and UE9 Pro because the U3 doesn't have the precision to read the low voltages output by a thermocouple. Please see the last section (7.5) for a full explanation.

1) Right click on **SEQUENCES**: in the Workspace and select **Add Sequence**. Give it the name **StartUp** and click OK.

2) In the sequence editor, check the box labeled **Auto-Start**, then put the following in the script:

   ```
   global offset = 0
   using("device.labjack.")
   include("c:\program files\labjack\drivers\labjackud.h")
   AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 18, 0,0)
   AddRequest(0, LJ_ioPUT_AIN_RANGE, 0, LJ_rgUNI5V, 0, 0)
   GoOne(0)
   ```

   The first line is used to adjust for any voltage offset in the signal. The next two lines are the standard lines required to access the LabJack UD from script. The next three lines set the gain and resolution of the UE9. We've chosen the most generic range for AIN0. Your UE9 might be able to use a tighter AIN range and achieve higher precision.

3) Click **Apply**, then go to **Debug - Run this Sequence** from the DAQFactory menu to actually set these settings.

4) Create a channel for AIN0 as described back in 3.1 to read the voltage.

5) To accurately read the thermocouple we also need to read the temperature of the cold junction which is essentially the temperature of the LabJack. This is internal A to D channel #133. Create another channel called CJC to read A to D channel #133. Click **Apply** to save your changes and start reading the inputs.

Now we can convert the thermocouple voltage input into temperature:

6) Click on **CONVERSIONS**: in the workspace, then **Add**. Call the conversion Thermocouple and put the following Formula:

   ```
   TypeK(Value - Offset, CJC[0]-273.15)
   ```
There are different formulas for each thermocouple type, such as TypeJ().

7) Go back to the channel table and for the AIN0 channel you created in step 4, select the Thermocouple conversion you just created and click Apply.

That is it. You can now display or plot the input channel. You can adjust the offset variable to adjust for any bias in the LabJack by changing it in the sequence we created in step 2 and rerunning it, or by creating a screen control to adjust its value (shown in the sample).

Sample file: LJJGuideSamples\UE9-Thermocouple.ctl

The sample doesn't use a conversion, but instead puts the formula in the components themselves. Conversions are typically easier since you typically only want to see the temperature, however, for calibration you often want the raw voltage and so can't use a conversion.

9.3.2 Amplified Thermocouple Readings

When using an amplifier, the steps are pretty much identical to the unamplified version except we need to adjust the input based on the amplification:

1) Right click on SEQUENCES: in the Workspace and select Add Sequence. Give it the name StartUp and click OK.

2) In the sequence editor, check the box labeled Auto-Start, then put the following in the script:

```
global offset = 0
global gain = 51
```

If using a UE9, you can add these lines as well:

```
using("device.labjack.")
include("c:\program files\labjack\drivers\labjackud.h")
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 18, 0,0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 0, LJ_rgUNI5V, 0, 0)
GoOne(0)
```

The first line is used to adjust for any voltage offset in the signal. The second to adjust for the gain of the amplifier. The next two optional UE9 lines are the standard lines required to access the LabJack UD from script. The next three lines set the gain and resolution of the UE9. We've chosen the most generic range for AIN0. Your UE9 might be able to use a tighter AIN range and achieve higher precision.

3) Click Apply, then go to Debug - Run this Sequence from the DAQFactory menu to actually set these settings.

4) Create a channel for AIN0 as described back in 3.1 to read the voltage.

5) To accurately read the thermocouple we also need to read the temperature of the cold junction
which is essentially the temperature of the LabJack. This is internal A to D channel #133 for the UE9 and #30 for the U3. Create another channel called CJC to read A to D channel #133 if using a UE9 or #30 for a U3. Click Apply to save your changes and start reading the inputs.

Now we can convert the thermocouple voltage input into temperature:

6) Click on **CONVERSIONS**: in the workspace, then **Add**. Call the conversion Thermocouple and put the following Formula:

\[ \text{TypeK}((\text{Value} - \text{Offset})/\text{gain}, \text{CJC}[0]-273.15) \]

There are different formulas for each thermocouple type, such as TypeJ().

7) Go back to the channel table and for the AIN0 channel you created in step 4, select the Thermocouple conversion you just created and click **Apply**.

That is it. You can now display or plot the input channel. You can adjust the offset variable to adjust for any bias in the LabJack by changing it in the sequence we created in step 2 and rerunning it, or by creating a screen control to adjust its value (shown in the samples). Same thing with the gain.

It doesn’t really matter what amplifier you use. You can just adjust the offset and gain appropriately.

Sample file: **LJGuideSamples\U3_LJTIA_Thermocouple.ctl**

Sample file: **LJGuideSamples\UE9_LJTIA_Thermocouple.ctl**

These sample doesn't use a conversion, but instead puts the formula in the components themselves. Conversions are typically easier since you typically only want to see the temperature, however, for calibration you often want the raw voltage and so can't use a conversion.

Sample file: **LJGuideSamples\U3_LJTIA_Thermocouple_Conversion.ctl**

This is identical to the U3_LJTIA_Thermocouple sample, but uses a Conversion to convert from volts to temperature. You can see the Conversion by clicking **CONVERSIONS**: in the workspace. A Conversion converts the data as it comes in so the channel (FIO4) shows temperature immediately. You never see volts. Conversions can also be applied to multiple channels. That said, for thermocouples, you will probably want a unique conversion for each thermocouple since each input might have a slightly different Offset and Gain. In this case, you'd need to replace "Offset" and "Gain" in the conversion with the actual number for the corresponding input once you have figured these out.

**9.1 - Low Speed Acquisition (< 100hz)**

Log in or register to post comments
9.1.1 Introduction

Because Windows needs to occasionally use the CPU in your computer to redraw the screen, process the mouse, and perform other tasks, you generally cannot read inputs or control outputs faster than 100hz using the computer as the timer. Of course with the new multicore CPU's, you may be able to tweak extra speed out DAQFactory’s polling loops, essentially putting the acquisition on one core, and letting Windows use the other core for display. Doing this sort of thing requires more advanced techniques described in the last chapter. This section describes how to do software (i.e. DAQFactory) polled reading and writing of analog and digital inputs and outputs. The techniques are essentially the same for both analog and digital inputs and outputs.

9.1.2 The Easy Way - With Channels

The easiest way to read analog and digital inputs at low speeds (i.e. < 100hz) or set analog and digital outputs is to use channels as we did in the Basic I/O chapter. No scripting is required, you can easily convert the readings using Conversions, and log it using Logging sets. For the U3, which allows the specification of the differential channel, you can simply put the channel number for the negative side in the Quick Note / Special / OPC column of the channel. If you need more advanced configuration, you can still use channels, combined with some basic scripting to send the configuration commands to the LabJack. For example, to set the UE9 resolution to 14 bit and the range on channels 2 and 3 to +/-5V:

```plaintext
using("device.labjack.")
include("c:\program files\labjack\drivers\labjackud.h")
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 14, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 2, LJ_rgBIP5V, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 3, LJ_rgBIP5V, 0, 0)
GoOne(0)
```

This was explained earlier in the chapter on scripting.

9.1.3 Basic Scripting using eGet

Channels, however, aren't always the best choice, and sometimes you need script. The next step up from channels is to use basic scripting and the eGet() function to retrieve a single value, or ePut() to set a single value. For example, to read channel 1 and then put the result into a channel called MyChannel we would do:

```plaintext
private err
private val
private string message
while(1)
   err = eGet(0, LJ_ioGET_AIN, 1, @val, 0)
   if (err)
      ErrorToString(err, @message)
      ? message
```
else
    MyChannel.AddValue(val)
endif
delay(1)
endwhile

Now truthfully, this next code snippet does the exact same thing, provided MyChannel is setup to read channel 1:

while(1)
    read(MyChannel)
    delay()
endwhile

But, this code does not allow for any direct error handling, and of course doesn't demonstrate the eGet function! eGet is also more useful when you don't know your channel numbers at runtime. In the first example, we used scalar values in our eGet() function call, but there is no reason why you couldn't use variables that could then be changed from elsewhere:

err = eGet(ID, LJ_ioGET_AIN, chan, @val, 0)

Another thing the first example shows is the AddValue() function of the channel MyChannel. This function essentially stuffs a value into a channel. This allows you to utilize the benefits of a channel (history, easy logging, etc), without actually using the channel Timing to trigger the reads. In this case, we are putting the result of the eGet call into MyChannel. MyChannel does not have to have the same channel number, I/O type, or even be Device Type "LabJack".

Note: if using the U3, you will need to configure the pins as analog inputs. With Channels, this is done automatically for you, but when scripting you have to do it yourself. It only takes one line to enable a channel:

ePut(ID, LJ_ioPUT_ANALOG_ENABLE_BIT, 1, 1, 0)

Sample file: LJGuideSamples\eGet.ctl

9.1.4 More Advanced Using Add / Go / Get

eGet can be handy for very simple scripts, but when doing a number of different requests, it is much better to use the AddRequest() / GoOne() / GetResult() functions. So, if we wanted to read channels 1 through 3 we could do:

eGet(0, LJ_ioGET_AIN, 1, @val1, 0)
eGet(0, LJ_ioGET_AIN, 2, @val2, 0)
eGet(0, LJ_ioGET_AIN, 3, @val3, 0)
or we could do:

```
AddRequest(0, LJ_iowGET_AIN, 1, 0, 0, 0)
AddRequest(0, LJ_iowGET_AIN, 2, 0, 0, 0)
AddRequest(0, LJ_iowGET_AIN, 3, 0, 0, 0)
GoOne(0)
GetResult(0, LJ_iowGET_AIN, 1, @val1)
GetResult(0, LJ_iowGET_AIN, 2, @val2)
GetResult(0, LJ_iowGET_AIN, 3, @val3)
```

OK, you may think that eGet looks much easier since its only three lines of code vs. seven, and it is easier on a basic level, but with ease you lose flexibility and efficiency. Using the second method is more efficient internally. The second method also allows you to do error handling easier using the GetNextError() function. With error handling, the eGet code ends up looking like this:

```
err = eGet(0, LJ_iowGET_AIN, 1, @val1, 0)
if (err)
    ... error!
endif
err = eGet(0, LJ_iowGET_AIN, 2, @val2, 0)
if (err)
    ... error!
endif
err = eGet(0, LJ_iowGET_AIN, 3, @val3, 0)
if (err)
    ... error!
endif
```

But the Add / Go / Get looks like this:

```
AddRequest(0, LJ_iowGET_AIN, 1, 0, 0, 0)
AddRequest(0, LJ_iowGET_AIN, 2, 0, 0, 0)
AddRequest(0, LJ_iowGET_AIN, 3, 0, 0, 0)
GoOne(0)
GetResult(0, LJ_iowGET_AIN, 1, @val1)
GetResult(0, LJ_iowGET_AIN, 2, @val2)
GetResult(0, LJ_iowGET_AIN, 3, @val3)
while (GetNextError(1,@io,@ch,@err)
    ... error!
endwhile
```

As you can see the second method is much cleaner and easier to read. The eGet() version would get worse and worse as you added more function calls. Using the second method also allows you to create a single error handler for the entire block, or as shown in the section on error handling, you can create a single function to do all your error handling for all your scripts.
Note: variable declarations are not shown in the above examples, but would be required. Likewise, ... error! would need to be replaced with script to actually do something in the case of an error.

9.1.5 Controlling outputs

We've talked a little about setting an output from a screen control in section 4.6, but this requires user input. We also talked about setting an output based on an input in section 7.1. In this section we saw that you can easily set an output channel by simply assigning a value to it. So, to set a DAC channel named ValvePosition to 3 volts (assuming no Conversion exists on the channel), we would simply do:

```
ValvePosition = 3
```

You can also apply conversions to outputs like we did with inputs in section 4.3, however, the conversions work in reverse. For inputs, the conversion takes the raw voltage (counts or other units) from the LabJack and converts it into engineering units like temperature or pressure. For outputs, the conversion takes the engineering units and converts into voltage. So, for if we had a proportional valve that takes a 0 to 5V signal and we want to be able to specify the percentage open, where 0% = 0V and 100% = 5V, the conversion would be:

```
Value / 20
```

Then, to open the valve to 60% we could just do:

```
ValvePosition = 60
```

and 3 volts would be outputted from the DAC of the LabJack.

Now as a further example, lets say we'd like to ramp the valve position from 0 to 100% over 60 seconds in steps of 1%. The script is quite simple:

```
// initialize valve position to 0
ValvePosition = 0
while (ValvePosition < 100)
    ValvePosition = ValvePosition + 1
    delay(0.6)
endwhile
```

Doing a ramp and soak is not much harder, just split out the while loops. Lets say we want to ramp to 40% in 20 seconds, soak for 10 seconds, ramp to 80% in 40 seconds, soak for 5 seconds, then ramp back down to 0 in 60 seconds:

```
// initialize valve position to 0
ValvePosition = 0
// ramp to 40
while (ValvePosition < 40)
    ValvePosition = ValvePosition + 1
// soak for 10 seconds
delay(10)
// ramp to 80
while (ValvePosition < 80)
    ValvePosition = ValvePosition + 1
// soak for 5 seconds
delay(5)
// ramp back down to 0
while (ValvePosition > 0)
    ValvePosition = ValvePosition - 1
```

21 Jun 2019
One important note: these sequences won't work if you try and ramp past the highest value of the output. If you try and set an output to an invalid value you will get an Alert and the channel WILL NOT update, so the while() will never exit because ValvePosition would never reach the desired point.

Sample file: LJGuideSamples\RampSample.ctl

9.2 - High Speed Acquisition - Streaming

9.2.1 Introduction

When you want to do things faster than 100hz, its usually best to let the LabJack perform the timing. The LabJack can then send data back in blocks and relieve your CPU from constantly having to do things. The LabJacks support streaming of both digital and analog inputs as well as timers and counters. This section explains how to setup and performed streamed data acquisition.

9.2.2 Basic Streaming

When you want to read inputs at rates faster than 100hz, or at very precise intervals, it is usually best to use the LabJack’s stream mode instead of having DAQFactory and the PC set the read rates. Anything above 100hz is difficult for a PC to perform since it has so many other tasks to do as well. To setup streaming in DAQFactory you will need to use a combination of Channels and sequence script. Streaming in DAQFactory is different from how the LabJack User Manual describes, as DAQFactory handles all the callback and data collection, putting the data into the appropriate channels. In this sample we’ll stream 2 channels.

1) Start DAQFactory up with a new document.
2) Click on **CHANNELS**: in the Workspace to go to the channel table.

3) Add two new channels, ChannelA and ChannelB, both **Device Type** = LabJack, **D#** = 0, **I/O Type** = A to D, and **Channel** numbers 2 and 3. Set the **Timing** for both channels to 0, and the **History**: set to 36000.

3) Click on **Apply** to save your changes.

4) Right click on **SEQUENCES**: in the Workspace and select **Add Sequence**. Call the new sequence **StartStream**

5) In the sequence editor window that appears, enter the following script:

```javascript
// standard initialization:
using("device.labjack")
include("c:\program files\LabJack\Drivers\labjackud.h")
// setup stream:
// set scan rate:
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chSTREAM_SCAN_FREQUENCY, 1000, 0, 0)
// setup channels to stream:
AddRequest(0, LJ_ioCLEAR_STREAM_CHANNELS, 0, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 2, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 3, 0, 0, 0)
GoOne(0)
// start the stream:
global scanrate = 0
eGet(0,LJ_ioSTART_STREAM, 0, @scanrate, 0)
// scanrate now has the actual scanrate, which you can display on the screen if you want.
```

6) Click on **Apply and Compile** to save your script.

7) Right click on **SEQUENCES**: in the Workspace and select **Add Sequence**. Call the new sequence **StopStream** and enter the following script:

```javascript
ePut(0,LJ_ioSTOP_STREAM, 0, 0, 0)
```

8) Click on **Apply and Compile** to save your script.

9) Click on **Page_0** under **PAGES**: in the workspace to display a blank page.

10) On that page, right click and select **Graphs - 2D Graph**.

11) While holding down the Ctrl key, click and drag the graph to move it to the top left corner of the page, then click and drag the bottom right corner of the graph to expand it to take up about 3/4 of the screen.

12) Right click on the graph and select **Properties....** For the **Y Expression** put **ChannelA**. Click on **New Trace** and for the **Y Expression** put **ChannelB**. Click **OK** to close the properties window.
13) Right click somewhere on the empty part of the page and select **Buttons - Button**. Right click on the new button and select **Properties**.

14) For **Caption**, put **Start**, then go to the **Action** tab and select **Start/Stop Sequence**, then select your **StartStream** sequence.

15) Repeat steps 13 and 14, but put **Stop** for the caption and **StopStream** for the sequence.

That is it. You should be able to click on the Start button and have streaming on channels 2 and 3 start up and be graphed. It is possible that the values will be off the scale of the graph, so you may want to click on the graph, then right click on the graph and select **AutoScale - AutoScale Y**.

One important point if you start tweaking this sample: the Channels that you created must have the same D# and channel number as the one you specified in the **LJ_ioADD_STREAM_CHANNEL** request. The I/O Type must be "A to D" as well, even if you are streaming digital inputs, timers or counters. If not then DAQFactory won’t know where to put the data that is streaming in.

**Note**: make sure you configure your inputs before starting the stream. For the U3, this means you have to set the pins to analog input as shown in the example file.

**Note**: you should not change the **LJ_chSTREAM_WAIT_MODE**, as all waiting is handled internally. If you change this, you will most likely cause streaming to stop functioning.

Sample file: **LJGuideSamples\BasicStream.ctl**

### 9.2.3 Streaming Other Inputs

You can stream other inputs besides the analog inputs of your LabJack. This is done by specifying special channel numbers when doing **LJ_ioADD_STREAM_CHANNEL**. The important part here is that even though the LabJack is actually streaming something other than an analog input, you MUST specify A to D for the I/O Type when creating your DAQFactory Channels to receive the data.

The available channel numbers are slightly different for each LabJack and listed here for your reference:

**U3:**

- 193 EIO_FIO
- 200 Timer0
- 201 Timer1
- 210 Counter0
You may notice that in DAQFactory, we have multiple TC_Capture channel numbers, where the LabJack documentation only lists one. This is to allow you to stream the high order word of multiple timers and counters and keep the data separate in separate channels. Internally, they are exactly the same, so you have to specify the appropriate TC_Capture immediately following its Timer or Counter channel #. In other words, to read the entire 32 bits of Timers 0 and 1, you'd
do:

AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 200, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 224, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 201, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 225, 0, 0, 0)

9.2.4 Triggered

Triggered streaming is currently only supported by the UE9 and UE9-Pro.

Triggered streaming is similar to regular streaming except instead of using the internal LabJack clock to determine when a scan of the stream channels occurs, an external pulse triggers the scan. The interval between external pulses must be less than the maximum stream rate for the current input resolution. The external pulses do not need to occur at a constant interval. To enable external triggering, just add the following line of script before adding your stream channels using LJ_ioADD_STREAM_CHANNEL:

AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSTREAM_EXTERNAL_TRIGGER, 1, 0, 0)

The trigger input will be the first available FIO pin based on which timers and counters you have enabled.

The only problem with triggered streaming is that the time of each data point will be off. This is because the LabJack buffers the scans and DAQFactory doesn't actually get the data until a full packet occurs. DAQFactory doesn't realize this and assigns times based on an assumed interval. If you have the bandwidth, i.e. your pulses are slow enough that you aren't close to the stream interval limit, you can use the system timer mode of the timers to retrieve exact relative times of your scans. To do this, you need to setup a timer for system timer in, and then add the timer to the list of channels to stream. Depending on how long your experiment runs, you may be able to get away with only SYSTIMERLOW. For the UE9 at 750khz, the low timer will roll over every 5726 seconds. Here's how to do it:

1) Enable two Timers:

AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 2, 0, 0)

2) Set the mode:

AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmSYSTIMERLOW, 0, 0)
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 1, LJ_tmSYSTIMERHIGH, 0, 0)
GoOne(ID)
3) Set up the stream to stream analog input 2 and 3 in external trigger mode:

```plaintext
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSTREAM_EXTERNAL_TRIGGER, 1, 0, 0)
// setup channels to stream:
AddRequest(ID, LJ_ioCLEAR_STREAM_CHANNELS, 0, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 3, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 4, 0, 0, 0)
```

4) Now we need to add our timers to the list of channels to stream. Make sure you use the order indicated:

```plaintext
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 200, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 224, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 201, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 225, 0, 0, 0)
```

5) Now finish up the stream setup:

```plaintext
GoOne(ID)
// start the stream:
global scanrate = 0
eGet(ID,LJ_ioSTART_STREAM, 0, @scanrate, 0)
// scanrate now has the actual scanrate, which you can display on the screen if you want.
```

6) Create 4 channels to receive this timing data in addition to the 2 you created to receive the analog input. All 6 channels are I/O type: A to D, **Timing = 0. Channel #**s will be 3, 4, 200, 201, 224, and 225.

That completes it. When you run your script, the stream will start, streaming both analog inputs 2 and 3 as well as the system timer. A scan will occur every time a trigger is detected on FIO2. FIO0 and FIO1 are used by the 2 timers. With each scan, your six channels will update. The time associated with these channels will be incorrect, but channels 200, 224, 201, and 225 will contain the 4 words that make up the 64 bit system timer value. While this is not absolute time, it will give you relative time between each triggered scan. Just use the difference in counts divided by the system clock speed of 750khz for the UE9 to determine the actual number of seconds between scans. The best way to do this is to create a calculated V channel:

7) Right click on **CHANNELS:** under **V:** in the Workspace. Note this is note the same **CHANNELS:** that we’ve been clicking before. Select **Add V Channel**

8) Call the new channel **TheTime**

9) Click on the new channel in the workspace. In the **Expression** window, put:

```plaintext
(TimerLowLow + TimerLowHigh*2^16 + TimerHighLow * 2^32 + TimerHighHigh * 2^48)
```
This assumes you named channel 200 TimerLowLow, 224 TimerLowHigh, 201 TimerHighLow, and 225 TimerHighHigh. You also may want to put a divisor at the end to convert to seconds:

\[ \text{UE9:} \]

\[
\frac{\left(\text{TimerLowLow} + \text{TimerLowHigh} \times 2^{16} + \text{TimerHighLow} \times 2^{32} + \text{TimerHighHigh} \times 2^{48}\right)}{750000}
\]

10) Click **Apply**.

At this point, you can reference this channel like you would any other, except putting V. in front of it. Instead of getting a channel reading, you'll get the result of the calculation. Since we didn't use any [0] notation, this is the entire array calculated from all the readings. If you want to graph your channels, you'd put:

\[ V.\text{TheTime} \]

as the **X Expression** in place of Time. You'll need to change the bottom axis type to **Lin**, undo **Use Time Width**, and adjust the **Scale From** and **Scale To**:

All of this is shown, complete, in the sample file:

Sample file: **LJGuideSamples\TriggeredStream.ctl**

### 9.2.5 Error Handling for Streaming

Streaming from a LabJack is what is called an asynchronous action. This means that the LabJack does its own thing and every so often it tells DAQFactory that there is new data or there is an error. For this reason, you cannot simply look at the error code returned by **LJ_ioSTART_STREAM** to catch all stream errors. This command may return a stream configuration type error, so you'll want to check for it using the same methods as the low speed acquisition, but will not handle errors in the actual stream. For this you have two choices:
1) You can create a simple sequence to retrieve the last stream error continuously and do something if it returns an error. The function GetLastStreamError() will return the code for the last stream error. This is reset when you start the stream. This, however, is not the best way to do this and will waste processor power.

2) You can use the DAQFactory OnAlert event, which only gets called when an actual error occurs. Using the event as an error handler is described in the previous section on error handling, and catching stream errors would be done the same way. One common error you might want to catch is if your LabJack accidentally gets unplugged while streaming. If you wanted to automatically restart streaming when it is reconnected you could do this in your OnAlert sequence:

```plaintext
if ((left(strAlert,10) == "D0050:00:2001") && (Streaming)) // 2001 is LJE_RECONNECT
    StartStream()
endif
```

Now, this assumes that you have a sequence called StartStream that will reconfigure the LabJack and actually restart the stream. It also assumes that you've created a global variable called "Streaming" that you set to 1 in StartStream, and to 0 in StopStream. This is so an accident unplug when you aren't streaming doesn't spontaneously start the streaming process. Finally this assumes you are using device number 0 / first found. Please see the section on OnAlert if you are using multiple LabJacks.

### 9.3 - Thermocouples

Log in or register to post comments

Thermocouples are a very common way to measure temperature. They come in a wide variety of forms and have a large range. Do to their small voltage output and something called a cold junction, they are, however, slightly more challenging to read. Most applications need a temperature range of about -50 to +150°C, though, and for these applications you might want to consider using a silicon type sensor, as they are cheaper, more accurate and much easier to use. Examples of these are the EI-1022 and EI-1034 sold by LabJack.

Thermocouples are simply two different metals that meet at a single point. Due to a general property of metals, a small voltage is generated that is proportional to the temperature of this junction. There are many different types of thermocouples, identified by a letter such as J type or K type which determine the metals used which then determines the temperature range and voltage output scaling. In all cases, this voltage is between about -5 and 5 millivolts, a rather small voltage.

It gets more complicated though: since the terminals on your DAQ device (the LabJack) are made of yet another metal, there are two more points where different metals are touching which are generating a voltage proportional to the temperature of this terminal. This is called the cold
junction. In order to get an accurate thermocouple reading, you have to adjust for the cold junction. This is called cold junction compensation.

Finally, we should mention that most thermocouples are only accurate to about 1 or 2 degrees C, though calibration can help a little with this.

Now that you have the basics down we can talk about how to read a thermocouple. There are several choices depending on what LabJack you have:

**U3:** the U3 has a resolution of 12 bit and a range of +/-2.44, the minimum voltage you can read is about 1.2 millivolts (4.88 volts / 4096 steps). Because a thermocouple typically outputs around 40 microvolts per degC, you only get a precision of about 30 degrees! Obviously that isn't going to work so you have to use an amplifier such as the LJTIA or EI1040, both available from LabJack, to amplify that millivolt signal before it gets to the converter.

**UE9:** the UE9 has a resolution of 16 bit so has 65536 steps instead of the 4096 of the U3. This helps some. The UE9 also has some built-in gain. Typically you get about 76 microvolts minimum step noise free, and 15 microvolts RMS. This means you get about 1 or 2 degrees C of resolution. This isn't bad and might work. For the exact specs, please review Appendix B of the UE9 Users Guide. If you need better than this, then you will need to get an LJTIA or EI1040 amplifier from LabJack to amplify the thermocouple signal. If you aren't in a rush for your measurements and the temperature is changing slowly, you can also use DAQFactory to oversample by taking multiple measurements and averaging them. This is done by checking the Avg? column of your input channel in the channel table and entering the number of oversample points in the #Avg column.

**UE9 Pro:** the UE9 Pro gets you an additional 2 bits of resolution over the UE9 (27 microvolts noise free minimum step, 5 microvolts RMS) which is enough to read a thermocouple directly with a reasonable amount of precision. You can use an amplifier as well if you want higher speed readings.

Now that we have that covered lets go over using both methods: unamplified and amplified.

### 9.3.1 Unamplified Thermocouple Readings

This applies only to the UE9 and UE9 Pro because the U3 doesn't have the precision to read the low voltages output by a thermocouple. Please see the last section (7.5) for a full explanation.

1) Right click on **SEQUENCES:** in the Workspace and select **Add Sequence.** Give it the name StartUp and click OK.

2) In the sequence editor, check the box labeled **Auto-Start,** then put the following in the script:

```plaintext
global offset = 0
using("device.labjack."
include("c:\program files\labjack\drivers\labjackud.h")
```
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 18, 0,0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 0, LJ_rgUNI5V, 0, 0)
GoOne(0)

The first line is used to adjust for any voltage offset in the signal. The next two lines are the standard lines required to access the LabJack UD from script. The next three lines set the gain and resolution of the UE9. We've chosen the most generic range for AIN0. Your UE9 might be able to use a tighter AIN range and achieve higher precision.

3) Click **Apply**, then go to **Debug - Run this Sequence** from the DAQFactory menu to actually set these settings.

4) Create a channel for AIN0 as described back in 3.1 to read the voltage.

5) To accurately read the thermocouple we also need to read the temperature of the cold junction which is essentially the temperature of the LabJack. This is internal A to D channel #133. Create another channel called CJC to read A to D channel #133. Click **Apply** to save your changes and start reading the inputs.

Now we can convert the thermocouple voltage input into temperature:

6) Click on **CONVERSIONS**: in the workspace, then **Add**. Call the conversion Thermocouple and put the following Formula:

TypeK(Value - Offset, CJC[0]-273.15)

There are different formulas for each thermocouple type, such as TypeJ().

7) Go back to the channel table and for the AIN0 channel you created in step 4, select the Thermocouple conversion you just created and click **Apply**.

That is it. You can now display or plot the input channel. You can adjust the offset variable to adjust for any bias in the LabJack by changing it in the sequence we created in step 2 and rerunning it, or by creating a screen control to adjust its value (shown in the sample).

Sample file: **LJGuideSamples\UE9_Thermocouple.ctl**

The sample doesn't use a conversion, but instead puts the formula in the components themselves. Conversions are typically easier since you typically only want to see the temperature, however, for calibration you often want the raw voltage and so can't use a conversion.

### 9.3.2 Amplified Thermocouple Readings

When using an amplifier, the steps are pretty much identical to the unamplified version except we need to adjust the input based on the amplification:

1) Right click on **SEQUENCES**: in the Workspace and select **Add Sequence**. Give it the name
2) In the sequence editor, check the box labeled **Auto-Start**, then put the following in the script:

```plaintext
global offset = 0
global gain = 51
```

If using a UE9, you can add these lines as well:

```plaintext
using("device.labjack.")
include("c:\program files\labjack\drivers\labjackud.h")
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 18, 0,0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 0, LJ_rgUNI5V, 0, 0)
GoOne(0)
```

The first line is used to adjust for any voltage offset in the signal. The second to adjust for the gain of the amplifier. The next two optional UE9 lines are the standard lines required to access the LabJack UD from script. The next three lines set the gain and resolution of the UE9. We've chosen the most generic range for AIN0. Your UE9 might be able to use a tighter AIN range and achieve higher precision.

3) Click **Apply**, then go to **Debug - Run this Sequence** from the DAQFactory menu to actually set these settings.

4) Create a channel for AIN0 as described back in 3.1 to read the voltage.

5) To accurately read the thermocouple we also need to read the temperature of the cold junction which is essentially the temperature of the LabJack. This is internal A to D channel #133 for the UE9 and #30 for the U3. Create another channel called CJC to read A to D channel #133 if using a UE9 or #30 for a U3. Click Apply to save your changes and start reading the inputs.

Now we can convert the thermocouple voltage input into temperature:

6) Click on **CONVERSIONS**: in the workspace, then **Add**. Call the conversion Thermocouple and put the following Formula:

```plaintext
TypeK((Value - Offset)/gain, CJC[0]-273.15)
```

There are different formulas for each thermocouple type, such as TypeJ().

7) Go back to the channel table and for the AIN0 channel you created in step 4, select the Thermocouple conversion you just created and click **Apply**.

That is it. You can now display or plot the input channel. You can adjust the offset variable to adjust for any bias in the LabJack by changing it in the sequence we created in step 2 and rerunning it, or by creating a screen control to adjust its value (shown in the samples). Same thing with the gain.
It doesn't really matter what amplifier you use. You can just adjust the offset and gain appropriately.

Sample file: LJGuideSamples\U3_LJTIA_Thermocouple.ctl

Sample file: LJGuideSamples\UE9_LJTIA_Thermocouple.ctl

These sample doesn't use a conversion, but instead puts the formula in the components themselves. Conversions are typically easier since you typically only want to see the temperature, however, for calibration you often want the raw voltage and so can't use a conversion.

Sample file: LJGuideSamples\U3_LJTIA_Thermocouple_Conversion.ctl

This is identical to the U3_LJTIA_Thermocouple sample, but uses a Conversion to convert from volts to temperature. You can see the Conversion by clicking CONVERSIONS: in the workspace. A Conversion converts the data as it comes in so the channel (FIO4) shows temperature immediately. You never see volts. Conversions can also be applied to multiple channels. That said, for thermocouples, you will probably want a unique conversion for each thermocouple since each input might have a slightly different Offset and Gain. In this case, you'd need to replace "Offset" and "Gain" in the conversion with the actual number for the corresponding input once you have figured these out.