LabJack

LabJack U3 User's Guide

Revision 0.96 (Preliminary)
2/22/2006

LabJack Corporation
www.labjack.com
support@labjack.com

For the latest version of this and other documents, go to www.labjack.com.

LabJack UE9 User’s Guide Revision History

V1.00 released …

Table Of Contents

Table Of Figures

1. Installation on Windows

It is recommended to install the software before making a USB connection to the LabJack U3. The LabJack UD driver requires a PC running Windows 98, ME, 2000, or XP. For other operating systems, go to labjack.com for available support, if any. Software will be installed to the LabJack directory which defaults to c:\Program Files\LabJack\.

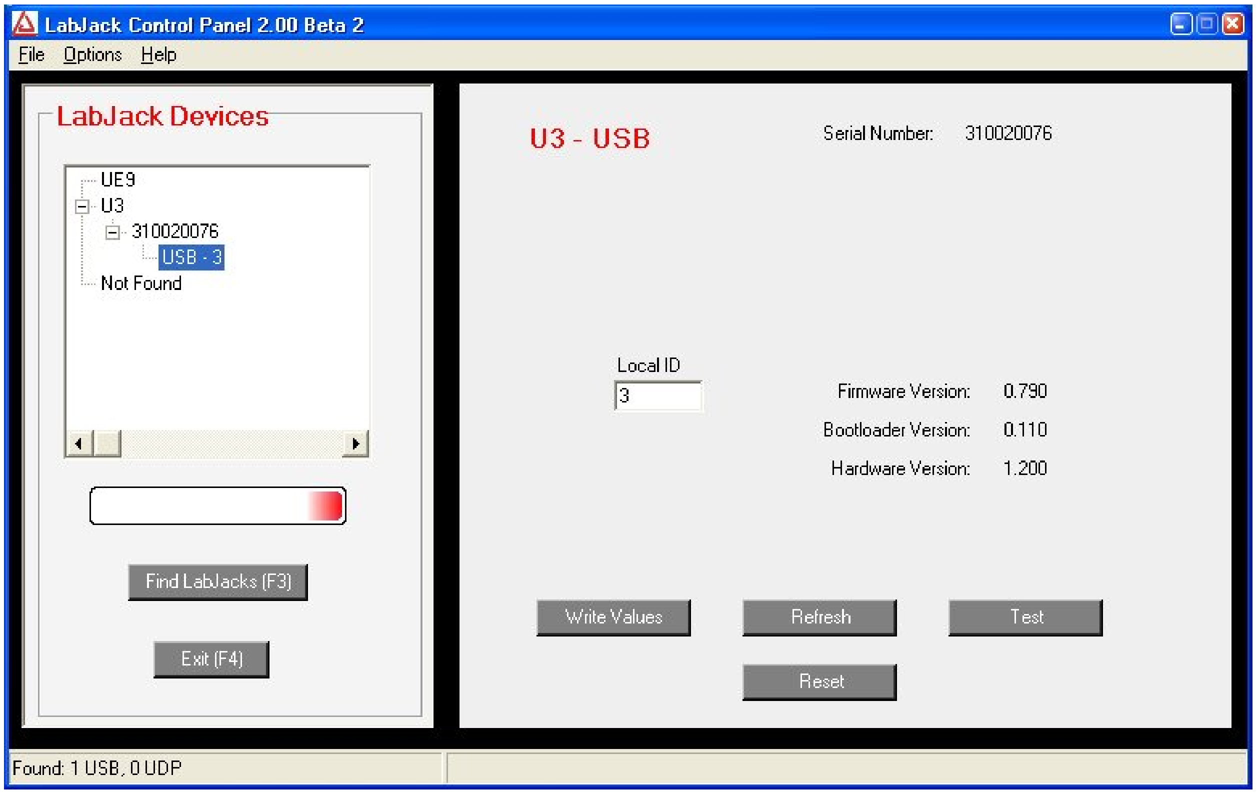
Check labjack.com for the latest software & drivers, but in order to install DAQFactory Express the CD must be used before installing updates.

When the USB cable is connected from the PC to the U3, on a USB port that has not enumerated a U3 before, Windows will bring up the add new hardware wizard. If this is the first time a U3 has been enumerated on any port on the PC, use the “specify location” option and browse to the appropriate driver folder. There is a folder for Windows 98/ME and another folder for Windows 2000/XP. These folders are installed in c:\Program Files\LabJack\drivers\install\UE9\. If a UE9 has been enumerated on the PC before, but just not on this particular port, the “install automatically” option can be used in the Windows new hardware wizard.

After installation of the software, run LJControlPanel to configure and test the unit. Then run LJSelfUpgrade to check for newer firmware.

1.1 Control Panel Application (LJControlPanel)

The LabJack Control Panel application (LJCP.exe) handles configuration and testing of the U3. Click on the “Find LabJacks” button to search for connected devices.



U3 - USB

Serial Number: 310020076

Local ID: 3

Firmware Version: 0.790

Bootloader Version: 0.110

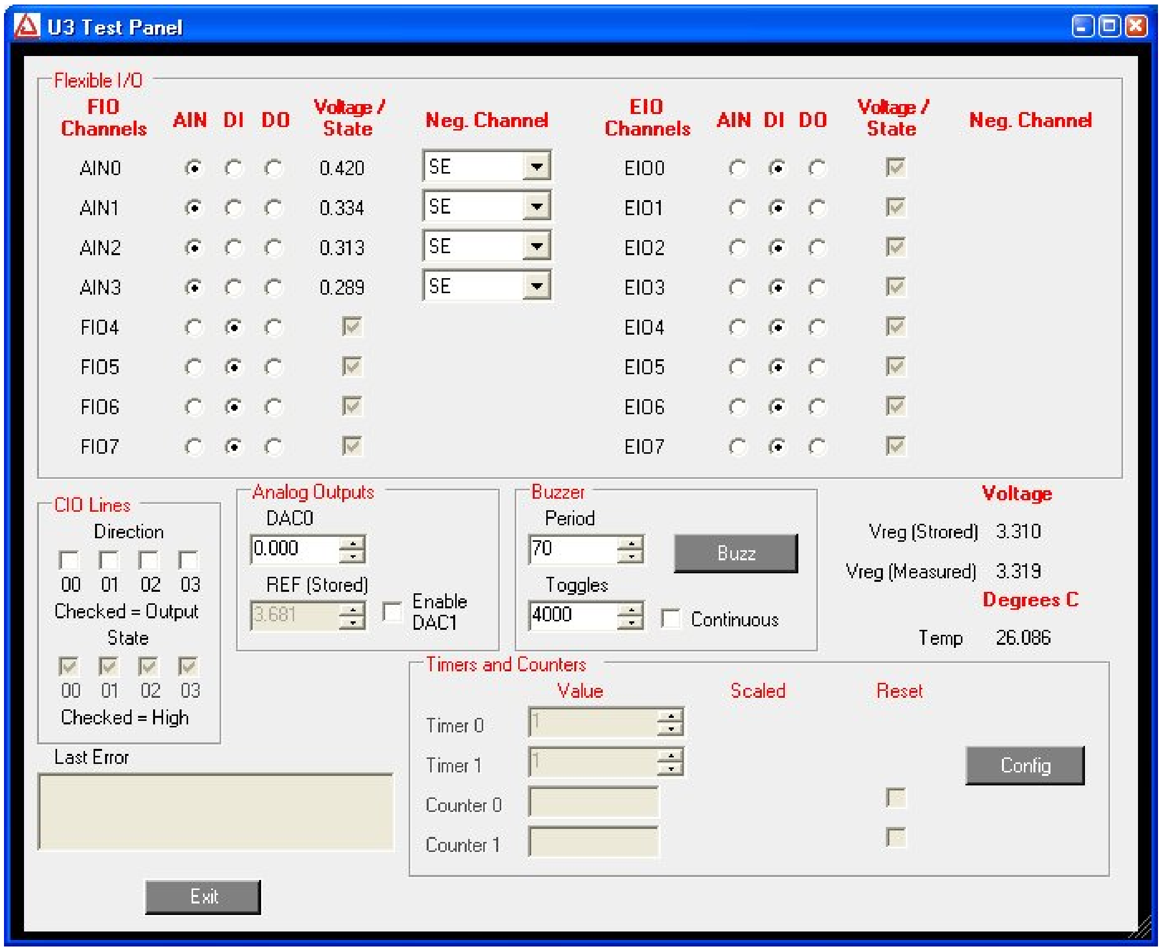
Hardware Version: 1.200

Found: 1 USB, 0 UDP

Figure 1-1. LJControlPanel Main Window

Figure 1-1 shows the results from a typical search. The application found one UE9 connected by USB and Ethernet. It also found a second UE9 that is accessible only by Ethernet. The USB connection has been selected in Figure 1-1, bringing up the configuration window on the right side.

- Refresh: Reload the window using values read from the device.
- Write to Device: Write the values from the window to the device. Depending on the values that have been changed, the application might prompt for a device reset.
- Reset: Click to reset the selected device.
- Test: Opens the window shown in Figure 1-2. This window continuously writes to and reads from the selected LabJack.



Flexible I/O

FIO Channels	AIN	DI	DO	Voltage / State	Neg. Channel	EIO Channels	AIN	DI	DO	Voltage / State	Neg. Channel
AIN0	●	○	○	0.420	SE	EIO0	○	●	○	✓	
AIN1	●	○	○	0.334	SE	EIO1	○	●	○	✓	
AIN2	●	○	○	0.313	SE	EIO2	○	●	○	✓	
AIN3	●	○	○	0.289	SE	EIO3	○	●	○	✓	
FIO4	○	●	○	✓		EIO4	○	●	○	✓	
FIO5	○	●	○	✓		EIO5	○	●	○	✓	
FIO6	○	●	○	✓		EIO6	○	●	○	✓	
FIO7	○	●	○	✓		EIO7	○	●	○	✓	

CIO Lines
Direction
☐ 00 ☐ 01 ☐ 02 ☐ 03
Checked = Output
State
✓ 00 ✓ 01 ✓ 02 ✓ 03
Checked = High

Analog Outputs
DAC0 0.000
REF (Stored) 3.681
☐ Enable DAC1

Buzzer
Period 70
Toggles 4000
☐ Continuous

Voltage
Vreg (Strored) 3.310
Vreg (Measured) 3.319
Degrees C
Temp 26.086

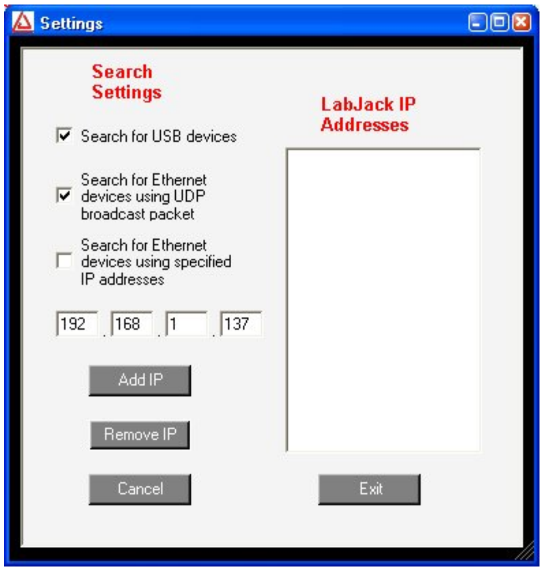
Last Error

Timers and Counters

	Value	Scaled	Reset
Timer 0	1		
Timer 1	1		
Counter 0			☐
Counter 1			☐

Figure 1-2. LJControlPanel U3 Test Window

Selecting Options=>Settings from the main LJControlPanel menu brings up the window shown in Figure 1-3. This window allows some features to of the LJControlPanel application to be customized.



[✓] Search for USB devices

[✓] Search for Ethernet devices using UDP broadcast packet

[ ] Search for Ethernet devices using specified IP addresses

192 . 168 . 1 . 137

Figure 1-3. LJControlPanel Settings Window

- Search for USB devices: If selected, LJControlPanel will include USB when searching for devices.
- Search for Ethernet devices using UDP broadcast packet: Does not apply to the U3.
- Search for Ethernet devices using specified IP addresses: Does not apply to the U3.

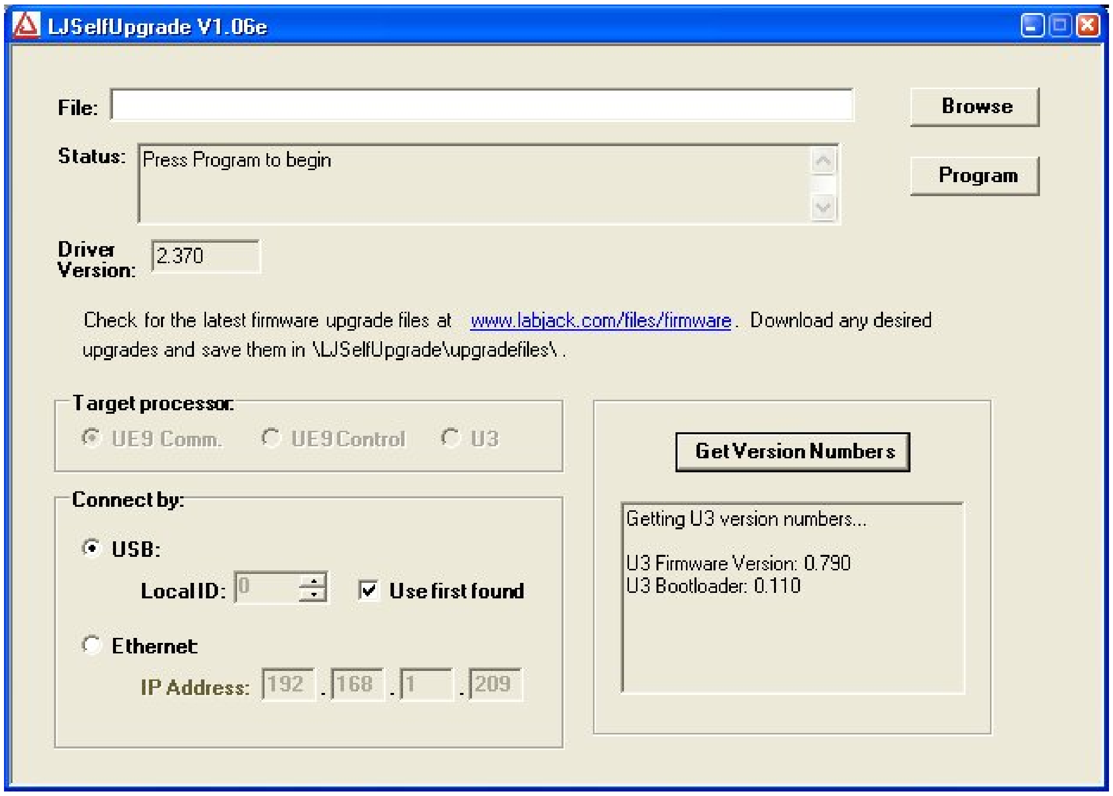
1.2 Self-Upgrade Application (LJSelfUpgrade)

The processor in the U3 has field upgradeable flash memory. The self-upgrade application shown in Figure 1-4 programs the latest firmware onto the processor.

USB is the only interface on the U3, and first found is the only option for self-upgrading the U3, so no changes are needed in the “Connect by:” box. There must only be one U3 connected to the PC when running LJSelfUpgrade.

Click on “Get Version Numbers”, to find out the current firmware versions on the device. Then use the provided Internet link to go to labjack.com and check for more recent firmware. Download firmware files to the …\LabJack\LJSelfUpgrade\upgradefiles\ directory.

Click the Browse button and select the upgrade file to program. Click the Program button to begin the self-upgrade process.



Status: Press Program to begin

Driver Version: 2.370

Check for the latest firmware upgrade files at www.labjack.com/files/firmware. Download any desired upgrades and save them in \LJSelfUpgrade\upgradefiles\ .

Target processor: [✓ UE9 Comm.] UE9 Control U3

Connect by: [✓ USB:] Local ID: 0 [✓ Use first found] Ethernet IP Address: 192 . 168 . 1 . 209

Getting U3 version numbers...

U3 Firmware Version: 0.790
U3 Bootloader: 0.110

Figure 1-4. Self-Upgrade Application

2. Hardware Description

The U3 has 3 different I/O areas:

- Communication Edge,
- Screw Terminal Edge,
- DB Edge.

The communication edge has a USB type B connector (with black cable connected in Figure 2-1). All power and communication is handled by the USB interface.

The screw terminal edge has convenient connections for the analog outputs and 8 flexible I/O (digital I/O, analog inputs, timers, or counters). The screw terminals are arranged in blocks of 4, with each block consisting of Vs, GND, and two I/O. There is also a status LED located on the left edge.

The DB Edge has a D-sub type connectors called DB15 which has the 8 EIO lines and 4 CIO lines.   The EIO lines are flexible like the FIO lines, while the CIO are dedicated digital I/O.

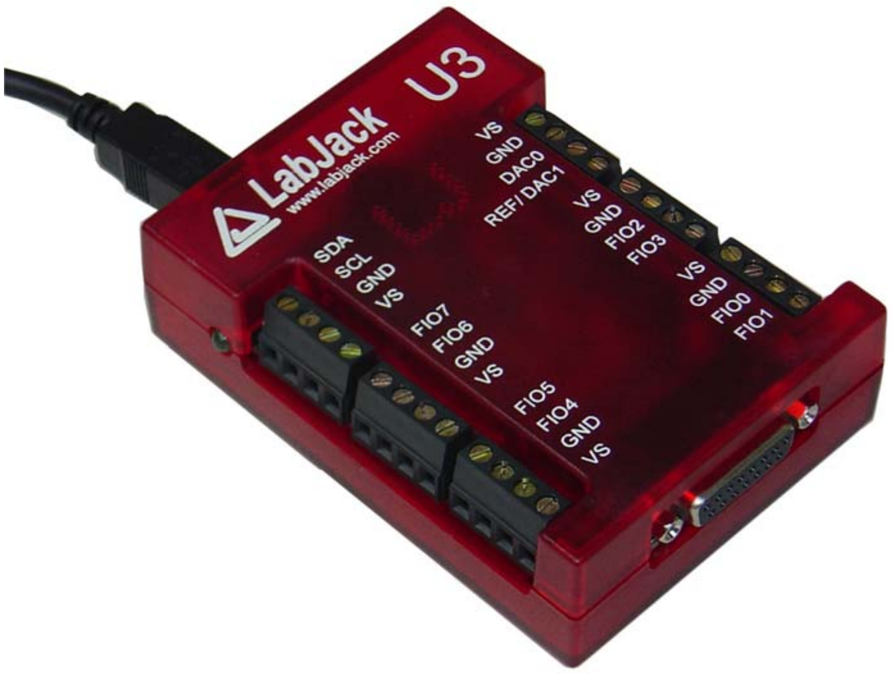


Figure 2-1. LabJack U3

2.1 USB

The U3 has a full-speed USB connection compatible with USB version 1.1 or 2.0. This connection provides communication and power (Vusb). USB ground is connected to the U3 ground (GND), and USB ground is generally the same as the ground of the PC chassis and AC mains.

The details of the U3 USB interface are handled by the high level drivers (Windows LabJackUD DLL), so the following information is really only needed when developing low-level drivers.

The USB interface consists of the normal bidirectional control endpoint 0 and two bidirectional bulk endpoints: Endpoint 1 and Endpoint 2. Endpoint 1 consists of a 128 byte OUT endpoint and a 128 byte IN endpoint. Endpoint 2 consists of a 0 byte OUT endpoint and a 256 byte IN endpoint. Endpoint 2 OUT is not supported by the firmware, and should never be used.

All commands should always be sent on Endpoint 1, and the responses to commands will also always be on Endpoint 1. Endpoint 2 is only used to send stream data from the U3 to the host.

2.2 Status LED

There is a green status LED on the LabJack U3. This LED blinks on reset, and then remains steadily lit.

2.3 GND and SGND

The GND connections available at the screw-terminals and DB connectors provide a common ground for all LabJack functions. This ground is the same as the ground line on the USB connection, which is often the same as ground on the PC chassis and therefore AC mains ground.

SGND is located on the screw terminal block with SDA and SCL. This terminal has a self-resetting thermal fuse in series with GND. This is often a good terminal to use when connecting the ground from another separately powered system that could unknowingly already share a common ground with the U3.

See the AIN, DAC, and Digital I/O Sections for more information about grounding.

2.4 Vs

The Vs terminals are designed as outputs for the internal supply voltage (nominally 5 volts). This will be the voltage provided from the USB cable. The Vs connections are outputs, not inputs. Do not connect a power source to Vs in normal situations.

2.5 Flexible I/O (FIO/EIO)

The first 16 I/O lines (FIO and EIO ports) on the LabJack U3 can be individually configured as digital input, digital output, or analog input. In addition, up to 2 of these lines can be configured as timers, and up to 2 of these lines can be configured as counters. If a line is configured as analog, it is called AINx according to the following table:

AIN0	FIO0	AIN8	EIO0
AIN1	FIO1	AIN9	EIO1
AIN2	FIO2	AIN10	EIO2
AIN3	FIO3	AIN11	EIO3
AIN4	FIO4	AIN12	EIO4
AIN5	FIO5	AIN13	EIO5
AIN6	FIO6	AIN14	EIO6
AIN7	FIO7	AIN15	EIO7

Table 2-1. Analog Input Pin Locations

Timers and counters can appear on various pins, but other I/O lines never move. For example, Timer1 can appear anywhere from FIO0 to EIO1, depending on TimerCounterPinOffset and whether Timer0 is enabled. On the other hand, FIO5 (for example), is always on the screw terminal labeled FIO5, and AIN5 (if enabled) is always on that same screw terminal.

The first 8 flexible I/O lines (FIO0-FIO7) appear on built-in screw terminals. The other 8 flexible I/O lines (EIO0-EIO7) are available on the DB15 connector.

Many software applications will need to initialize the flexible I/O to a known pin configuration. That requires calls to the low-level functions ConfigIO and ConfigTimerClock. Following are the values to set the pin configuration to the factory default state:

Byte #			
6	WriteMask	15	Write all parameters.
8	TimerCounterConfig	0	No timers/counters. Offset=0.
9	DAC1Enable	0	DAC1 disabled.
10	FIOAnalog	0	FIO all digital.
11	EIOAnalog	0	EIO all digital.

Table 2-x. ConfigIO Factory Default Values

Byte #			
8	TimerClockConfig	130	Set clock to 24 MHz.
9	TimerClockDivisor	0	Divisor = 0.

Table 2-x. ConfigTimerClock Factory Default Values

When using the high-level LabJackUD driver, this could be done with requests to the following IOTypes:

```
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 0, 0);
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 0, 0);
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tc24MHZ, 0);
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 0, 0);
ePut (lngHandle, LJ_ioPUT_COUNTER_ENABLE, 0, 0, 0);
ePut (lngHandle, LJ_ioPUT_COUNTER_ENABLE, 1, 0, 0);
ePut (lngHandle, LJ_ioPUT_DAC_ENABLE, 1, 0, 0);
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_PORT, 0, 0, 16);
```

… or with a single request to the following IOType created exactly for this purpose:

```
ePut (lngHandle, LJ_ioPIN_CONFIGURATION_RESET, 0, 0, 0);
```

2.6 AIN

The LabJack U3 has up to 16 analog inputs available on the flexible I/O lines (FIO0-FIO7 and EIO0-EIO7). Single-ended measurements can be taken of any line compared to ground, or differential measurements can be taken of any line to any other line.

Analog input resolution is 12-bits. The range of single-ended analog inputs is typically 0-2.4 volts, and the range of differential analog inputs is typically +/- 2.4 volts (pseudobipolar). For valid measurements, the voltage on every analog input pin, with respect to ground, must be within -0.3 to +3.6 volts.

The analog inputs have a QuickSample option where each conversion is done faster at the expense of increased noise. This is enabled by passing a nonzero value for put_config special channel `LJ_chAIN_RESOLUTION`. There is also a LongSettling option where additional settling time is added between the internal mulitplexer configuration and the analog to digital conversion. This is enabled by passing a nonzero value for put_config special channel `LJ_chAIN_SETTLING_TIME`.

Command/response (software timed) analog input reads typically take 0.6-4.0 ms depending on number of channels and communication configuration. Input streaming is not supported at this time.

2.6.1 Channel Numbers

The LabJack U3 has up to 16 external analog inputs, plus a few internal channels. The low-level functions specify a positive and negative channel for each analog input conversion. With the LabJackUD driver, the IOType `LJ_ioGET_AIN` is used for single-ended channels only, and thus the negative channel is set to 31. There is an additional IOType called `LJ_ioGET_AIN_DIFF` that allows the user to specify the positive and negative channel.

Positive Channel #	
0-7	AIN0-AIN7 (FIO0-FIO7)
8-15	AIN8-AIN15 (EIO0-EIO7)
30	Temp Sensor
31	Vreg

Table 2-2. Positive Channel Numbers

Channel 31 puts the internal Vreg (~3.3 volts) on the positive input of the ADC. See Section 2.6.4 for information about the internal temperature sensor.

Negative Channel #	
0-7	AIN0-AIN7 (FIO0-FIO7)
8-15	AIN8-AIN15 (EIO0-EIO7)
30	Vref
31	Single-Ended
32	Special 0-3.6 (UD Only)

Table 2-3. Negative Channel Numbers

If the negative channel is set to anything besides 31, the U3 does a differential conversion and returns a pseudobipolar value. If the negative channel is set to 31, the U3 does a single-ended conversion returns a unipolar value. Channel 30 puts the internal voltage reference Vref (~2.44 volts) on the negative input of the ADC.

Channel 32 is a special negative channel supported by the LabJack driver. When used, the driver will actually pass 30 as the negative channel to the U3, and when the result is returned the driver adds Vref to the value. This results is a full span on the positive channel of about 0 to 4.88 volts (versus ground), but since the voltage on any analog input cannot exceed 3.6 volts, only 75% of the converters range is used and the span is about 0 to 3.6 volts.

2.6.2 Converting Binary Readings to Voltages

Following are the nominal input voltage ranges for the analog inputs, assuming that DAC1 is not enabled.

	Max V	Min V
Single-Ended	2.44	0.0
Differential	2.44	-2.44
Special 0-3.6	3.6	0.0

Table 2-4. Nominal Analog Input Voltage Ranges (DAC1 Disabled)

Note that the minimum differential input voltage of -2.44 volts means that the positive channel can be as much as 2.44 volts less than the negative channel, not that a channel can measure 2.44 volts less than ground. The voltage of any analog input pin, compared to ground, must be in the range -0.3 to +3.6 volts.

The "Special 0-3.6" range is obtained by doing a differential measurement where the negative channel is set to the internal Vref (2.44 volts).

Although the binary readings have 12-bit resolution, they are returned justified as 16-bit values, so the approximate nominal conversion from binary to voltage is:

Volts(uncalibrated) = (Bits/65536)*Span (Single-Ended)

Volts(uncalibrated) = (Bits/65536)*Span – Span/2 (Differential)

Where span is the maximum voltage minus the minimum voltage from the table above. The actual nominal conversions are provided in the tables below, and should be used if the actual calibration constants are not read for some reason. Most applications will use the actual calibrations constants (Slope and Offset) stored in the internal flash.

Volts = (Slope * Bits) + Offset

Since the U3 uses multiplexed channels connected to a single analog-to-digital converter (ADC), all channels have the same calibration for a given configuration.

Table 2-4 shows where the various calibration values are stored in the Mem area. Generally when communication is initiated with the U3, three calls will be made to the ReadMem function to retrieve the first 3 blocks of memory. This information can then be used to convert all analog input readings to voltages. The high level Windows DLL (LabJackUD) does this automatically.

Block #	Starting Byte		Nominal Value	
0	0	AIN SE Slope	3.7231E-05	volts/bit
0	8	AIN SE Offset	0.0000E+00	volts
0	16	AIN Diff Slope	7.4463E-05	volts/bit
0	24	AIN Diff Offset	-2.4400E+00	volts
1	0	DAC0 Slope	5.1717E+01	bits/volt
1	8	DAC0 Offset	0.0000E+00	bits
1	16	DAC1 Slope	5.1717E+01	bits/volt
1	24	DAC1 Offset	0.0000E+00	bits
2	0	Temp Slope	1.3021E-02	degK/bit
2	8	Vref @Cal	2.4400E+00	volts
2	16	Vref*1.5 @Cal	3.6600E+00	volts
2	24	Vreg @Cal	3.3000E+00	volts

Table 2-5. Calibration Constant Memory Locations

Each value in Table 2.4 is stored in 64-bit fixed point format (signed 32.32, little endian, 2's complement). Following are some examples of fixed point byte arrays and the associated floating point double values.

Fixed Point Byte Array (LSB, …, MSB)	Floating Point Double
{0,0,0,0,0,0,0,0}	0.0000000000
{0,0,0,0,1,0,0,0}	1.0000000000
{0,0,0,0,255,255,255,255}	-1.0000000000
{51,51,51,51,0,0,0,0}	0.2000000000
{205,204,204,204,255,255,255,255}	-0.2000000000
{73,20,5,0,0,0,0,0}	0.0000775030
{225,122,20,110,2,0,0,0}	2.4300000000
{102,102,102,38,42,1,0,0}	298.1500000000

Table 2-6. Fixed Point Conversion Examples

The previous information assumed that DAC1 is disabled. If DAC1 is enabled, then the internal reference (Vref = 2.44 volts) is not available for the ADC, and instead the internal regulator voltage (Vreg = 3.3 volts) is used as the reference for the ADC. Vreg is not as stable as Vref, but more stable than Vs (5 volt power supply). Following are the nominal input voltage ranges for the analog inputs, assuming that DAC1 is enabled.

	Max V	Min V
Single-Ended	3.3	0.0
Differential	3.3	-3.3
Special 0-3.6	3.6	0.0

Table 2-7. Nominal Analog Input Voltage Ranges (DAC1 Enabled)

Note that the minimum differential input voltage of -3.3 volts means that the positive channel can be as much as 3.3 volts less than the negative channel, not that a channel can measure 3.3 volts less than ground. The voltage of any analog input pin, compared to ground, must be in the range -0.3 to +3.6 volts.

The "Special 0-3.6" range is obtained by doing a differential measurement where the negative channel is set to the ADC reference, which in this case is Vreg.

2.6.3 Typical Analog Input Connections

A common question is "can this sensor/signal be measured with the U3". Unless the signal has a voltage (referred to U3 ground) beyond the limits in Appendix A, it can be connected without damaging the U3, but more thought is required to determine what is necessary to make useful measurements with the U3 or any measurement device.

Voltage (versus ground): The single-ended analog inputs on the U3 measure a voltage with respect to U3 ground. The differential inputs measure the voltage difference between two channels, but the voltage on each channel with respect to ground must still be within the common mode limits specified in Appendix A. When measuring parameters other than voltage, or voltages too big or too small for the U3, some sort of sensor or transducer is required to produce the proper voltage signal. Examples are a temperature sensor, amplifier, resistive voltage divider, or perhaps a combination of such things.

Impedance: When connecting the U3, or any measuring device, to a signal source, it must be considered what impact the measuring device will have on the signal. The main consideration is whether the currents going into or out of the U3 analog input will cause noticeable voltage errors due to the impedance of the source. To maintain consistent 12-bit results, it is recommended to keep the source impedance less than 10 kΩ.

Resolution (and Accuracy): Based on the measurement type and resolution of the U3, the resolution can be determined in terms of voltage or engineering units. Assume a 0-10 mV signal, corresponding to 0-100 degrees C, is connected to the U3. Samples are then acquired using the 0-2.44 volt single-ended input range, resulting in a voltage resolution of about 2.44/4096 = 596 μV. That means there will be about 17 discrete steps across the 10 mV span of the signal, and the temperature resolution is about 6 degrees C. If this experiment required a resolution of 1 degrees C, this configuration would not be sufficient. Accuracy will also need to be considered. Appendix A places some boundaries on expected accuracy, but an in-system calibration can generally be done to provide absolute accuracy down to the INL limits of the U3.

Speed: How fast does the signal need to be sampled? For instance, if the signal is a waveform, what information is needed: peak, average, RMS, shape, frequency, … ? Answers to these questions will help decide how many points are needed per waveform cycle, and thus what sampling rate is required. In the case of multiple channels, the scan rate is also considered. See Sections 3.1 and 3.2.

2.6.3.1 Signal from the LabJack

One example of measuring a signal from the U3 itself, is with an analog output. All I/O on the U3 share a common ground, so the voltage on an analog output (DAC) can be measured by simply connecting a single wire from that terminal to an AIN terminal (FIO/EIO). The analog output must be set to a voltage within the range of the analog input.

2.6.3.2 Unpowered isolated signal

An example of an unpowered isolated signal would be a photocell where the sensor leads are not shorted to any external voltages. Such a sensor typically has two leads, where the positive lead connects to an AIN terminal and the negative lead connects to a GND terminal.

2.6.3.3 Signal powered by the LabJack

A typical example of this type of signal is a 3-wire temperature sensor. The sensor has a power and ground wire that connect to Vs and GND on the LabJack, and then has a signal wire that simply connects to an AIN terminal.

Another variation is a 4-wire sensor where there are two signal wires (positive and negative) rather than one. If the negative signal is the same as power ground, or can be shorted ground, then the positive signal can be connected to AIN and a single-ended measurement can be made. A typical example where this does not work is a bridge type sensor, such as pressure sensor, providing the raw bridge output (and no amplifier). In this case the signal voltage is the difference between the positive and negative signal, and the negative signal cannot be shorted to ground. Such a signal could be measured using a differential input on the U3.

2.6.3.4 Signal powered externally

An example is a box with a wire coming out that is defined as a 0-5 volt analog signal and a second wire labeled as ground. The signal is known to have 0-5 volts compared to the ground wire, but the complication is what is the voltage of the box ground compared to the LabJack ground.

If the box is known to be electrically isolated from the LabJack, the box ground can simply be connected to LabJack GND. An example would be if the box was plastic, powered by an internal battery, and does not have any wires besides the signal and ground which are connected to AINx and GND on the LabJack.

If the box ground is known to be the same as the LabJack GND, then perhaps only the one signal wire needs to be connected to the LabJack, but it generally does not hurt to go ahead and connect the ground wire to LabJack GND with a 100 Ω resistor. You definitely do not want to connect the grounds without a resistor.

If little is known about the box ground, a DMM can be used to measure the voltage of box ground compared to LabJack GND. As long as an extreme voltage is not measured, it is generally OK to connect the box ground to LabJack GND, but it is a good idea to put in a 100 Ω series resistor to prevent large currents from flowing on the ground. Use a small wattage resistor (typically 1/8 or 1/4 watt) so that it blows if too much current does flow. The only current that should flow on the ground is the return of the analog input bias current, which is on the order of … for the U3.

The SGND terminal (on the same terminal block as SDA/SCL) can be used instead of GND for externally powered signals. A series resistor is not needed as SGND is fused to prevent overcurrent, but a resistor will eliminate confusion that can be caused if the fuse is tripping and resetting.

In general, if there is uncertainty, a good approach is to use a DMM to measure the voltage on each signal/ground wire without any connections to the UE9. If no large voltages are noted, connect the ground to UE9 SGND with a 100 Ω series resistor. Then again use the DMM to measure the voltage of each signal wire before connecting to the UE9.

Another good general rule is to use the minimum number of ground connections. For instance, if connecting 8 sensors powered by the same external supply, or otherwise referred to the same external ground, only a single ground connection is needed to the UE9. Perhaps the ground leads from the 8 sensors would be twisted together, and then a single wire would be connected to a 100 Ω resistor which is connected to UE9 ground.

2.6.3.5 Amplifying small signal voltages

The best results are generally obtained when a signal voltage spans the full analog input range of the LabJack. If the signal is too small it can be amplified before connecting to the LabJack. The following figure shows an operational amplifier (op-amp) configured as non-inverting:

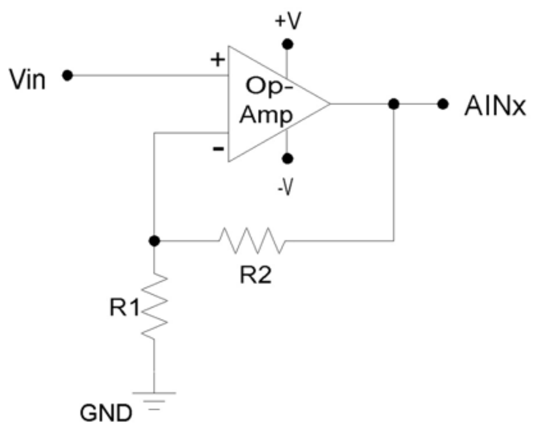


Figure 2-3. Non-Inverting Op-Amp Configuration

The gain of this configuration is:

Vout = Vin * (1 + (R2/R1))

100 kΩ is a typical value for R2. Note that if R2=0 (short-circuit) and R1=inf (not installed), a simple buffer with a gain equal to 1 is the result.

There are numerous criteria used to choose an op-amp from the thousands that are available. One of the main criteria is that the op-amp can handle the input and output signal range. Often, a single-supply rail-to-rail input and output (RIRO) is used as it can be powered from Vs and GND and pass signals within the range 0-Vs. The OPA344 from Texas Instruments (ti.com) is good for many 5 volt applications.

The op-amp is used to amplify (and buffer) a signal that is referred to the same ground as the LabJack (single-ended). If instead the signal is differential (i.e. there is a positive and negative signal both of which are different than ground), an instrumentation amplifier (in-amp) should be used. An in-amp converts a differential signal to single-ended, and generally has a simple method to set gain.

2.6.3.6 Signal voltages beyond 0-2.44 volts

The nominal maximum analog input voltage range for the U3 is 0-2.44 volts. The simplest way to handle higher unipolar voltages is with a resistive voltage divider. The following figure shows the basic voltage divider assuming that the source voltage (Vin) is referred to the same ground as the U3 (GND).

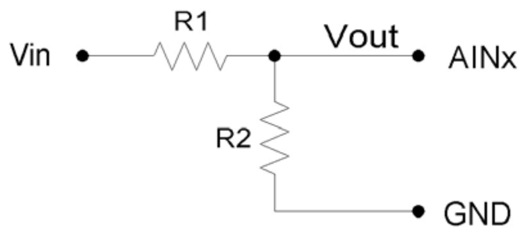


Figure 2-4. Voltage Divider Circuit

The attenuation of this circuit is determined by the equation:

Vout = Vin * (R2 / (R1+R2))

This divider is easily implemented by putting a resistor (R1) in series with the signal wire, and placing a second resistor (R2) from the AIN terminal to a GND terminal. To maintain specified analog input performance, R1 should not exceed 10 kΩ, so R1 can generally be fixed at 10 kΩ and R2 can be adjusted for the desired attenuation. For instance, R2 = 9.1 kΩ provides a divide by 2.1, so a ~0-5 volt input will be scaled to the input range of the U3.

The divide by 2 configuration where R1 = R2 = 10 kΩ, presents a 20 kΩ load to the source, meaning that a 5 volt signal will have to be able to source/sink up to +250 µA. Some signal sources might require a load with higher resistance, in which case a buffer should be used. The following figure shows a resistive voltage divider followed by an op-amp configured as non-inverting unity-gain (i.e. a buffer).

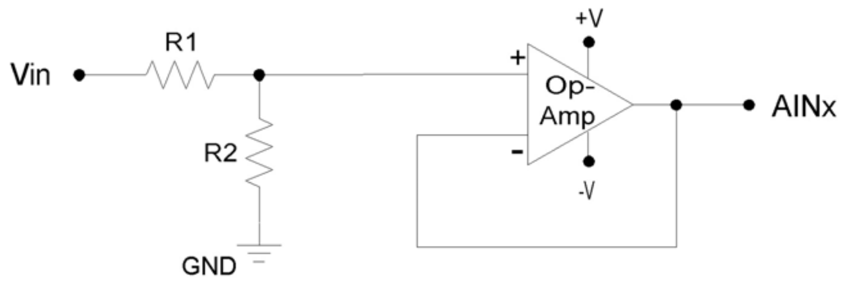


Figure 2-5. Buffered Voltage Divider Circuit

The op-amp is chosen to have low input bias currents so that large resistors can be used in the voltage divider. For 0-5 volt applications, where the amp will be powered from Vs and GND, a good choice would be the OPA344 from Texas Instruments (ti.com). The OPA344 has a very small bias current that changes little across the entire voltage range. Note that when powering the amp from Vs and GND, the input and output to the op-amp is limited to that range, so if Vs is 4.8 volts your signal range will be 0-4.8 volts.

2.6.3.7 Measuring current (including 4-20 mA) with a resistive shunt

The following figure shows a typical method to measure the current through a load, or to measure the 4-20 mA signal produced by a 2-wire current loop sensor. The current shunt shown in the figure is simply a resistor.

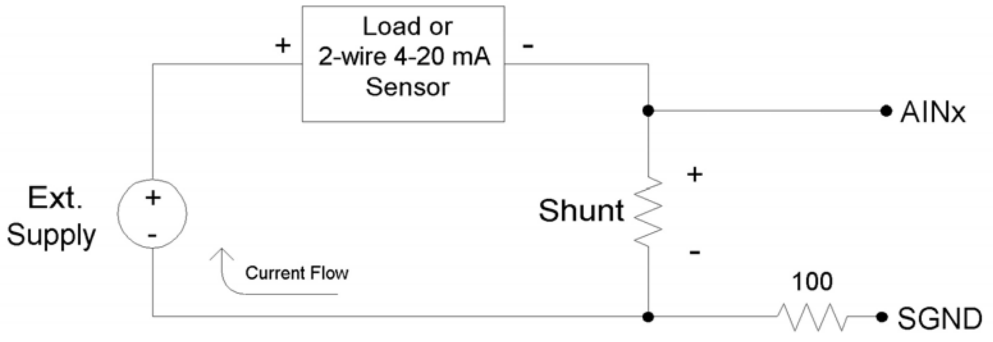


Figure 2-6. Current Measurement With Arbitrary Load or 2-Wire 4-20 mA Sensor

When measuring a 4-20 mA signal, a typical value for the shunt would be 120 Ω. This results in a 0.48 to 2.40 volt signal corresponding to 4-20 mA. The external supply must provide enough voltage for the sensor and the shunt, so if the sensor requires 5 volts the supply must provide at least 7.4 volts.

For applications besides 4-20 mA, the shunt is chosen based on the maximum current and how much voltage drop can be tolerated across the shunt. For instance, if the maximum current is 1.0 amp, and 1.0 volts of drop is the most that can be tolerated without affecting the load, a 1.0 Ω resistor could be used. That equates to 1.0 watts, though, which would require a special high wattage resistor. A better solution would be to use a 0.1 Ω shunt, and then use an amplifier to increase the small voltage produced by that shunt. If the maximum current to measure is too high (e.g. 100 amps), it will be difficult to find a small enough resistor and a hall-effect sensor should be considered instead of a shunt.

The following figure shows typical connections for a 3-wire 4-20 mA sensor. A typical value for the shunt would be 120 Ω which results in 0.48 to 2.40 volts.

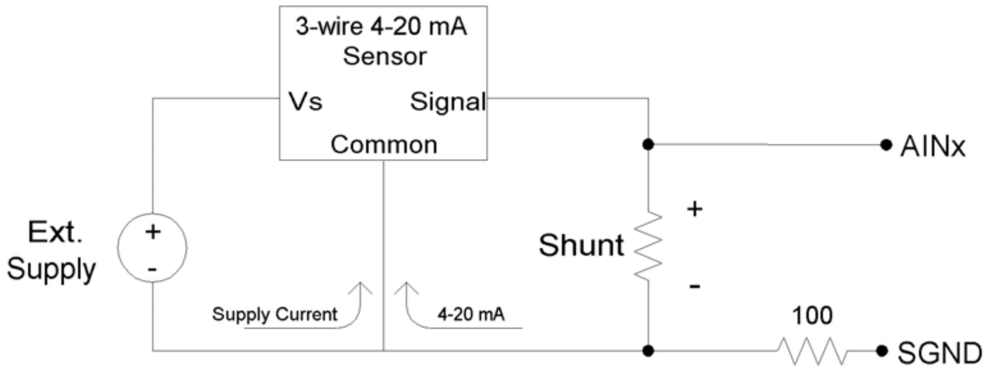


Figure 2-7. Current Measurement With 3-Wire 4-20 mA (Sourcing) Sensor

The sensor shown in Figure 2-7 is a sourcing type, where the signal sources the 4-20 mA current which is then sent through the shunt resistor and sunk into ground. Another type of 3-wire sensor is the sinking type, where the 4-20 mA current is sourced from the positive supply, sent through the shunt resistor, and then sunk into the signal wire. If sensor ground is connected to U3 ground, the sinking type of sensor presents a problems, as least one side of the resistor has a high common mode voltage (equal to the positive sensor supply). If the sensor is isolated, a possible solution is to connect the sensor signal or positive sensor supply to U3 ground (instead of sensor ground). This requires a good understanding of grounding and isolation in the system.

2.6.3.8 Floating/Unconnected Inputs

The reading from a floating (no external connection) analog input channel can be tough to predict and is likely to vary with sample timing and adjacent sampled channels. Keep in mind that a floating channel is not at 0 volts, but rather is at an undefined voltage. In order to see 0 volts, a 0 volt signal (such as GND) should be connected to the input.

Some data acquisition devices use a resistor, from the input to ground, to bias an unconnected input to read 0. This is often just for "cosmetic" reasons so that the input reads close to 0 with floating inputs, and a reason not to do that is that this resistor can degrade the input impedance of the analog input.

In a situation where it is desired that a floating channel read a particular voltage, say to detect a broken wire, a resistor can be placed from the AINx screw terminal to the desired voltage (GND, VS, DACx, ...). A 100 kΩ resistor should pull the analog input readings to within 50 mV of any desired voltage, but obviously degrades the input impedance to 100 kΩ. For the specific case of pulling a floating channel to 0 volts, a 1 MΩ resistor to GND can typically be used to provide analog input readings of less than 50 mV.

2.6.4 Internal Temperature Sensor

The U3 has an internal temperature sensor. Although this sensor measures the temperature inside the U3, which is warmer than ambient, it has been calibrated to read actual ambient temperature. For accurate measurements the temperature of the entire U3 must stabilize relative to the ambient temperature, which can take on the order of 1 hour. Best results will be obtained in still air in an environment with slowly changing ambient temperatures.

2.7 DAC

The LabJack U3 has 1 or 2 analog outputs (DAC0 and DAC1) that are available on the screw terminals. Each analog output can be set to a voltage between about 0.04 and 4.95 volts with 8-bits of resolution. The maximum output voltage is limited by the supply voltage to the U3.

The second analog output is only available in certain configurations. In particular, if the analog inputs are using the internal 2.4 volt reference (the most accurate option), then DAC1 outputs a fixed voltage of 1.5*Vref. If DAC1 is enabled, the analog inputs use Vreg (3.3 volts) as the ADC reference, which is not as stable as the internal 2.4 volt reference.

The DAC outputs are derived as a percentage of Vreg, and then amplified by 1.5, so any changes in Vreg will have a proportionate affect on the DAC outputs. Vreg is more stable than Vs (5 volt supply voltage), as it is the output from a 3.3 volt regulator.

The DACs are derived from PWM signals that are affected by the timer clock frequency (Section 2.x). The default timer clock frequency of the U3 is set to 24 MHz, as this results in the minimum DAC output noise. If the frequency is lowered, the DACs will have more noise, where the frequency of the noise is the timer clock frequency divided by 2^8.

The analog outputs have filters with a 3 dB cutoff around 16 Hz, limiting the frequency of output waveforms to less than that.

The analog output commands are sent as raw binary values (low level functions). For a desired output voltage, the binary value can be approximated as:

Bits(uncalibrated) = (Volts/4.95)*256

For a proper calculation, though, use the calibration values (Slope and Offset) stored in the internal flash on the Control processor (Table 2-X):

Bits = (Slope * Volts) + Offset

The analog outputs can withstand a continuous short-circuit to ground, even when set at maximum output.

Voltage should never be applied to the analog outputs, as they are voltage sources themselves. In the event that a voltage is accidentally applied to either analog output, they do have protection against transient events such as ESD (electrostatic discharge) and continuous overvoltage (or undervoltage) of a few volts.

2.7.1 Typical Analog Output Connections

2.7.1.1 High Current Output

The DACs on the U3 can output quite a bit of current, but have 50 Ω of source impedance that will cause voltage drop. To avoid this voltage drop, an op-amp can be used to buffer the output, such as the non-inverting configuration shown in Figure 2-2. A simple RC filter can be added between the DAC output and the amp input for further noise reduction. Note that the ability of the amp to source/sink current near the power rails must still be considered. A possible op-amp choice would be the TLV246x family (ti.com).

2.7.1.2 Different Output Ranges

The typical output range of the DACs is about 0.04 to 4.95 volts. For other unipolar ranges, an op-amp in the non-inverting configuration (Figure 2-3) can be used to provide the desired gain. For example, to increase the maximum output from 4.95 volts to 10.0 volts, a gain of 2.02 is required. If R2 (in Figure 2-3) is chosen as 100 kΩ, then an R1 of 97.6 kΩ is the closest 1% resistor that provides a gain greater than 2.02. The +V supply for the op-amp would have to be greater than 10 volts.

For bipolar output ranges, such as ±10 volts, a similar op-amp circuit can be used to provide gain and offset, but of course the op-amp must be powered with supplies greater than the desired output range (depending on the ability of the op-amp to drive it's outputs close to the power rails). If ±10, ±12, or ±15 volt supplies are available, consider using the LT1490A op-amp (linear.com), which can handle a supply span up to 44 volts.

A reference voltage is also required to provide the offset. In the following circuit, DAC1 is used to provide a reference voltage. The actual value of DAC1 can be adjusted such that the circuit output is 0 volts at the DAC0 mid-scale voltage, and the value of R1 can be adjusted to get the desired gain. A fixed reference (such as 2.5 volts) could also be used instead of DAC1.

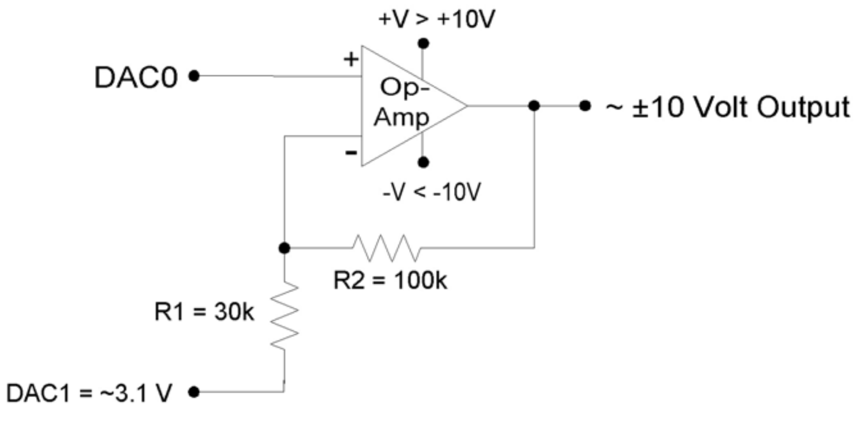


Figure 2-8. ±10 Volt DAC Output Circuit

A two-point calibration should be done to determine the exact input/output relationship of this circuit. Refer to application note SLOA097 from ti.com for further information about gain and offset design with op-amps.

2.8 Digital I/O

The LabJack U3 has up to 20 digital I/O channels. 16 are available from the flexible I/O lines, and 4 dedicated digital I/O (CIO0-CIO3) are available on the DB15 connector. Each digital line can be individually configured as input, output-high, or output-low. The digital I/O use 3.3 volt logic and are 5 volt tolerant.

The LabJackUD driver uses the following bit numbers to specify all the digital lines:

0-7	FIO0-FIO7
8-15	EIO0-EIO7
16-19	CIO0-CIO3

The 8 FIO lines appear on the built-in screw-terminals, while the 8 EIO and 4 CIO lines appear only on the DB15 connector. See the DB15 Section of this User's Guide for more information.

All the digital I/O include an internal series resistor that provides overvoltage/short-circuit protection. These series resistors also limit the ability of these lines to sink or source current. Refer to the specifications in Appendix A.

All digital I/O on the U3 have 3 possible states: input, output-high, or output-low. Each bit of I/O can be configured individually. When configured as an input, a bit has a ~100 kΩ pull-up resistor to 3.3 volts (all digital I/O are 5 volt tolerant). When configured as output-high, a bit is connected to the internal 3.3 volt supply (through a series resistor). When configured as output-low, a bit is connected to GND (through a series resistor).

The power-up condition of the digital I/O can be configured by the user. From the factory, all digital I/O are configured to power-up as inputs. Note that even if the power-up default for a line is changed to output-high or output-low, there is a delay of about xxx ms at power-up where all digital I/O are in the factory default condition.

The low-level Feedback function (Section 5.2.5) writes and reads all digital I/O. For information about using digital I/O under the Windows LabJackUD driver, see Section 4.3.5. See Section 3.1 for timing information.

Many function parameters contain specific bits within a single integer parameter to write/read specific information. In particular, most digital I/O parameters contain the information for each bit of I/O in one integer, where each bit of I/O corresponds to the same bit in the parameter (e.g. the direction of FIO0 is set in bit 0 of parameter FIODir). For instance, in the low-level function ControlU3, the parameter FIODirection is a single byte (8 bits) that writes/reads the power-up direction of each of the 8 FIO lines:

- if FIODirection is 0, all FIO lines are input,
- if FIODirection is 1 (2^0), FIO0 is output, FIO1-FIO7 are input,
- if FIODirection is 5 ($2^0 + 2^2$), FIO0 and FIO2 are output, all other FIO lines are input,
- if FIODirection is 255 ($2^0 + \ldots + 2^7$), FIO0-FIO7 are output.

2.8.1 Typical Digital I/O Connections

2.8.1.1 Input: Driven Signals

The most basic connection to a U3 digital input is a driven signal, often called push-pull. With a push-pull signal the source is typically providing a high voltage for logic high and zero volts for logic low. This signal is generally connected directly to the U3 digital input, considering the voltage specifications in Appendix A. If the signal is over 5 volts, it can still be connected with a series resistor. The digital inputs have protective devices that clamp the voltage at GND and VS, so the series resistor is used to limit the current through these protective devices. For instance, if a 24 volt signal is connected through a 10 kΩ series resistor, about 19 volts will be dropped across the resistor, resulting in a current of 1.9 mA, which is no problem for the U3.

The other possible consideration with the basic push-pull signal is the ground connection. If the signal is known to already have a common ground with the U3, then no additional ground connection is used. If the signal is known to not have a common ground with the U3, then the signal ground can simply be connected to U3 GND. If there is uncertainty about the relationship between signal ground and U3 ground (e.g. possible common ground through AC mains), then a ground connection with a 100 Ω series resistor is generally recommended (see Section 2.7.3.4).

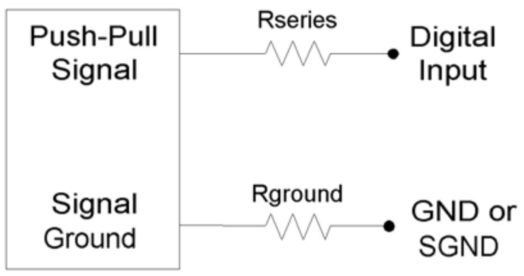


Figure 2-9. Driven Signal Connection To Digital Input

Figure 2-9 shows typical connections. Rground is typically 0-100 Ω. Rseries is typically 0 Ω (short-circuit) for 3.3/5 volt logic, or 10 kΩ for high-voltage logic. Note that an individual ground connection is often not needed for every signal. Any signals powered by the same external supply, or otherwise referred to the same external ground, should share a single ground connection to the U3 if possible.

When dealing with a new sensor, a push-pull signal is often incorrectly assumed when in fact the sensor provides an open-collector signal as described next.

2.8.1.2 Input: Open-Collector Signals

Open-collector (also called open-drain) is a very common type of digital signal. Rather than providing 5 volts and ground, like the push-pull signal, an open-collector signal provides ground and high-impedance. This type of signal can be thought of as a switch connected to ground. Since the U3 digital inputs have a 100 kΩ internal pull-up resistor, an open-collector signal can generally be connected directly to the input. When the signal is inactive, it is not driving any voltage and the pull-up resistor pulls the digital input to logic high. When the signal is active, it drives 0 volts which overpowers the pull-up and pulls the digital input to logic low. Sometimes, an external pull-up (e.g. 4.7 kΩ from Vs to digital input) will be installed to increase the strength and speed of the logic high condition.

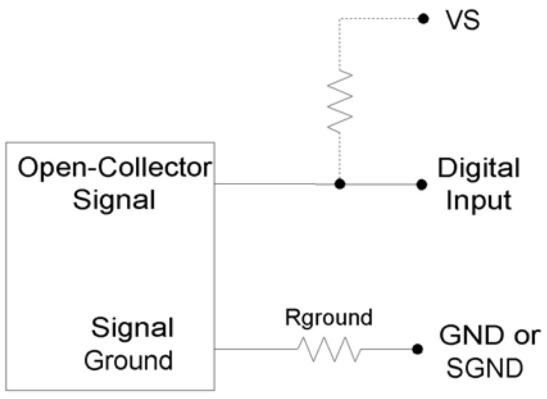


Figure 2-10. Driven Signal Connection To Digital Input

Figure 2-10 shows typical connections. Rground is typically 0-100 Ω, and the external pull-up resistor is generally not required. Note that an individual ground connection is often not needed for every signal. Any signals powered by the same external supply, or otherwise referred to the same external ground, should share a single ground connection to the U3 if possible.

2.8.1.3 Input: Mechanical Switch Closure

To detect whether a mechanical switch is open or closed, connect one side of the switch to U3 ground and the other side to a digital input. The behavior is very similar to the open-collector described above.

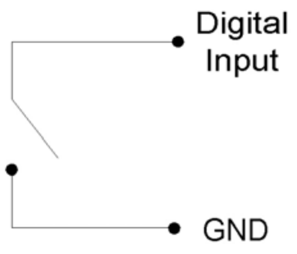


Figure 2-11. Basic Mechanical Switch Connection To Digital Input

When the switch is open, the internal 100 kΩ pull-up resistor will pull the digital input to about 3.3 volts (logic high). When the switch is closed, the ground connection will overpower the pull-up resistor and pull the digital input to 0 volts (logic low). Since the mechanical switch does not have any electrical connections, besides to the LabJack, it can safely be connected directly to GND, without using a series resistor or SGND.

When the mechanical switch is closed (and even perhaps when opened), it will bounce briefly and produce multiple electrical edges rather than a single high/low transition. For many basic digital input applications, this is not a problem as the software can simply poll the input a few times in succession to make sure the measured state is the steady state and not a bounce. For applications using timers or counters, however, this usually is a problem. The hardware counters, for instance, are very fast and will increment on all the bounces. Some solutions to this issue are:

- Software Debounce: If it is known that a real closure cannot occur more than once per some interval, then software can be used to limit the number of counts to that rate.

- Firmware Debounce: See section 2.10.1 for information about timer mode 6.
- Active Hardware Debounce: Integrated circuits are available to debounce switch signals. This is the most reliable hardware solution. See the MAX6816 (maxim-ic.com) or EDE2008 (elabinc.com).
- Passive Hardware Debounce: A combination of resistors and capacitors can be used to debounce a signal. This is not foolproof, but works fine in most applications.

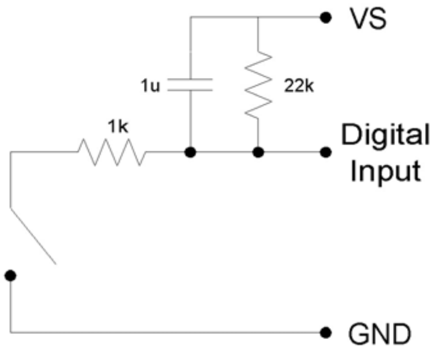


Figure 2-12. Passive Hardware Debounce

Figure 2-12 shows one possible configuration for passive hardware debounce. First, consider the case where the 1 kΩ resistor is replaced by a short circuit. When the switch closes it immediately charges the capacitor and the digital input sees logic low, but when the switch opens the capacitor slowly discharges through the 22 kΩ resistor with a time constant of 22 ms. By the time the capacitor has discharged enough for the digital input to see logic high, the mechanical bouncing is done. The main purpose of the 1 kΩ resistor is to limit the current surge when the switch is close. 1 kΩ limits the maximum current to about 5 mA, but better results might be obtained with smaller resistor values.

2.8.1.4 Output: Controlling Relays

All the digital I/O lines have series resistance that restricts the amount of current they can sink or source. Many SSRs can be controlled directly by the EIO/CIO lines (not the FIO), but the resulting control voltage levels might be marginal when comparing to the SSR specifications. To control higher currents with any of the digital I/O, some sort of buffer is used. The buffer could be a discrete transistor (e.g. 2N2222), a specific chip (e.g. ULN2003), or an op-amp.

Note that the U3 DACs can source enough current to control almost any SSR and even some mechanical relays, and thus can be a convenient way to control 1 or 2 relays.

The RB12 relay board is a useful accessory available from LabJack. This board connects to the DB15 connector on the U3 and accepts up to 12 industry standard I/O modules (designed for Opto22 G4 modules and similar).

2.9 Timers/Counters

The U3 has 2 timers (Timer0-Timer1) and 2 counters (Counter0-Counter1). When any of these timers or counters are enabled, they take over an FIO/EIO line in sequence (Timer0, Timer1, Counter0, then Counter1), starting with FIO0+TimerCounterPinOffset. Some examples:

<u>1 Timer enabled, Counter0 disabled, Counter1 disabled, and TimerCounterPinOffset=0:</u>
FIO0=Timer0

<u>1 Timer enabled, Counter0 disabled, Counter1 enabled, and TimerCounterPinOffset=2:</u>

FIO2=Timer0
FIO3=Counter1

<u>2 Timers enabled, Counter0 enabled, Counter1 enabled, and TimerCounterPinOffset=8:</u>
EIO0=Timer0
EIO1=Timer1
EIO2=Counter0
EIO3=Counter1

Timers and counters can appear on various pins, but other I/O lines never move. For example, Timer1 can appear anywhere from FIO0 to EIO1, depending on TimerCounterPinOffset and whether Timer0 is enabled. On the other hand, FIO5 (for example), is always on the screw terminal labeled FIO5, and AIN5 (if enabled) is always on that same screw terminal.

Note that Counter0 is not available with certain timer clock base frequencies. In such a case, it does not use an external FIO/EIO pin. Counter0 does not use an external An error will result if an attempt is made to enable Counter0 when one of these frequencies are configured. Similarly, an error will result if an attempt is made to configure one of these frequencies when Counter0 is enabled.

See Section 2.9.1 for information about signal connections.

Each counter (Counter0 or Counter1) consists of a 32-bit register that accumulates the number of falling edges detected on the external pin. If a counter is reset and read in the same function call, the read returns the value just before the reset.

The timers (Timer0-Timer1) have various modes available:

<u>Timer Modes</u>	
0	16-bit PWM output
1	8-bit PWM output
2	Period input (32-bit, rising edges)
3	Period input (32-bit, falling edges)
4	Duty cycle input
5	Firmware counter input
6	Firmware counter input (with debounce)
7	Frequency output
8	Quadrature input
9	Timer stop input (odd timers only)
10	System timer low read **(default mode)**
11	System timer high read
12	Period input (16-bit, rising edges)
13	Period input (16-bit, falling edges)

Both timers use the same timer clock. There are 7 choices for the timer base clock:

TimerBaseClock	
0	2 MHz
1	6 MHz
2	24 MHz (Default)
3	500 kHz /Divisor
4	2 MHz /Divisor
5	6 MHz /Divisor
6	24 MHz /Divisor

The first 3 clocks have a fixed frequency, and are not affected by TimerClockDivisor. The frequency of the last 4 clocks can be further adjusted by TimerClockDivisor, but when using these clocks Counter0 is not available. When Counter0 is not available, it does not use an external FIO/EIO pin.

Note that the DACs (Section 2.x) are derived from PWM signals that are affected by the timer clock frequency. The default timer clock frequency of the U3 is set to 24 MHz, as this results in the minimum DAC output noise. If the frequency is lowered, the DACs will have more noise, where the frequency of the noise is the timer clock frequency divided by 2^8.

2.9.1 Timer Mode Descriptions

16-Bit PWM Output (Mode 0)

Outputs a pulse width modulated rectangular wave output. Value passed should be 0-65535, and determines what portion of the total time is spent low (out of 65536 total increments). That means the duty cycle can be varied from 100% (0 out of 65536 are low) to 0.0015% (65535 out of 65536 are low).

The overall frequency of the PWM output is the clock frequency specified by TimerClockBase/TimerClockDivisor divided by 2^{16}. The following table shows the range of available PWM frequencies based on timer clock settings.

		PWM16 Frequency		
TimerBaseClock		Divisor=1	Divisor=256	
0	2 MHz	30.52	N/A	2000000
1	6 MHz	91.55	N/A	6000000
2	24 MHz (Default)	366.21	N/A	24000000
3	500 kHz /Divisor	7.63	0.030	500000
4	2 MHz /Divisor	30.52	0.119	2000000
5	6 MHz /Divisor	91.55	0.358	6000000
6	24 MHz /Divisor	366.21	1.431	24000000

The same clock applies to all timers, so all 16-bit PWM channels will have the same frequency and will have their falling edges at the same time.

8-Bit PWM Output (Mode 1)

Outputs a pulse width modulated rectangular wave output. Value passed should be 0-65535, and determines what portion of the total time is spent low (out of 65536 total increments). The lower byte is actually ignored since this is 8-bit PWM. That means the duty cycle can be varied from 100% (0 out of 65536 are low) to 0.4% (65280 out of 65536 are low).

The overall frequency of the PWM output is the clock frequency specified by TimerClockBase/TimerClockDivisor divided by 2^8. The following table shows the range of available PWM frequencies based on timer clock settings.

		PWM8 Frequency		
TimerBaseClock		Divisor=1	Divisor=256	
0	2 MHz	7812.50	N/A	2000000
1	6 MHz	23437.50	N/A	6000000
2	24 MHz (Default)	93750.00	N/A	24000000
3	500 kHz /Divisor	1953.13	7.629	500000
4	2 MHz /Divisor	7812.50	30.518	2000000
5	6 MHz /Divisor	23437.50	91.553	6000000
6	24 MHz /Divisor	93750.00	366.211	24000000

The same clock applies to all timers, so all 8-bit PWM channels will have the same frequency and will have their falling edges at the same time.

Period Measurement (32-Bit, Modes 2 & 3)

Mode 2: On every rising edge seen by the external pin, this mode records the number of clock cycles (clock frequency determined by TimerClockBase/TimerClockDivisor) between this rising edge and the previous rising edge. The value is updated on every rising edge, so a read returns the time between the most recent pair of rising edges.

In this 32-bit mode, the Control processor must jump to an interrupt service routine to record the time, so small errors can occur if another interrupt is already in progress. The possible error sources are:

- Other edge interrupt timer modes (2/3/4/5/8/9/12/13). If an interrupt is already being handled due to an edge on another timer, delays of a couple microseconds per timer are possible. That means if five other edge detecting timers are enabled, it is possible (but not likely) to have about 10 microseconds of delay.
- If a stream is in progress, every sample is acquired in a high-priority interrupt. These interrupts could cause delays of up to 10 microseconds.
- The always active UE9 system timer causes an interrupt 11.4 times per second. If this interrupt happens to be in progress when the edge occurs, a delay of about 1 microsecond is possible. If the software watchdog is enabled, the system timer interrupt takes longer to execute and a delay of a few microseconds is possible.

Writing a value of zero to the timer performs a reset. After reset, a read of the timer value will return zero until a new edge is detected. If a timer is reset and read in the same function call, the read returns the value just before the reset.

Mode 3 is the same except that falling edges are used instead of rising edges.

Duty Cycle Measurement (Mode 4)

Records the high and low time of a signal on the external pin, which provides the duty cycle, pulse width, and period of the signal. Returns 4 bytes, where the first two bytes (least significant word or LSW) are a 16-bit value representing the number of clock ticks during the high signal, and the second two bytes (most significant word or MSW) are a 16-bit value representing the number of clock ticks during the low signal. The clock frequency is determined by TimerClockBase/TimerClockDivisor. The appropriate value is updated on every edge, so a read returns the most recent high/low times.

To select a clock frequency, consider the longest expected high or low time, and set the clock frequency such that the 16-bit registers will not overflow.

When using the LabJackUD driver the value returned is the entire 32-bit value. To determine the high and low time this value should be split into a high and low word. One way to do this is to do a modulus divide by 2^{16} to determine the LSW, and a normal divide by 2^{16} (keep the quotient and discard the remainder) to determine the MSW.

Writing a value of zero to the timer performs a reset. After reset, a read of the timer value will return zero until a new edge is detected. If a timer is reset and read in the same function call, the read returns the value just before the reset.

Firmware Counter Input (Mode 5)

On every rising edge seen by the external pin, this mode increments a 32-bit register. Unlike the pure hardware counters, these timer counters require that the firmware jump to an interrupt service routine on each edge.

Writing a value of zero to the timer performs a reset. After reset, a read of the timer value will return zero until a new edge is detected. If a timer is reset and read in the same function call, the read returns the value just before the reset.

Firmware Counter Input With Debounce (Mode 6)

Intended for frequencies less than 10 Hz, this mode adds a debounce feature to the firmware counter, which is particularly useful for signals from mechanical switches. On every applicable edge seen by the external pin, this mode increments a 32-bit register. Unlike the pure hardware counters, these timer counters require that the firmware jump to an interrupt service routine on each edge.

When configuring only (UpdateConfig=1), the low byte of the timer value is a number from 0-255 that specifies the debounce period in ~30 ms intervals. In the high byte of the timer value, bit 0 determines whether negative edges (bit 0 clear) or positive edges (bit 0 set) are counted.

Assume this mode is enabled with a value of 1, meaning that the debounce period is 30 ms and negative edges will be counted. When the input detects a negative edge, it increments the count by 1, and then waits 30 ms before re-arming the edge detector. Any negative edges within the 30 ms debounce period are ignored. This is good behavior for a normally-high signal where the switch closure causes a brief low signal (Figure 2-9). The debounce period can be set long enough so that bouncing on both the switch closure and switch open is ignored.

When only updating and not configuring, writing a value of zero to the timer performs a reset. After reset, a read of the timer value will return zero until a new edge is detected. If a timer is reset and read in the same function call, the read returns the value just before the reset.

Frequency Output (Mode 7)

Outputs a square wave at a frequency determined by TimerClockBase/TimerClockDivisor divided by 2*Timer#Value. The Value passed should be between 0-255, where 0 is a divisor of 256. By changing the clock configuration and timer value, a wide range of frequencies can be output, as shown in the following table:

Mode 7 Frequencies

TimerBaseClock		Divisor=1 Value=1	Divisor=1 Value=256	
0	2 MHz	1000000.00	3906.25	2000000
1	6 MHz	3000000.00	11718.75	6000000
2	24 MHz (Default)	12000000.00	46875.00	24000000

		Divisor=1 Value=1	Divisor=256 Value=256	
3	500 kHz /Divisor	250000.00	3.815	500000
4	2 MHz /Divisor	1000000.00	15.259	2000000
5	6 MHz /Divisor	3000000.00	45.776	6000000
6	24 MHz /Divisor	12000000.00	183.105	24000000

Quadrature Input (Mode 8)
Requires both timers, where Timer0 will be quadrature channel A, and Timer1 will be quadrature channel B. Timer#Value passed has no effect. The U3 does 4x quadrature counting, and returns the current count as a signed 32-bit integer (2's complement). The same current count is returned on both timer value parameters.

Writing a value of zero to either or both timers performs a reset of both. After reset, a read of either timer value will return zero until a new quadrature count is detected. If a timer is reset and read in the same function call, the read returns the value just before the reset.

Timer Stop Input (Mode 9)
This mode should only be assigned Timer1. On every falling edge seen by the external pin, this mode increments an 8-bit register. When that register matches the specified timer value (stop count value), Timer0 is stopped. The range for the stop count value is 1-65535. Generally, the signal applied to Timer1 is from Timer0, which is configured as output. One place where this might be useful is for stepper motors, allowing control over a certain number of steps.

Once this timer reaches the specified stop count value, and stops the adjacent timer, the timers must be reconfigured to restart the output.

When Timer0 is stopped, it is still enabled but just not outputting anything. Thus rather than returning to whatever previous digital I/O state it had, it goes to input (which has a 100 kΩ pull-up). That means the best results are obtained if Timer0 was initially configured as input (factory default), rather than output-high or output-low.

The read from this timer mode returns the number of edges counted, but does not increment past the stop count value.

System Timer Low/High Read (Modes 10 & 11)
The LabJack U3 has a free-running internal 64-bit system timer with a frequency of 2 Mz. Timer modes 10 & 11 return the lower or upper 32-bits of this timer. An FIO line is allocated for these modes like normal, even though they are internal readings and do not require any external connections.

Period Measurement (16-Bit, Modes 12 & 13)
Similar to the 32-bit edge-to-edge timing modes described above (modes 2 & 3), except that hardware capture registers are used to record the edge times. This limits the times to 16-bit

values, but is accurate to the resolution of the clock, and not subject to any errors due to firmware processing delays.

2.9.2 Timer Operation/Performance Notes

Note that the specified timer clock frequency is the same for all timers. That is, TimerClockBase and TimerClockDivisor are singular values that apply to all timers. Modes 0, 1, 2, 3, 4, 7, 12, and 13, all are affected by the clock frequency, and thus the simultaneous use of these modes has limited flexibility. This is often not an issue for modes 2 and 3 since they use 32-bit registers.

The output timer modes (0, 1, and 7) are handled totally by hardware. Once started, no processing resources are used and other U3 operations do not affect the output.

The edge-detecting timer input modes do require U3 processing resources, as an interrupt is required to handle each edge. Timer modes 2, 3, 5, 9, 12, and 13 must process every applicable edge (rising **or** falling). Timer modes 4 and 8 must process every edge (rising **and** falling). To avoid missing counts, keep the total number of processed edges (all timers) less than nn,000 per second. That means that in the case of a single timer, there should be no more than 1 edge per 10 µs. For multiple timers, all can process an edge simultaneously, but if for instance 6 timers get an edge at the same time, 60 µs should be allowed before any further edges are applied. If streaming is occurring at the same time, the maximum edge rate will be less (nn,000 per second), and since each edge requires processing time the sustainable stream rates can also be reduced.

2.10 SCL and SDA (or SCA)

Reserved for future use.

2.11 DB15

The DB15 connector brings out 12 additional digital I/O. It has the potential to be used as an expansion bus, where the 8 EIO are data lines and the 4 CIO are control lines.

In the Windows LabJackUD driver, the EIO are addressed as digital I/O bits 8 through 15, and the CIO are addressed as bits 16-19.

0-7	FIO0-FIO7
8-15	EIO0-EIO7
16-19	CIO0-CIO3

These 12 channels include an internal series resistor that provides overvoltage/short-circuit protection. These series resistors also limit the ability of these lines to sink or source current. Refer to the specifications in Appendix A.

All digital I/O on the U3 have 3 possible states: input, output-high, or output-low. Each bit of I/O can be configured individually. When configured as an input, a bit has a ~100 kΩ pull-up resistor to 3.3 volts. When configured as output-high, a bit is connected to the internal 3.3 volt supply (through a series resistor). When configured as output-low, a bit is connected to GND (through a series resistor).

DB15 Pinouts

1	Vs	9	CIO0
2	CIO1	10	CIO2
3	CIO3	11	GND
4	EIO0	12	EIO1
5	EIO2	13	EIO3
6	EIO4	14	EIO5
7	EIO6	15	EIO7
8	GND		

2.11.1 CB15 Terminal Board

The CB15 terminal board connects to the LabJack U3's DB15 connector. It provides convenient screw terminal access to the 12 digital I/O available on the DB15 connector. The CB15 is designed to connect directly to the LabJack, or can connect via a standard 15-line 1:1 male-female DB15 cable.

2.11.2 RB12 Relay Board

The RB12 provides a convenient interface for the U3 to industry standard digital I/O modules, allowing electricians, engineers, and other qualified individuals, to interface a LabJack with high voltages/currents. The RB12 relay board connects to the DB15 connector on the LabJack, using the 12 EIO/CIO lines to control up to 12 I/O modules. Output or input types of digital I/O modules can be used. The RB12 is designed to accept G4 series digital I/O modules from Opto22, and compatible modules from other manufacturers such as the G5 series from Grayhill. Output modules are available with voltage ratings up to 200 VDC or 280 VAC, and current ratings up to 3.5 amps.

3. Operation

3.1 Command/Response

Everything besides streaming is done in command/response mode, meaning that all communication is initiated by a command from the host which is followed by a response from the U3.

For everything besides pin configuration, the low-level Feedback function is the primary function used, as it writes and reads virtually all I/O on the U3. The Windows LabJackUD driver uses the Feedback function under-the-hood to handle most requests besides configuration and streaming.

The following tables show typical measured execution times for the Feedback function. The time varies primarily with the number of analog inputs requested, and is not noticeably affected by the number of digital I/O and DAC operations. These times were measured by calling the function 1000 times and dividing the total time by 1000, and thus include everything (Windows latency, communication time, U3 processing time, etc.). A "USB high-high" configuration means the U3 is connected to a high-speed USB2 hub which is then connected to a high-speed USB2 host. Even though the U3 is not a high-speed USB device, such a configuration does provide improved performance.

# AIN	USB high-high [milliseconds]	USB other [milliseconds]
0	0.6	4.0
1	0.7	4.0
4	1.6	4.0
8	2.6	4.0
16	5.0	TBD

Table 3-1. Typical Feedback Function Execution Times w/ QuickSample=TRUE (Preliminary)

The above times include a write to DAC1 and a read from all available digital I/O lines. As an example with the LabJackUD driver: If requests are added to read 16 bits of digital I/O, update DAC0, and read 4 analog inputs, the GoOne() function call can be expected to typically take about 1.6 ms to execute via a USB high-high connection with the QuickSample option enabled. The AddRequest() and GetResult() calls take relatively no time at all.

4. LabJackUD High-Level Driver

The low-level U3 functions are described in Section 5, but most Windows applications will use the LabJackUD driver instead.

The driver requires a PC running Windows 98, ME, 2000, or XP. It is recommended to install the software before making a USB connection to a LabJack.

The download version of the installer consists of a single executable. This installer places the driver (LabJackUD.dll) in the Windows System directory, along with a support DLL (LabJackUSB.dll). Generally this is c:\Windows\System\ on Windows 98/ME, and c:\Windows\System32\ on Windows 2000/XP.

Other files, including the header and Visual C library file, are installed to the LabJack drivers directory which defaults to c:\Program Files\LabJack\drivers\.

4.1 Overview

The general operation of the LabJackUD functions is as follows:

- Open a LabJack.
- Build a list of requests to perform.
- Execute the list.
- Read the result of each request.

For example, to write an analog output and read an analog input:

```
//Use the following line to open the first found LabJack U3
//over USB and get a handle to the device.
//The general form of the open function is:
//OpenLabJack (DeviceType, ConnectionType, Address, FirstFound, *Handle)

//Open the first found LabJack U3 over USB.
lngErrorcode = OpenLabJack (LJ_dtU3, LJ_ctUSB, "1", TRUE, &lngHandle);

//Request that DAC0 be set to 2.5 volts.
//The general form of the AddRequest function is:
//AddRequest (Handle, IOType, Channel, Value, x1, UserData)
lngErrorcode = AddRequest (lngHandle, LJ_ioPUT_DAC, 0, 2.50, 0, 0);

//Request a read from AIN3 (FIO3), assuming it has been enabled as
//an analog line.
lngErrorcode = AddRequest (lngHandle, LJ_ioGET_AIN, 3, 0, 0, 0);

//Execute the requests.
lngErrorcode = GoOne (lngHandle);

//Get the result of the DAC0 request just to check for an errorcode.
//The general form of the GetResult function is:
//GetResult (Handle, IOType, Channel, *Value)
lngErrorcode = GetResult (lngHandle, LJ_ioPUT_DAC, 0, 0);

//Get the AIN3 voltage.  We pass the address to dblValue and the
//voltage will be returned in that variable.
lngErrorcode = GetResult (lngHandle, LJ_ioGET_AIN, 3, &dblValue);
```

The AddRequest/Go/GetResult method is often the most efficient. As shown above, multiple requests can be executed with a single Go() or GoOne() call, and the driver might be able to optimize the requests into fewer low-level calls. The other option is to use the eGet or ePut functions which combine the AddRequest/Go/GetResult into one call. The above code would then look like (assuming the U3 is already open):

```
//Set DAC0 to 2.5 volts.
//The general form of the ePut function is:
//ePut (Handle, IOType, Channel, Value, x1)
lngErrorcode = ePut (lngHandle, LJ_ioPUT_DAC, 0, 2.50, 0);

//Read AIN3.
//The general form of the eGet function is:
//eGet (Handle, IOType, Channel, *Value, x1)
lngErrorcode = eGet (lngHandle, LJ_ioGET_AIN, 3, &dblValue, 0);
```

In the case of the U3, the first example using add/go/get handles both the DAC command and AIN read in a single low-level call, while in the second example using ePut/eGet two low-level commands are used. Examples in the following documentation will use both the add/go/get method and the e-function method, and they are generally interchangeable. See Section 4.3 for more pseudocode examples.

All the request and result functions, including the e-functions, always have 4 common parameters, and some of the functions have 2 extra parameters:

- **Handle –** This is an input to all request/result functions that tells the function what LabJack it is talking to. The handle is obtained from the OpenLabJack function.
- **IOType –** This is an input to all request/result functions that specifies what type of action is being done.
- **Channel –** This is an input to all request/result functions that generally specifies which channel of I/O is being written/read, although with the config IOTypes special constants are passed for channel to specify what is being configured.
- **Value –** This is an input or output to all request/result functions that is used to write or read the value for the item being operated on.
- **x1 –** This parameter is only used in some of the request/result functions, and is used when extra information is needed for certain IOTypes.
- **UserData –** This parameter is only used in some of the request/result functions, and is data that is simply passed along with the request, and returned unmodified by the result. Can be used to store any sort of information with the request, to allow a generic parser to determine what should be done when the results are received.

4.1.1 Function Flexibility

The driver is designed to be flexible so that it can work with various different LabJacks with different capabilities. It is also designed to work with different development platforms with different capabilities. For this reason, many of the functions are repeated with different forms of parameters, although their internal functionality remains mostly the same. In this documentation, a group of functions will often be referred to by their shortest name. For example, a reference to Add or AddRequest most likely refers to any of the three variations: AddRequest(), AddRequestS() or AddRequestSS().

In the sample code, alternate functions (S or SS versions) can generally be substituted as desired, changing the parameter types accordingly. All samples here are written in pseudo-C.

Functions with an “S” or “SS” appended are provided for programming languages that can’t include the LabJackUD.h file and therefore can’t use the constants included. It is generally poor programming form to hardcode numbers into function calls, if for no other reason than it is hard to read. Functions with a single “S” replace the IOType parameter with a const char * which is a string. A string can then be passed with the name of the desired constant. Functions with a double “SS” replace both the IOType and Channel with strings. OpenLabJackS replaces both DeviceType and ConnectionType with strings since both take constants.

For example:

In C, where the LabJackUD.h file can be included and the constants used directly:
```
AddRequest(Handle, LJ_ioGET_CONFIG, LJ_ioHARDWARE_VERSION,0,0,0);
```

The bad way (hard to read) when LabJackUD.h cannot be included:
```
AddRequest(Handle, 1001, 10, 0, 0, 0);
```

The better way when LabJackUD.h cannot be included, is to pass strings:
```
AddRequestSS(Handle, “LJ_ioGET_CONFIG”, “LJ_ioHARDWARE_VERSION”,0,0,0);
```

Continuing on this vein, the function StringToConstant() is useful for error handling routines, or with the GetFirst/Next functions which do not take strings. The StringToConstant() function takes a string and returns the numeric constant. So, for example:

```
LJ_ERROR err;
err = AddRequestSS(Handle, “LJ_ioGETCONFIG”, “LJ_ioHARDWARE_VERSION”,0,0,0);
if (err == StringToConstant(“LJE_INVALID_DEVICE_TYPE”))
  do some error handling..
```

Once again, this is much clearer than:

```
if (err == 2)
```

4.1.2 Multi-Threaded Operation

This driver is completely thread safe. With some very minor exceptions, all these functions can be called from multiple threads at the same time and the driver will keep everything straight. Because of this Add, Go, and Get must be called from the same thread for a particular set of requests/results. Internally the list of requests and results are split by thread. This allows multiple threads to be used to make requests without accidentally getting data from one thread into another. If requests are added, and then results return LJE_NO_DATA_AVAILABLE or a similar error, chances are the requests and results are in different threads.

The driver tracks which thread a request is made in by the thread ID. If a thread is killed and then a new one is created, it is possible for the new thread to have the same ID. Its not really a problem if Add is called first, but if Get is called on a new thread results could be returned from the thread that already ended.

As mentioned, the list of requests and results is kept on a thread-by-thread basis. Since the driver cannot tell when a thread has ended, the results are kept in memory for that thread regardless. This is not a problem in general as the driver will clean it all up when unloaded. When it can be a problem is in situations where threads are created and destroyed continuously. This will result in the slow consumption of memory as requests on old threads are left behind. Since each request only uses 44 bytes, and as mentioned the ID's will eventually get recycled, it will not be a huge memory loss. In general, even without this issue, it is strongly recommended to not create and destroy a lot of threads. It is terribly slow and inefficient. Use thread pools and other techniques to keep new thread creation to a minimum. That is what is done internally.

The one big exception to the thread safety of this driver is in the use of the Windows TerminateThread() function. As is warned in the MSDN documentation, using TerminateThread() will kill the thread without releasing any resources, and more importantly, releasing any synchronization objects. If TerminateThread() is used on a thread that is currently in the middle of a call to this driver, more than likely a synchronization object will be left open on the particular device and access to the device will be impossible until the application is restarted. On some devices, it can be worse. On devices that have interprocess synchronization, such as the U12, calling TerminateThread() may kill all access to the device through this driver no matter which process is using it and even if the application is restarted. Avoid using TerminateThread()! All device calls have a timeout, which defaults to 1 second, but can be changed. Make sure to wait at least as long as the timeout for the driver to finish.

4.2 Function Reference

The LabJack driver file is named LabJackUD.dll, and contains the functions described in this section.

Some parameters are common to many functions:

- **LJ_ERROR –** A LabJack specific numeric error code. 0 means no error. (long, signed 32-bit integer).
- **LJ_HANDLE –** This value is returned by OpenLabJack, and then passed on to other functions to identify the opened LabJack. (long, signed 32-bit integer).

To maintain compatibility with as many languages as possible, every attempt has been made to keep the parameter types very basic. Also, many functions have multiple prototypes. The declarations that follow, are written in C.

To help those unfamiliar with strings in C, these functions expect null terminated 8 bit ASCII strings. How this translates to a particular development environment is beyond the scope of this documentation. A const char * is a pointer to a string that won't be changed by the driver. Usually this means it can simply be a constant such as "this is a string". A char * is a pointer to a string that will be changed. Enough bytes must be preallocated to hold the possible strings that will be returned. Functions with char * in their declaration will have the required length of the buffer documented below.

Pointers must be initialized in general, although null (0) can be passed for unused or unneeded values. The pointers for GetStreamData and RawIn/RawOut requests are not optional. Arrays and char * type strings must be initialized to the proper size before passing to the DLL.

4.2.1 ListAll()

Returns all the devices found of a given DeviceType and ConnectionType. Currently only USB is supported.

ListAllS() is a special version where DeviceType and ConnectionType are strings rather than longs. This is useful for passing string constants in languages that cannot include the header file. The strings should contain the constant name as indicated in the header file (such as "LJ_dtUE9" and "LJ_ctUSB"). The declaration for the S version of open is the same as below except for (const char *pDeviceType, const char *pConnectionType, …).

Declaration:

```
LJ_ERROR _stdcall ListAll (  long DeviceType,
                             long ConnectionType,
                             long *pNumFound,
                             long *pSerialNumbers,
                             long *pIDs,
                             double *pAddresses)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **DeviceType –** The type of LabJack to search for. Constants are in the labjackud.h file.

- **ConnectionType –** Enter the constant for the type of connection to use in the search. Currently, only USB is supported for this function.
- **pSerialNumbers –** Must pass a pointer to a buffer with at least 128 elements.
- **pIDs –** Must pass a pointer to a buffer with at least 128 elements.
- **pAddresses –** Must pass a pointer to a buffer with at least 128 elements.

Outputs:

- **pNumFound –** Returns the number of devices found, and thus the number of valid elements in the return arrays.
- **pSerialNumbers –** Array contains serial numbers of any found devices.
- **pIDs –** Array contains local IDs of any found devices.
- **pAddresses –** Array contains IP addresses of any found devices. The function DoubleToStringAddress() is useful to convert these to string notation.

4.2.2 OpenLabJack()

Call OpenLabJack() before communicating with a device. This function can be called multiple times, however, once a LabJack is open, it remains open until your application ends (or the DLL is unloaded). If OpenLabJack is called repeatedly with the same parameters, thus requesting the same type of connection to the same LabJack, the driver will simply return the same LJ_HANDLE every time. Internally, nothing else happens. This includes when the device is reset, or disconnected. Once the device is reconnected, the driver will maintain the same handle. If an open call is made for USB, and then Ethernet, a different handle will be returned for each connection type and both connections will be open.

OpenLabJackS() is a special version of open where DeviceType and ConnectionType are strings rather than longs. This is useful for passing string constants in languages that cannot include the header file. The strings should contain the constant name as indicated in the header file (such as "LJ_dtUE9" and "LJ_ctUSB"). The declaration for the S version of open is the same as below except for (const char *pDeviceType, const char *pConnectionType, …).

Declaration:

```
LJ_ERROR _stdcall OpenLabJack ( long DeviceType,
                                long ConnectionType,
                                const char *pAddress,
                                long FirstFound,
                                LJ_HANDLE *pHandle)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **DeviceType –** The type of LabJack to open. Constants are in the labjackud.h file.
- **ConnectionType –** Enter the constant for the type of connection, USB or Ethernet.
- **pAddress –** For USB, pass the local ID of the desired LabJack. For Ethernet pass the IP address of the desired LabJack. If FirstFound is true, Address is ignored.
- **FirstFound –** If true, then the Address and ConnectionType parameters are ignored and the driver opens the first LabJack found with the specified DeviceType. Generally only recommended when a single LabJack is connected. Currently only supported with USB. If a USB device is not found, it will try Ethernet but with the given Address.

Outputs:

- **pHandle –** A pointer to a handle for a LabJack.

4.2.3 eGet() and ePut()

The eGet and ePut functions do AddRequest, Go, and GetResult in one step.

The eGet versions are designed for inputs or retrieving parameters as they take a pointer to a double where the result is placed, but can be used for outputs if pValue is preset to the desired value. This is also useful for things like StreamRead where a value is input and output (number of scans requested and number of scans returned).

The ePut versions are designed for outputs or setting configuration parameters and will not return anything except the error code.

eGetS() and ePutS() are special versions of these functions where IOType is a string rather than a long. This is useful for passing string constants in languages that cannot include the header file, and is generally used with all IOTypes except put/get config. The string should contain the constant name as indicated in the header file (such as "LJ_ioANALOG_INPUT"). The declarations for the S versions are the same as the normal versions except for (…, const char *pIOType, …).

eGetSS() and ePutSS() are special versions of these functions where IOType and Channel are strings rather than longs. This is useful for passing string constants in languages that cannot include the header file, and is generally only used with the put/get config IOTypes. The strings should contain the constant name as indicated in the header file (such as "LJ_ioPUT_CONFIG" and "LJ_chLOCALID"). The declaration for the SS versions are the same as the normal versions except for (…, const char *pIOType, const char *pChannel, …).

The declaration for ePut is the same as eGet except that Value is not a pointer (…, double Value, …), and thus is an input only.

Declaration:

```
LJ_ERROR _stdcall eGet (    LJ_HANDLE Handle,
                            long IOType,
                            long Channel,
                            double *pValue,
                            long x1)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **Handle –** Handle returned by OpenLabJack().
- **IOType –** The type of request. See …
- **Channel –** The channel number of the particular IOType.
- **pValue –** Pointer to Value sends and receives data.
- **x1 –** Optional parameter used by some IOTypes.

Outputs:

- **pValue –** Pointer to Value sends and receives data.

4.2.4 AddRequest()

Adds an item to the list of requests to be performed on the next call to Go() or GoOne().

When AddRequest() is called on a particular Handle, all previous data is erased and cannot be retrieved by any of the Get functions until a Go function is called again. This is on a device by device basis, so you can call AddRequest() with a different handle while a device is busy performing its I/O.

AddRequest() only clears the request and result lists on the device handle passed and only for the current thread. For example, if a request is added to each of two different devices, and then a new request is added to the first device but not the second, a call to Go() will cause the first device to execute the new request and the second device to execute the original request.

In general, the execution order of a list of requests in a single Go call is unpredictable, except that all configuration type requests are executed before acquisition and output type requests.

AddRequestS() is a special version of the Add function where IOType is a string rather than a long. This is useful for passing string constants in languages that cannot include the header file, and is generally used with all IOTypes except put/get config. The string should contain the constant name as indicated in the header file (such as "LJ_ioANALOG_INPUT"). The declaration for the S version of Add is the same as below except for (…, const char *pIOType, …).

AddRequestSS() is a special version of the Add function where IOType and Channel are strings rather than longs. This is useful for passing string constants in languages that cannot include the header file, and is generally only used with the put/get config IOTypes. The strings should contain the constant name as indicated in the header file (such as "LJ_ioPUT_CONFIG" and "LJ_chLOCALID"). The declaration for the SS version of Add is the same as below except for (…, const char *pIOType, const char *pChannel, …).

<u>Declaration:</u>

```
LJ_ERROR _stdcall AddRequest (  LJ_HANDLE Handle,
                                long IOType,
                                long Channel,
                                double Value,
                                long x1,
                                double UserData)
```

<u>Parameter Description:</u>

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **Handle –** Handle returned by OpenLabJack().
- **IOType –** The type of request. See …
- **Channel –** The channel number of the particular IOType.
- **Value –** Value passed for output channels.
- **x1 –** Optional parameter used by some IOTypes.
- **UserData –** Data that is simply passed along with the request, and returned unmodified by GetFirstResult() or GetNextResult(). Can be used to store any sort of information with the request, to allow a generic parser to determine what should be done when the results are received.

Outputs:

- **None**

4.2.5 Go()

After using AddRequest() to make an internal list of requests to perform, call Go() to actually perform the requests. This function causes all requests on all open LabJacks to be performed. After calling Go(), call GetResult() or similar to retrieve any returned data or errors.

Go() can be called repeatedly to repeat the current list of requests. Go() does not clear the list of requests. Rather, after a call to Go(), the first subsequent AddRequest() call to a particular device will clear the previous list of requests on that particular device only.

Note that for a single Go() or GoOne() call, the order of execution of the request list cannot be predicted. Since the driver does internal optimization, it is quite likely not the same as the order of AddRequest() function calls. One thing that is known, is that configuration settings like ranges, stream settings, and such, will be done before the actual acquisition or setting of outputs.

<u>Declaration:</u>
LJ_ERROR _stdcall Go()

<u>Parameter Description:</u>
Returns: LabJack errorcodes or 0 for no error.
Inputs:

- **None**

Outputs:

- **None**

4.2.6 GoOne()

After using AddRequest() to make an internal list of requests to perform, call GoOne() to actually perform the requests. This function causes all requests on one particular LabJack to be performed. After calling GoOne(), call GetResult() or similar to retrieve any returned data or errors.

GoOne() can be called repeatedly to repeat the current list of requests. GoOne() does not clear the list of requests. Rather, after a particular device has performed a GoOne(), the first subsequent AddRequest() call to that device will clear the previous list of requests on that particular device only.

Note that for a single Go() or GoOne() call, the order of execution of the request list cannot be predicted. Since the driver does internal optimization, it is quite likely not the same as the order of AddRequest() function calls. One thing that is known, is that configuration settings like ranges, stream settings, and such, will be done before the actual acquisition or setting of outputs.

<u>Declaration:</u>
LJ_ERROR _stdcall GoOne(LJ_HANDLE Handle)

<u>Parameter Description:</u>
Returns: LabJack errorcodes or 0 for no error.
Inputs:

- **Handle –** Handle returned by OpenLabJack().

Outputs:

- **None**

4.2.7 GetResult()

Calling either Go function creates a list of results that matches the list of requests. Use GetResult() to read the result and errorcode for a particular IOType and Channel. Normally this function is called for each associated AddRequest() item. Even if the request was an output, the errorcode should be evaluated.

None of the Get functions will clear results from the list. The first AddRequest() call subsequent to a Go call will clear the internal lists of requests and results for a particular device.

When processing raw in/out or stream data requests, the call to a Get function does not actually cause the data arrays to be filled. The arrays are filled during the Go call (if data is available), and the Get call is used to find out many elements were placed in the array.

GetResultS() is a special version of the Get function where IOType is a string rather than a long. This is useful for passing string constants in languages that cannot include the header file, and is generally used with all IOTypes except put/get config. The string should contain the constant name as indicated in the header file (such as “LJ_ioANALOG_INPUT”). The declaration for the S version of Get is the same as below except for (…, const char *pIOType, …).

GetResultSS() is a special version of the Get function where IOType and Channel are strings rather than longs. This is useful for passing string constants in languages that cannot include the header file, and is generally only used with the put/get config IOTypes. The strings should contain the constant name as indicated in the header file (such as “LJ_ioPUT_CONFIG” and “LJ_chLOCALID”). The declaration for the SS version of Get is the same as below except for (…, const char *pIOType, const char *pChannel, …).

It is acceptable to pass NULL (or 0) for any pointer that is not required.

Declaration:

```
LJ_ERROR _stdcall GetResult (      LJ_HANDLE Handle,
                                   long IOType,
                                   long Channel,
                                   double *pValue)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **Handle –** Handle returned by OpenLabJack().
- **IOType –** The type of request. See …
- **Channel –** The channel number of the particular IOType.

Outputs:

- **pValue –** A pointer to the result value.

4.2.8 GetFirstResult() and GetNextResult()

Calling either Go function creates a list of results that matches the list of requests. Use GetFirstResult() and GetNextResult() to step through the list of results in order. When either function returns LJE_NO_MORE_DATA_AVAILABLE, there are no more items in the list of results. Items can be read more than once by calling GetFirstResult() to move back to the beginning of the list.

UserData is provided for tracking information, or whatever else the user might need.

None of the Get functions clear results from the list. The first AddRequest() call subsequent to a Go call will clear the internal lists of requests and results for a particular device.

When processing raw in/out or stream data requests, the call to a Get function does not actually cause the data arrays to be filled. The arrays are filled during the Go call (if data is available), and the Get call is used to find out many elements were placed in the array.

It is acceptable to pass NULL (or 0) for any pointer that is not required.

The parameter lists are the same for the GetFirstResult() and GetNextResult() declarations.

Declaration:

```
LJ_ERROR _stdcall GetFirstResult ( LJ_HANDLE Handle,
                                   long *pIOType,
                                   long *pChannel,
                                   double *pValue,
                                   long *px1,
                                   double *pUserData)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **Handle –** Handle returned by OpenLabJack().

Outputs:

- **pIOType –** A pointer to the IOType of this item in the list.
- **pChannel –** A pointer to the channel number of this item in the list.
- **pValue –** A pointer to the result value.
- **px1 –** A pointer to the x1 parameter of this item in the list.
- **pUserData –** A pointer to data that is simply passed along with the request, and returned unmodified. Can be used to store any sort of information with the request, to allow a generic parser to determine what should be done when the results are received.

4.2.9 DoubleToStringAddress()

Some pseudo-channels of the config IOType pass IP address (and others) in a double. This function is used to convert the double into a string in normal decimal-dot or hex-dot notation.

Declaration:

```
LJ_ERROR _stdcall DoubleToStringAddress (    double Number,
                                             char *pString,
                                             long HexDot)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **Number –** Double precision number to be converted.

- **pString –** Must pass a buffer for the string of at least 24 bytes.
- **HexDot –** If not equal to zero, the string will be in hex-dot notation rather than decimal-dot.

Outputs:

- **pString –** A pointer to the string representation.

4.2.10 StringToDoubleAddress()

Some pseudo-channels of the config IOType pass IP address (and others) in a double. This function is used to convert a string in normal decimal-dot or hex-dot notation into a double.

Declaration:

```
LJ_ERROR _stdcall StringToDoubleAddress (        const char *pString,
                                                 double *pNumber,
                                                 long HexDot)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **pString –** A pointer to the string representation.
- **HexDot –** If not equal to zero, the passed string should be in hex-dot notation rather than decimal-dot.

Outputs:

- **pNumber –** A pointer to the double precision representation.

4.2.11 StringToConstant()

Converts the given string to the appropriate constant number. Used internally by the S functions, but could be useful to the end user when using the GetFirst/Next functions without the ability to include the header file. In this case a comparison could be done on the return values such as:

```
if (IOType == StringToConstant("LJ_ioANALOG_INPUT"))
```

This function returns LJ_INVALID_CONSTANT if the string is not recognized.

Declaration:

```
long _stdcall StringToConstant ( const char *pString)
```

Parameter Description:

Returns: Constant number of the passed string.

Inputs:

- **pString –** A pointer to the string representation of the constant.

Outputs:

- **None**

4.2.12 ErrorToString()

Outputs a string describing the given error code or an empty string if not found.

Declaration:

```
void _stdcall ErrorToString (   LJ_ERROR ErrorCode,
```

char *pString)

Parameter Description:
Returns: LabJack errorcodes or 0 for no error.
Inputs:

- **ErrorCode –** LabJack errorcode.
- **pString –** Must pass a buffer for the string of at least 256 bytes.

Outputs:

- ***pString –** A pointer to the string representation of the errorcode.

4.2.13 GetDriverVersion()

Returns the version number of this Windows LabJack driver.

Declaration:
double _stdcall GetDriverVersion();

Parameter Description:
Returns: Driver version.
Inputs:

- **None**

Outputs:

- **None**

4.2.14 ResetLabJack()

Sends a reset command to the LabJack hardware.

Resetting the LabJack does not invalidate the handle, thus the device does not have to be opened again after a reset, but a Go call is likely to fail for a couple seconds after until the LabJack is ready.

In a future driver release, this function will probably be given an additional parameter that determines the type of reset.

Declaration:
LJ_ERROR _stdcall ResetLabJack (LJ_HANDLE Handle);

Parameter Description:
Returns: LabJack errorcodes or 0 for no error.
Inputs:

- **Handle –** Handle returned by OpenLabJack().

Outputs:

- **None**

4.3 Example Pseudocode

The following pseudocode examples are simplified for clarity, and in particular no error checking is shown. The language used for the pseudocode is C.

4.3.1 Open

The initial step is to open the LabJack and get a handle that the driver uses for further interaction. The DeviceType for the U3 is:

```
LJ_dtUE9
```

There are two choices for ConnectionType for the U3:

```
LJ_ctUSB
LJ_ctETHERNET
```

Following is example pseudocode to open a U3 over USB:

```
//Open the first found LabJack U3 over USB.
OpenLabJack (LJ_dtU3, LJ_ctUSB, "1", TRUE, &lngHandle);
```

The reason for the quotes around the address (“1”), is because the address parameter is a string in the OpenLabJack function.

The ampersand (&) in front of lngHandle is a C notation that means we are passing the address of that variable, rather than the value of that variable. In the definition of the OpenLabJack function, the handle parameter is defined with an asterisk (*) in front, meaning that the function expects a pointer, i.e. an address.

In general, a function parameter is passed as a pointer (address) rather than a value, when the parameter might need to output something. The parameter value passed to a function in C cannot be modified in the function, but the parameter can be an address that points to a value that can be changed. Pointers are also used when passing arrays, as rather than actually passing the array, an address to the first element in the array is passed.

4.3.2 Configuration

One of the most important operations on the U3 is configuring the flexible I/O as digital or analog. The following 4 IOTypes are used for that:

```
LJ_ioPUT_ANALOG_ENABLE_BIT
LJ_ioGET_ANALOG_ENABLE_BIT
LJ_ioPUT_ANALOG_ENABLE_PORT      //x1 is number of bits.
LJ_ioGET_ANALOG_ENABLE_PORT      //x1 is number of bits.
```

When a request is done with one of the port IOTypes, the Channel parameter is used to specify the starting bit number, and the x1 parameter is used to specify the number of applicable bits. Following are some pseudocode examples:

```
//Configure FIO3 as an analog input.
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_BIT, 3, 1, 0);

//Configure FIO3 as digital I/O.
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_BIT, 3, 0, 0);

//Configure FIO0-FIO2 and EIO0-EIO7 as analog, all others as digital.  That
```

```
//means a starting channel of 0, a value of b1111111100000111 (=d65287), and
//all 16 bits will be updated.
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_PORT, 0, 65287, 16);

//Configure FIO2-FIO4 as analog, and FIO5-FIO6 as digital, without
//configuring any other bits.   That means a starting channel of 2,
//a value of b00111 (=d7), and 5 bits will be updated.
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_PORT, 2, 7, 5);
```

Because of the pin configuration interaction between digital I/O, analog inputs, and timers/counters, many software applications will need to initialize the flexible I/O to a known pin configuration. One way to do this is with the following pseudocode:

```
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 0, 0);
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 0, 0);
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tc24MHZ, 0);
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 0, 0);
ePut (lngHandle, LJ_ioPUT_COUNTER_ENABLE, 0, 0, 0);
ePut (lngHandle, LJ_ioPUT_COUNTER_ENABLE, 1, 0, 0);
ePut (lngHandle, LJ_ioPUT_DAC_ENABLE, 1, 0, 0);
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_PORT, 0, 0, 16);
```

This disables all timers and counters, sets the timer/counter pin offset to 0, sets the timer clock base to 24 MHz (no divisor), sets the timer clock divisor to 0, disables DAC1, and sets all flexible I/O to digital. A simpler option is using the following IOType created exactly for this purpose, which does the same thing as the 8 function calls above:

```
ePut (lngHandle, LJ_ioPIN_CONFIGURATION_RESET, 0, 0, 0);
```

There are two IOTypes used to write or read general U3 configuration parameters:

```
LJ_ioPUT_CONFIG
LJ_ioGET_CONFIG
```

The following constants are then used in the channel parameter of the config function call to specify what is being written or read:

```
LJ_chLOCALID
LJ_chHARDWARE_VERSION
LJ_chSERIAL_NUMBER
LJ_chFIRMWARE_VERSION
LJ_chBOOTLOADER_VERSION
LJ_chPRODUCTID
LJ_chLED_STATE
```

Following is example pseudocode to write and read the local ID:

```
//Set the local ID to 4.
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chLOCALID, 4, 0);

//Read the local ID.
eGet (lngHandle, LJ_ioGET_CONFIG, LJ_chLOCALID, &dblValue, 0);
```

4.3.3 Analog Inputs

The IOTypes to retrieve a command/response analog input reading are:

```
LJ_ioGET_AIN        //Single-ended.  Negative channel is fixed as 31.
LJ_ioGET_AIN_DIFF   //Specify negative channel in x1.
```

The following are special channels, used with the get/put config IOTypes, to configure parameters that applies to all analog inputs:

```
LJ_chAIN_RESOLUTION        //QuickSample enabled if TRUE.
LJ_chAIN_SETTLING_TIME     //LongSettling enabled if TRUE.
LJ_chAIN_BINARY
```

Following is example pseudocode to read analog inputs:

```
//Execute the pin_configuration_reset IOType so that all
//pin assignments are in the factory default condition.
//The ePut function is used, which combines the add/go/get.
ePut (lngHandle, LJ_ioPIN_CONFIGURATION_RESET, 0, 0, 0);

//Configure FIO1, FIO2, and FIO6 as analog, all others as
//digital (see Section 4.3.2).
//The ePut function is used, which combines the add/go/get.
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_PORT, 0, 70, 16);


//Now, an add/go/get block to execute multiple requests.

//Request a single-ended read from AIN2.
AddRequest (lngHandle, LJ_ioGET_AIN, 2, 0, 0, 0);

//Request a differential read of AIN1-AIN6.
AddRequest (lngHandle, LJ_ioGET_AIN_DIFF, 1, 0, 6, 0);

//Request a differential read of AIN1-Vref.
AddRequest (lngHandle, LJ_ioGET_AIN_DIFF, 1, 0, 30, 0);

//Request a single-ended read of AIN1.
AddRequest (lngHandle, LJ_ioGET_AIN_DIFF, 1, 0, 31, 0);

//Request a read of AIN1 using the special 0-3.6 volt range.
AddRequest (lngHandle, LJ_ioGET_AIN_DIFF, 1, 0, 32, 0);

//Execute the requests.
GoOne (lngHandle);

//Since multiple requests were made with the same IOType
//and Channel, and only x1 was different, GetFirst/GetNext
//must be used to retrieve the results.  The simple
//GetResult function does not use the x1 parameter and
//thus there is no way to specify which result is desired.
//Rather than specifying the IOType and Channel of the
//result to be read, the GetFirst/GetNext functions retrieve
//the results in order.  Normally, GetFirst/GetNext are best
//used in a loop, but here they are simply called in succession.

//Retrieve AIN2 voltage.  GetFirstResult returns the IOType,
//Channel, Value, x1, and UserData from the first request.
//In this example we are just retrieving the results in order
//and Value is the only parameter we need.
GetFirstResult (lngHandle, 0, 0, &dblValue, 0, 0);

//Get the AIN1-AIN6 voltage.
GetNextResult (lngHandle, 0, 0, &dblValue, 0, 0);

//Get the AIN1-Vref voltage.
```

```
GetNextResult (lngHandle, 0, 0, &dblValue, 0, 0);

//Get the AIN1 voltage.
GetNextResult (lngHandle, 0, 0, &dblValue, 0, 0);

//Get the AIN1 voltage (special 0-3.6 volt range).
GetNextResult (lngHandle, 0, 0, &dblValue, 0, 0);
```

4.3.4 Analog Outputs

The IOType to set the voltage on an analog output is:

```
LJ_ioPUT_DAC
```

The following are IOTypes used to write/read the enable bit for DAC1:

```
LJ_ioPUT_DAC_ENABLE
LJ_ioGET_DAC_ENABLE
```

The following is a special channel, used with the get/put config IOTypes, to configure a parameter that applies to all DACs:

```
LJ_chDAC_BINARY
```

Following is example pseudocode to set DAC0 to 2.5 volts:

```
//Set DAC0 to 2.5 volts.
ePut (lngHandle, LJ_ioPUT_DAC, 0, 2.50, 0);
```

4.3.5 Digital I/O

There are eight IOTypes used to write or read digital I/O information:

```
LJ_ioGET_DIGITAL_BIT       //Also sets direction to input.
LJ_ioGET_DIGITAL_BIT_DIR
LJ_ioGET_DIGITAL_BIT_STATE
LJ_ioGET_DIGITAL_PORT      //Also sets directions to input.  x1 is number of bits.
LJ_ioGET_DIGITAL_PORT_DIR        //x1 is number of bits.
LJ_ioGET_DIGITAL_PORT_STATE      //x1 is number of bits.

LJ_ioPUT_DIGITAL_BIT       //Also sets direction to output.
LJ_ioPUT_DIGITAL_PORT      //Also sets directions to output.  x1 is number of bits.
```

When a request is done with one of the port IOTypes, the Channel parameter is used to specify the starting bit number, and the x1 parameter is used to specify the number of applicable bits. The bit numbers corresponding to different I/O are:

0-7	FIO0-FIO7
8-15	EIO0-EIO7
16-19	CIO0-CIO3

Note that the GetResult function does not have an x1 parameter. That means that if two (or more) port requests are added with the same IOType and Channel, but different x1, the result retrieved by GetResult would be undefined. The GetFirstResult/GetNextResult commands do have the x1 parameter, and thus can handle retrieving responses from multiple port requests with the same IOType and Channel.

Following is example pseudocode for various digital I/O operations:

```
//Execute the pin_configuration_reset IOType so that all
//pin assignments are in the factory default condition.
//The ePut function is used, which combines the add/go/get.
ePut (lngHandle, LJ_ioPIN_CONFIGURATION_RESET, 0, 0, 0);


//Now, an add/go/get block to execute multiple requests.

//Request a read from FIO2.
AddRequest (lngHandle, LJ_ioGET_DIGITAL_BIT, 2, 0, 0, 0);

//Request a read from FIO4-EIO5 (10-bits starting
//from digital channel #4).
AddRequest (lngHandle, LJ_ioGET_DIGITAL_PORT, 4, 0, 10, 0);

//Set FIO3 to output-high.
AddRequest (lngHandle, LJ_ioPUT_DIGITAL_BIT, 3, 1, 0, 0);

//Set EIO6-CIO2 (5-bits starting from digital channel #14)
//to b10100 (=d20).  That is EIO6=0, EIO7=0, CIO0=1,
//CIO1=0, and CIO2=1.
AddRequest (lngHandle, LJ_ioPUT_DIGITAL_PORT, 14, 20, 5, 0);

//Execute the requests.
GoOne (lngHandle);

//Get the FIO2 read.
GetResult (lngHandle, LJ_ioGET_DIGITAL_BIT, 2, &dblValue);

//Get the FIO4-EIO5 read.
GetResult (lngHandle, LJ_ioGET_DIGITAL_PORT, 4, &dblValue);
```

4.3.6 Timers & Counters

There are eight IOTypes used to write or read digital I/O information:

```
LJ_ioGET_COUNTER
LJ_ioPUT_COUNTER_ENABLE
LJ_ioGET_COUNTER_ENABLE
LJ_ioPUT_COUNTER_RESET

LJ_ioGET_TIMER
LJ_ioPUT_TIMER_VALUE
LJ_ioPUT_TIMER_MODE
LJ_ioGET_TIMER_MODE
```

In addition to specifying the channel number, the following mode constants are passed in the value parameter when doing a request with the timer mode IOType:

```
LJ_tmPWM16                 //16-bit PWM output
LJ_tmPWM8                  //8-bit PWM output
LJ_tmRISINGEDGES32         //Period input (32-bit, rising edges)
LJ_tmFALLINGEDGES32        //Period input (32-bit, falling edges)
LJ_tmDUTYCYCLE             //Duty cycle input
LJ_tmFIRMCOUNTER           //Firmware counter input
LJ_tmFIRMCOUNTERDEBOUNCE   //Firmware counter input (with debounce)
LJ_tmFREQOUT               //Frequency output
LJ_tmQUAD                  //Quadrature input
LJ_tmTIMERSTOP             //Timer stop input (odd timers only)
LJ_tmSYSTIMERLOW           //System timer low read (no FIO)
```

```
LJ_tmSYSTIMERHIGH          //System timer high read (no FIO)
LJ_tmRISINGEDGES16         //Period input (16-bit, rising edges)
LJ_tmFALLINGEDGES16        //Period input (16-bit, falling edges)
```

The following are special channels, used with the get/put config IOTypes, to configure a parameter that applies to all timers/counters:

```
LJ_chNUMBER_TIMERS_ENABLED
LJ_chTIMER_CLOCK_BASE
LJ_chTIMER_CLOCK_DIVISOR
LJ_chTIMER_COUNTER_PIN_OFFSET
```

With the clock base special channel above, the following constants are passed in the value parameter to select the frequency:

```
LJ_tc2MHZ           //2 MHz clock base
LJ_tc6MHZ           //6 MHz clock base
LJ_tc24MHZ          //24 MHz clock base
LJ_tc500KHZ_DIV     //500 kHz clock base w/ divisor (no Counter0)
LJ_tc2MHZ_DIV       //2 MHz clock base w/ divisor (no Counter0)
LJ_tc6MHZ_DIV       //6 MHz clock base w/ divisor (no Counter0)
LJ_tc24MHZ_DIV      //24 MHz clock base w/ divisor (no Counter0)
```

Following is example pseudocode for configuring various timers and a hardware counter:

```
//Execute the pin_configuration_reset IOType so that all
//pin assignments are in the factory default condition.
//The ePut function is used, which combines the add/go/get.
ePut (lngHandle, LJ_ioPIN_CONFIGURATION_RESET, 0, 0, 0);


//First, an add/go/get block to configure the timers and counters.

//Set the pin offset to 2, which causes the timers to start on FIO2.
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 2, 0, 0);

//Enable both timers.  They will use FIO2-FIO3
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 2, 0, 0);

//Make sure Counter0 is disabled.
AddRequest (lngHandle, LJ_ioPUT_COUNTER_ENABLE, 0, 0, 0, 0);

//Enable Counter1.  It will use the next available line, FIO4.
AddRequest (lngHandle, LJ_ioPUT_COUNTER_ENABLE, 1, 1, 0, 0);

//All output timers use the same timer clock, configured here. The
//base clock is set to 24MHZ_DIV, meaning that the clock divisor
//is supported and Counter0 is not available.
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tc24MHZ_DIV, 0, 0);

//Set the timer clock divisor to 24, creating a 1 MHz timer clock.
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 24, 0, 0);

//Configure Timer0 as 8-bit PWM.  It will have a frequency
//of 1M/256 = 3906.25 Hz.
AddRequest (lngHandle, LJ_ioPUT_TIMER_MODE, 0, LJ_tmPWM8, 0, 0);

//Initialize the 8-bit PWM with a 50% duty cycle.
AddRequest (lngHandle, LJ_ioPUT_TIMER_VALUE, 0, 32768, 0, 0);

//Configure Timer1 as duty cycle input.
AddRequest (lngHandle, LJ_ioPUT_TIMER_MODE, 1, LJ_tmDUTYCYCLE, 0, 0);

//Execute the requests.
```

```
GoOne (lngHandle);
```

The following pseudocode demonstrates reading input timers/counters and updating the values of output timers. The e-functions are used in the following pseudocode, but some applications might combine the following calls into a single add/go/get block so that a single low-level call is used.

```
//Change Timer0 PWM duty cycle to 25%.
ePut (lngHandle, LJ_ioPUT_TIMER_VALUE, 0, 49152, 0);

//Read duty-cycle from Timer1.
eGet (lngHandle, LJ_ioGET_TIMER, 1, &dblValue, 0);

//The duty cycle read returns a 32-bit value where the
//least significant word (LSW) represents the high time
//and the most significant word (MSW) represents the low
//time.  The times returned are the number of cycles of
//the timer clock.  In this case the timer clock was set
//to 1 MHz, so each cycle is 1 microsecond.
dblHighCycles = (double)(((unsigned long)dblValue) % (65536));
dblLowCycles = (double)(((unsigned long)dblValue) / (65536));
dblDutyCycle = 100 * dblHighCycles / (dblHighCycles + dblLowCycles));
dblHighTime = 0.000001 * dblHighCycles;
dblLowTime = 0.000001 * dblLowCycles;

//Read the count from Counter1.  This is an unsigned 32-bit value.
eGet (lngHandle, LJ_ioGET_COUNTER, 1, &dblValue, 0);
```

Following is pseudocode to reset the input timer and the counter:

```
//Reset the duty-cycle measurement (Timer1) to zero, by writing
//a value of zero.  The duty-cycle measurement is continuously
//updated, so a reset is normally not needed, but one reason
//to reset to zero is to detect whether there has been a new
//measurement or not.
ePut (lngHandle, LJ_ioPUT_TIMER_VALUE, 1, 0, 0);

//Reset Counter1 to zero.
ePut (lngHandle, LJ_ioPUT_COUNTER_RESET, 1, 1, 0);
```

Note that if a timer/counter is read and reset at the same time (in the same Add/Go/Get block), the read will return the value just before reset.

4.3.7 Stream Mode

Not supported at this time.

4.3.8 Raw Output/Input

There are two IOTypes used to write or read raw data. These can be used to make low-level function calls (Section 5) through the UD driver. The only time these generally might be used is to access some low-level device functionality not available in the UD driver.

```
LJ_ioRAW_OUT
LJ_ioRAW_IN
```

When using these IOTypes, channel # specifies the desired communication pipe. For the U3, 0 is the normal pipe while 1 is the streaming pipe. The number of bytes to write/read is specified in value (1-16384), and x1 is a pointer to a byte array for the data. When retrieving the result, the value returned is the number of bytes actually read/written.

Following is example pseudocode to write and read the low-level command ConfigTimerClock (Section 5.2.4).

```
writeArray[2] = {0x05,0xF8,0x02,0x0A,0x00,0x00,0x00,0x00,0x00,0x00};
numBytesToWrite = 10;
numBytesToRead = 10;

//Raw Out.  This command writes the bytes to the device.
eGet(lngHandle, LJ_ioRAW_OUT, 0, &numBytesToWrite, pwriteArray);

//Raw In.  This command reads the bytes from the device.
eGet(lngHandle, LJ_ioRAW_IN, 0, &numBytesToRead, preadArray);
```

4.3.9 Miscellaneous

The U3 has a buzzer that can be used to make noise. The buzzer has a resonant frequency of about 4 kHz where it is the loudest. The frequency of the signal sent to the buzzer is determined by toggling the signal every n iterations of the main U3 firmware loop. The following IOType is used to control the buzzer:

```
// Channel = 0 buzz for a count, Channel = 1 buzz continuous
// Value is the Period
// X1 is the toggle count when channel = 0
LJ_ioBUZZER
```

If Channel=1, the buzzer goes continuously until commanded again. If Channel=0, the buzzer is toggled the number of times specified in x1. Value specifies the number of firmware loops per toggle. Following is a pseudocode example.

```
//Buzz at about 4 kHz for about 1 second.
ePut (lngHandle, LJ_ioBUZZER, 0, 70, 4000);
```

4.4 Errorcodes

All functions return an LJ_ERROR errorcode as listed in the following table.

Errorcode	Name	Description
-2	LJE_UNABLE_TO_READ_CALDATA	Warning: Defaults used instead.
-1	LJE_DEVICE_NOT_CALIBRATED	Warning: Defaults used instead.
0	LJE_NOERROR	
1	LJE_TOO_MANY_INTERNAL_CHANNELS	Cannot read more than two internal channels with a single Go.
2	LJE_INVALID_CHANNEL_NUMBER	Channel that does not exist (e.g. DAC2 on a UE9), or data from stream is requested on a channel that is not in the scan list.
3	LJE_INVALID_RAW_INOUT_PARAMETER	
4	LJE_UNABLE_TO_START_STREAM	
5	LJE_UNABLE_TO_STOP_STREAM	
6	LJE_NOTHING_TO_STREAM	
7	LJE_UNABLE_TO_CONFIG_STREAM	
8	LJE_BUFFER_OVERRUN	
9	LJE_STREAM_NOT_RUNNING	
10	LJE_INVALID_PARAMETER	
11	LJE_INVALID_STREAM_FREQUENCY	
12	LJE_INVALID_AIN_RANGE	
13	LJE_STREAM_CHECKSUM_ERROR	
14	LJE_STREAM_COMMAND_ERROR	
15	LJE_STREAM_ORDER_ERROR	
1000	LJE_MIN_GROUP_ERROR	
1001	LJE_UNKNOWN_ERROR	Unrecognized error that is caught.
1002	LJE_INVALID_DEVICE_TYPE	
1003	LJE_INVALID_HANDLE	
1004	LJE_DEVICE_NOT_OPEN	AddRequest() called even though Open() failed.
1005	LJE_NO_DATA_AVAILABLE	GetResponse() called without calling a Go function, or a channel is passed that was not in the request list.
1006	LJE_NO_MORE_DATA_AVAILABLE	
1007	LJE_LABJACK_NOT_FOUND	LabJack not found at the given id or address.
1008	LJE_COMM_FAILURE	Unable to send or receive the correct number of bytes.
1009	LJE_CHECKSUM_ERROR	
1010	LJE_DEVICE_ALREADY_OPEN	
1011	LJE_COMM_TIMEOUT	
1012	LJE_USB_DRIVER_NOT_FOUND	

Table 4-1. Error Codes

Note: some errors are specific to a request. For example, LJE_INVALID_CHANNEL_NUMBER. If this error occurs, other requests are not affected. Other errors, such as Comm Failure will cause all pending requests for a particular Go() to fail with the same error. If you receive this type of error you can not assume the state of any of the requests.

5. Low-Level Function Reference

This section describes the low level functions of the U3. These are commands sent over USB directly to the processor on the U3.

The majority of users, particularly on Windows, will use our high-level LabJack UD function library rather than these low-level functions.

5.1 General Protocol

Following is a description of the general U3 low-level communication protocol. There are two types of commands:

Normal: 1 command word plus 0-7 data words.
Extended: 3 command words plus 0-125 data words.

Normal commands have a smaller packet size and can be faster in some situations. Extended commands provide more commands, better error detection, and a larger maximum data payload.

Normal command format:

Byte	
0	Checksum8: Includes bytes 1-15.
1	Command Byte: DCCCCWWW
	Bit 7: Destination bit:
	0 = Local,
	1 = Remote.
	Bits 6-3: Normal command number (0-14).
	Bits 2-0: Number of data words.
2-15	Data words.

Extended command format:

Byte	
0	Checksum8: Includes bytes 1-5.
1	Command Byte: D1111CCC
	Bit 7: Destination bit:
	0 = Local,
	1 = Remote.
	Bits 6-3: 1111 specifies that this is an extended command.
	Bits 2-0: Used with some commands.
2	Number of data words.
3	Extended command number.
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6-255	Data words.

Checksum calculations:

All checksums are a "1's complement checksum". We use an 8-bit and 16-bit, both of which are unsigned. Sum all applicable bytes in an accumulator, 1 at a time. Each time you add another byte, check for overflow (carry bit) and if true add one to the accumulator.

In a high-level language, do the following for the 8-bit normal command checksum:

-Get the subarray consisting of bytes 1 and up.
-Convert bytes to U16 and sum into a U16 accumulator.
-Divide by 2^8 and sum the quotient and remainder.
-Divide by 2^8 and sum the quotient and remainder.

In a high-level language, do the following for an extended command 16-bit checksum:

-Get the subarray consisting of bytes 6 and up.
-Convert bytes to U16 and sum into a U16 accumulator (can't overflow).

Then do the following for the 8-bit extended checksum:

-Get the subarray consisting of bytes 1 through 5.
-Convert bytes to U16 and sum into a U16 accumulator.
-Divide by 2^8 and sum the quotient and remainder.
-Divide by 2^8 and sum the quotient and remainder.

<u>Destination bit:</u>

This bit specifies whether the command is destined for the local or remote target. This bit is ignored on the U3.

<u>Multi-byte parameters:</u>

In the following function definitions there are various multi-byte parameters. The least significant byte of the parameter will always be found at the lowest byte number. For instance, bytes 10 through 13 of CommConfig are the IP address which is 4 bytes long. Byte 10 is the least significant byte (LSB), and byte 13 is the most significant byte (MSB).

<u>Masks:</u>

Some functions have mask parameters. The WriteMask found in some functions specifies which parameters are to be written. If a bit is 1, that parameter will be updated with the new passed value. If a bit is 0, the parameter is not changed and only a read is performed.

The AINMask found in some functions specifies which analog inputs are acquired. This is a 16-bit parameter where each bit corresponds to AIN0-AIN15. If a bit is 1, that channel will be acquired.

The digital I/O masks, such as FIOMask, specify that the passed value for direction and state are updated if a bit 1. If a bit of the mask is 0 only a read is performed on that bit of I/O.

5.2 Low-Level Functions

5.2.1 BadChecksum

If the processor detects a bad checksum in any command, the following 2-byte normal response will be sent and nothing further will be done.

Response:

Byte	
0	0xB8
1	0xB8

5.2.2 ConfigU3

Writes and reads various configuration settings. Although this function has many of the same parameters as other functions, most parameters in this case are affecting the power-up values, not the current values.

If WriteMask is nonzero, some or all default values are written to flash. The U3 flash has a rated endurance of at least 20000 writes, which is plenty for reasonable operation, but if this function is called in a high-speed loop with a nonzero WriteMask, the flash could eventually be damaged.

There is a hardware method to restore bytes 9-20 to the factory default value of 0x00. Power up the U3 with a short from FIO2<=>SCL, then remove the jumper and power cycle the device again.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x0A
3	0x08
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	WriteMask0 Bit 3: LocalID Bit 2: DAC Defaults Bit 1: Digital I/O Defaults Bit 0: Reserved
7	WriteMask1 (Reserved)
8	LocalID
9	TimerCounterConfig Bits 4-7: TimerCounterPinOffset Bit 3: Enable Counter1 Bit 2: Enable Counter0 Bits 0-1: Number of timers enabled
10	FIOAnalog
11	FIODirection
12	FIOState
13	EIOAnalog
14	EIODirection
15	EIOState
16	CIODirection
17	CIOState
18	DAC1Enable
19	DAC0
20	DAC1
21	0x00
22	0x00
23	0x00
24	0x00
25	0x00

Response:

Byte	
0	Checksum8
1	0xF8
2	0x10
3	0x08
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	Reserved
8	Reserved
9-10	FirmwareVersion
11-12	BootloaderVersion
13-14	HardwareVersion
15-18	SerialNumber
19-20	ProductID
21	LocalID
22	TimerCounterMask
23	FIOAnalog
24	FIODirection
25	FIOState
26	EIOAnalog
27	EIODirection
28	EIOState
29	CIODirection
30	CIOState
31	DAC1Enable
32	DAC0
33	DAC1
34	0x00
35	0x00
36	0x00
37	0x00

- WriteMask: Has bits that determine which, if any, of the parameters will be written to flash as the reset defaults. If a bit is 1, that parameter will be updated with the new passed value. If a bit is 0, the parameter is not changed and only a read is performed. Note that reads return reset defaults, not necessarily current values (except for LocalID). For instance, the value returned by FIODirection is the directions at reset, not necessarily the current directions.
- LocalID: If the WriteMask bit 0 is set, the values passed become the default values, meaning they are written to flash and used at reset. This is a user-configurable ID that can be used to identify a specific LabJack. The return value of this parameter is the current value and the power-up default value.
- TimerCounterConfig: If the WriteMask bit 1 is set, the value passed becomes the default value, meaning it is written to flash and used at reset. The return value of this parameter is a read of the power-up default. See Section 5.2.3.
- FIO/EIO/CIO: If the WriteMask bit 1 is set, the values passed become the default values, meaning they are written to flash and used at reset. Regardless of the mask bit, this function has no effect on the current settings. The return value of these parameters are a read of the power-up defaults.

- DAC: If the WriteMask bit 2 is set, the values passed become the default values, meaning they are written to flash and used at reset. Regardless of the mask bit, this function has no effect on the current settings. The return values of these parameters are a read of the power-up defaults.
- FirmwareVersion: Fixed parameter specifies the version number of the main firmware. A firmware upgrade will generally cause this parameter to change.
- BootloaderVersion: Fixed parameter specifies the version number of the bootloader.
- HardwareVersion: Fixed parameter specifies the version number of the hardware.
- SerialNumber: Fixed parameter that is unique for every LabJack.
- ProductID: (3) Fixed parameter identifies this LabJack as a U3.

5.2.3 ConfigIO

Writes and reads the current IO configuration.

Command:

Byte		
0	Checksum8	
1	0xF8	
2	0x03	
3	0x0B	
4	Checksum16 (LSB)	
5	Checksum16 (MSB)	
6	WriteMask	
		Bit 4: Reserved, Pass 0
		Bit 3: EIOAnalog
		Bit 2: FIOAnalog
		Bit 1: DAC1Enable
		Bit 0: TimerCounterConfig
7	Reserved	
8	TimerCounterConfig	
		Bits 4-7: TimerCounterPinOffset
		Bit 3: Enable Counter1
		Bit 2: Enable Counter0
		Bits 0-1: Number of timers enabled
9	DAC1Enable	
		Bit 1: Reserved, Pass 0
		Bit 0: Enable DAC1
10	FIOAnalog	
11	EIOAnalog	

Response:

Byte	
0	Checksum8
1	0xF8
2	0x03
3	0x0B
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	Reserved
8	TimerCounterConfig
9	DAC1Enable
10	FIOAnalog
11	EIOAnalog

- WriteMask: Has bits that determine which, if any, of the parameters will be written.
- TimerCounterConfig: Used to enable/disable timers and counters. Timers/counters will be assigned to IO pins starting with FIO0 plus TimerCounterPinOffset (which is a value from 0-8). Timer0 takes the first IO pin, then Timer1, Counter0, and Counter1. Whenever this function is called and timers are enabled, the timers are initialized to mode 10, so the desired timer mode must always be specified after every call to this

function. Note that Counter0 is not available when using a timer clock base that supports a timer clock divisor (TimerClockBase = 3-6).

- DAC1Enable: Bit 0 enables DAC1. When DAC1 is disabled, it outputs a constant voltage of 1.5 times the internal Vref (~2.44 volts). When DAC1 is enabled, the internal Vref is not available for the analog inputs and Vreg (~3.3 volts) is used as the AIN reference.
- FIOAnalog: Each bit determines whether that bit of FIO is analog input (=1) or digital I/O (=0).
- EIOAnalog: Each bit determines whether that bit of EIO is analog input (=1) or digital I/O (=0).

5.2.4 ConfigTimerClock

Writes and read the timer clock configuration.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x02
3	0x0A
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Reserved
7	Reserved
8	TimerClockConfig Bit 7: Configure the clock Bits 2-0: TimerClockBase b000: 2 MHz b001: 6 MHz b010: 24 MHz (Default) b011: 500 kHz /Divisor b100: 2 MHz /Divisor b101: 6 MHz /Divisor b110: 24 MHz /Divisor
9	TimerClockDivisor (0 = ÷256)

Response:

Byte	
0	Checksum8
1	0xF8
2	0x02
3	0x0A
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	Reserved
8	TimerClockConfig
9	TimerClockDivisor (0 = ÷256)

- TimerClockConfig: Bit 7 determines whether the new TimerClockBase and TimerClockDivisor are written, or if just a read is performed. Bits 0-2 specify the TimerClockBase. If TimerClockBase is 3-6, then Counter0 is not available.
- TimerClockDivisor: The base timer clock is divided by this value, or divided by 256 if this value is 0. Only applies if TimerClockBase is 3-6.

5.2.5 Feedback

A flexible function that handles all command/response functionality. One or more IOTypes are used to perform a single write/read or multiple writes/reads.

Note that the general protocol described in Section 4.1 defines byte 2 of an extended command as the number of data words, which is the number of words in a packet beyond the first 3 (a word is 2 bytes). Also note that the overall size of a packet must be an even number of bytes, so in this case an extra 0x00 is added to the end of the command and/or response if needed to accomplish this.

Since this command has a flexible size, byte 2 will vary. For instance, if a single IOType of FIOStateRead (d14) is passed, byte 2 would be equal to 1 for the command and 2 for the response. If a single IOType of LED (d9) is passed, an extra 0 must be added to the command to make the packet have an even number of bytes, and byte 2 would be equal to 2. The response would also need an extra 0 to be even, and byte 2 would be equal to 2.

Command:

Byte	
0	Checksum8
1	0xF8
2	0.5 + Number of Data Words (IOTypes and Data)
3	0x00
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Echo
7-63	IOTypes and Data

Response:

Byte	
0	Checksum8
1	0xF8
2	1.5 + Number of Data Words (If Errorcode = 0)
3	0x00
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	ErrorFrame
8	Echo
9-63	Data

- IOTypes & Data: One or more IOTypes can be passed in a single command, up to the maximum packet size. More info about the available IOTypes is below. In the outgoing command each IOType is passed and accompanied by 0 or more data bytes. In the incoming response, only data bytes are returned without the IOTypes.
- Echo: This byte is simply echoed back in the response. A host application might pass sequential numbers to ensure the responses are in order and associated with the proper command.

- ErrorFrame: If Errorcode is not zero, this parameter indicates which IOType caused the error. For instance, if the 3rd passed IOType caused the error, the ErrorFrame would be equal to 3. Also note that data is only returned for IOTypes before the one that caused the error, so if any IOType causes an error the overall function response will have less bytes than expected.

IOTypes for Feedback Command:

Name	IOType (dec)	WriteBytes	ReadBytes
AIN	1	3	2
LED	9	2	0
BitStateRead	10	2	1
BitStateWrite	11	2	0
BitDirRead	12	2	1
BitDirWrite	13	2	0
PortStateRead	26	1	3
PortStateWrite	27	7	0
PortDirRead	28	1	3
PortDirWrite	29	7	0
DAC0	34	2	0
DAC1	35	2	0
Timer0	42	4	4
Timer0Config	43	4	0
Timer1	44	4	4
Timer1Config	45	4	0
Counter0	54	2	4
Counter1	55	2	4
Buzzer	63	6	0

5.2.5.1 AIN: IOType=1

AIN, 3 Command Bytes:

0	IOType=1
1	Bits 4-0: Positive Channel
	Bit 6: LongSettling
	Bit 7: QuickSample
2	Negative Channel

2 Response Bytes:

0	AIN LSB
1	AIN MSB

This IOType returns a single analog input reading.

- Positive Channel: 0-15 for AIN0-AIN15, 30 for temp sensor, or 31 for Vreg. Note that AIN0-AIN7 appear on FIO0-FIO7, and AIN8-AIN15 appear on EIO0-EIO7.
- LongSettling: If this bit is set, additional settling time is added between the mulitplexer configuration and the analog to digital conversion.
- QuickSample: If this bit is set, a faster analog input conversion is done, at the expense of increased noise.
- Negative Channel: 0-15 for AIN0-AIN15, 30 for Vref, or 31 for single-ended. Note that AIN0-AIN7 appear on FIO0-FIO7, and AIN8-AIN15 appear on EIO0-EIO7.
- AIN LSB & MSB: Analog input reading is returned justified as a 16-bit value. Differential readings are signed, while single-ended readings are unsigned.

5.2.5.2 LED: IOType=9

LED, 2 Command Bytes:

0	IOType=9
1	State

0 Response Bytes:

This IOType simply turns the status LED on or off.

- State: 1=On, 0=Off.

5.2.5.3 BitStateRead: IOType=10

BitStateRead, 2 Command Bytes:

0	IOType=10
1	Bits 0-4: IO Number

1 Response Byte:

0	Bit 0: State

This IOType reads the state of a single bit of digital I/O. Only lines configured as digital (not analog) return valid readings.

- IO Number: 0-7=FIO, 8-15=EIO, or 16-19=CIO.
- State: 1=High, 0=Low.

5.2.5.4 BitStateWrite: IOType=11

BitStateWrite, 2 Command Bytes:

0	IOType=11
1	Bits 0-4: IO Number
	Bit 7: State

0 Response Bytes:

This IOType writes the state of a single bit of digital I/O. Note that the desired line must be configured as digital (not analog) and must be configured as output.

- IO Number: 0-7=FIO, 8-15=EIO, or 16-19=CIO.
- State: 1=High, 0=Low.

5.2.5.5 BitDirRead: IOType=12

BitDirRead, 2 Command Bytes:

0	IOType=12
1	Bits 0-4: IO Number

1 Response Byte:

0	Bit 0: Direction

This IOType reads the direction of a single bit of digital I/O. This is the digital direction only, and does not provide any information as to whether the line is configured as digital or analog.

- IO Number: 0-7=FIO, 8-15=EIO, or 16-19=CIO.
- Direction: 1=Output, 0=Input.

5.2.5.6 BitDirWrite: IOType=13

BitDirWrite, 2 Command Bytes:

0	IOType=13
1	Bits 0-4: IO Number
	Bit 7: Direction

0 Response Bytes:

This IOType writes the direction of a single bit of digital I/O.

- IO Number: 0-7=FIO, 8-15=EIO, or 16-19=CIO.
- Direction: 1=Output, 0=Input.

5.2.5.7 PortStateRead: IOType=26

PortStateRead, 1 Command Byte:

0	IOType=26

3 Response Bytes:

0-2	State

This IOType reads the state of all digital I/O, where 0-7=FIO, 8-15=EIO, and 16-19=CIO. Only lines configured as digital (not analog) return valid readings.

- State: Each bit of this value corresponds to the specified bit of I/O such that 1=High and 0=Low. If all are low, State=d0. If all 20 standard digital I/O are high, State=d1048575. If FIO0-FIO2 are high, EIO0-EIO2 are high, CIO0 are high, and all other I/O are low (b000000010000011100000111), State=d67335.

5.2.5.8 PortStateWrite: IOType=27

PortStateWrite, 7 Command Bytes:

0	IOType=27
1-3	WriteMask
4-6	State

0 Response Bytes:

This IOType writes the state of all digital I/O, where 0-7=FIO, 8-15=EIO, and 16-19=CIO. Note that the desired lines must be configured as digital (not analog) and must be configured as output.

- WriteMask: Each bit specifies whether to update the corresponding bit of I/O.
- State: Each bit of this value corresponds to the specified bit of I/O such that 1=High and 0=Low. To set all low, State=d0. To set all 20 standard digital I/O high, State=d1048575. To set FIO0-FIO2 high, EIO0-EIO2 high, CIO0 high, and all other I/O low (b000000010000011100000111), State=d67335.

5.2.5.9 PortDirRead: IOType=28

PortDirRead, 1 Command Byte:

0	IOType=28

3 Response Bytes:

0-2	Direction

This IOType reads the directions of all digital I/O, where 0-7=FIO, 8-15=EIO, and 16-19=CIO. These are the digital directions only, and do not provide any information as to whether the lines are configured as digital or analog.

- Direction: Each bit of this value corresponds to the specified bit of I/O such that 1=Output and 0=Input. If all are input, Direction=d0. If all 20 standard digital I/O are output, Direction=d1048575. If FIO0-FIO2 are output, EIO0-EIO2 are output, CIO0 are output, and all other I/O are input (b000000010000011100000111), Direction=d67335.

5.2.5.10 PortDirWrite: IOType=29

<u>**PortDirWrite**, 7 Command Bytes:</u>

0	IOType=29
1-3	WriteMask
4-6	Direction

<u>0 Response Bytes:</u>

This IOType writes the direction of all digital I/O, where 0-7=FIO, 8-15=EIO, and 16-19=CIO. Note that the desired lines must be configured as digital (not analog).

- WriteMask: Each bit specifies whether to update the corresponding bit of I/O.
- Direction: Each bit of this value corresponds to the specified bit of I/O such that 1=Output and 0=Input. To configure all as input, Direction=d0. For all 20 standard digital I/O as output, Direction=d1048575. To configure FIO0-FIO2 as output, EIO0-EIO2 as output, CIO0 as output, and all other I/O as input (b00000010000011100000111), Direction=d67335.

5.2.5.11 DAC#: IOType=34,35

<u>**DAC#**, 2 Command Bytes:</u>

0	IOType=34,35
1	Value

<u>0 Response Bytes:</u>

This IOType controls a single analog output.

- Value: 0=Minimum, 255=Maximum.

5.2.5.12 Timer#: IOType=42,44

Timer#, 4 Command Bytes:

0	IOType=42,44
1	Bit 0: UpdateReset
2	Value LSB
3	Value MSB

4 Response Bytes:

0	Timer LSB
1	Timer
2	Timer
3	Timer MSB

This IOType provides the ability to update/reset a given timer, and read the timer value.

- Value: These values are only updated if the UpdateReset bit is 1. The meaning of this parameter varies with the timer mode.
- Timer: Returns the value from the timer module. This is the value before reset (if reset was done).

5.2.5.13 Timer#Config: IOType=43,45

Timer#Config, 4 Command Bytes:

0	IOType=43,45
1	TimerMode
2	Value LSB
3	Value MSB

0 Response Bytes:

This IOType configures a particular timer.

- TimerMode: See Section 2.9 for more information about the available modes.
- Value: These values are only updated if the UpdateReset bit is 1. The meaning of this parameter varies with the timer mode.

5.2.5.14 Counter#: IOType=54,55

Counter#, 2 Command Bytes:

0	IOType=54,55
1	Bit 0: Reset

4 Response Bytes:

0	Counter LSB
1	Counter
2	Counter
3	Counter MSB

This IOType reads a hardware counter, and optionally can do a reset.

- Reset: These values are only updated if the UpdateReset bit is 1. The meaning of this parameter varies with the timer mode.
- Counter: Returns the current count from the counter if enabled. This is the value before reset (if reset was done).

5.2.5.15 Buzzer: IOType=63

Buzzer, 6 Command Bytes:

0	IOType=63
1	Bit 0: Continuous
2	Period LSB
3	Period MSB
4	Toggles LSB
5	Toggles MSB

0 Response Bytes:

This IOType is used to make the buzzer buzz.

- Continuous: If this bit is set, the buzzer will toggle continuously.
- Period: This value determines how many main firmware loops the processor will execute before toggling the buzzer voltage.
- Toggles: If Continuous is false, this value specifies how many times the buzzer will toggle.

5.2.6 ReadMem (ReadCal)

Reads 1 block (32 bytes) from the non-volatile user or calibration memory. Command number 0x2A accesses the user memory area which consists of 256 bytes (block numbers 0-7). Command number 0x2D accesses the calibration memory area which consists of 96 bytes (block numbers 0-2). Do not call this function while streaming.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x01
3	0x2A (0x2D)
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	0x00
7	BlockNum

Response:

Byte	
0	Checksum8
1	0xF8
2	0x11
3	0x2A (0x2D)
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	0x00
8-39	32 Bytes of Data

5.2.7 WriteMem (WriteCal)

Writes 1 block (32 bytes) to the non-volatile user or calibration memory. Command number 0x28 accesses the user memory area which consists of 256 bytes (block numbers 0-7). Command number 0x2B accesses the calibration memory area which consists of 96 bytes (block numbers 0-2). Memory must be erased before writing. Do not call this function while streaming.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x11
3	0x28 (0x2B)
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	0x00
7	BlockNum
8-39	32 Bytes of Data

Response:

Byte	
0	Checksum8
1	0xF8
2	0x01
3	0x28 (0x2B)
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	0x00

5.2.8 EraseMem (EraseCal)

The U3 uses flash memory that must be erased before writing. Command number 0x29 erases the entire user memory area. Command number 0x2C erases the entire calibration memory area. The EraseCal command has two extra constant bytes, to make it more difficult to call the function accidentally. Do not call this function while streaming.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x00 (0x01)
3	0x29 (0x2C)
4	Checksum16 (LSB)
5	Checksum16 (MSB)
(6)	(0x4C)
(7)	(0x6C)

Response:

Byte	
0	Checksum8
1	0xF8
2	0x01
3	0x29 (0x2C)
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	0x00

5.2.9 Reset

Causes a soft or hard reset. A soft reset consists of re-initializing most variables without re-enumeration. A hard reset is a reboot of the processor and does cause re-enumeration.

Command:

Byte	
0	Checksum8
1	0x99
2	ResetOptions Bit 1: Hard Reset Bit 0: Soft Reset
3	0x00

Response:

Byte	
0	Checksum8
1	0x99
2	0x00
3	Errorcode

5.2.10 StreamConfig (Preliminary)

Stream mode operates on a table of channels that are scanned at the specified scan rate. Before starting a stream, you need to call this function to configure the table and scan clock.

Command:

Byte	
0	Checksum8
1	0xF8
2	NumChannels + 3
3	0x11
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	NumChannels
7	SamplesPerPacket (1-16)
8	Reserved
9	ScanConfig Bit 7: Enable scan pulse output. Bit 6: Enable external scan trigger. Bit 3: Internal stream clock frequency. b00: 2 MHz b01: 24 MHz
10-11	Scan Interval (1-65535)
12	PChannel
13	NChannel

Repeat 12-13 for each channel

Response:

Byte	
0	Checksum8
1	0xF8
2	0x01
3	0x11
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	0x00

- NumChannels: This is the number of channels you will sample per scan (1-128).
- SamplesPerPacket: Specifies how many samples will be pulled out of the U3 FIFO buffer and returned per data read packet. For faster stream speeds, 16 samples per packet are required for data transfer efficiency. A small number of samples per packet would be desirable for low-latency data retrieval.
- ScanConfig: If you enable the scan pulse output, Counter1 will pulse low just before each scan (master mode). If you enable the external scan trigger, the UE9 scans the table each time it detects a falling edge on Counter1 (slave mode). Valid combinations for bits 6 and 7 are b00, b01, or b10. You cannot pass b11. Counter1 is automatically enabled and disabled by the stream functions. To provide the highest timing resolution, the scan clock is generally set to the highest setting possible.

- ScanInterval: (1-65535) This value divided by the clock frequency defined in the ScanConfig parameter, gives the interval (in seconds) between scans.
- PChannel/NChannel: For each channel, these two parameters specify the positive and negative voltage measurement point. PChannel is 0-7 for FIO0-FIO7, 8-15 for EIO0-EIO15, 30 for temp sensor, or 31 for Vreg. NChannel is 0-7 for FIO0-FIO7, 8-15 for EIO0-EIO15, 30 for Vref, or 31 for single-ended.

5.2.11 StreamStart (Preliminary)

Once the stream settings are configured, this function is called to start the stream.

Command:

Byte	
0	0xA8
1	0xA8

Response:

Byte	
0	Checksum8
1	0xA9
2	Errorcode
3	0x00

5.2.12 StreamData (Preliminary)

After starting the stream, the data will be sent as available in the following format. Reads oldest data from buffer.

Response:

Byte	
0	Checksum8
1	0xF9
2	4 + SamplesPerPacket
3	0xC0
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6-9	TimeStamp
10	PacketCounter
11	Errorcode
12-13	Sample0
44 (max)	Backlog
45 (max)	0x00

- TimeStamp: Reserved.
- PacketCounter: An 8-bit (0-255) counter that is incremented by one for each packet of data. Useful to make sure packets are in order and no packets are missing.
- Sample#: Stream data is placed in a FIFO (first in first out) buffer, so Sample0 is the oldest data read from the buffer and Sample15 is the 16th oldest sample. Analog input reading is returned justified as a 16-bit value. Differential readings are signed, while single-ended readings are unsigned.
- Backlog: When streaming, the processor acquires data at precise intervals, and transfers it to a FIFO buffer until it can be sent to the host. This value represents how much data is left in the buffer after this read.

5.2.13 StreamStop (Preliminary)

Command:

Byte	
0	0xB0
1	0xB0

Response:

Byte	
0	Checksum8
1	0xB1
2	Errorcode
3	0x00

5.3 Errorcodes

Following is a list of the low-level function errorcodes.

Code	
1	SCRATCH_WRT_FAIL
2	SCRATCH_ERASE_FAIL
3	DATA_BUFFER_OVERFLOW
4	ADC0_BUFFER_OVERFLOW
5	FUNCTION_INVALID
6	SWDT_TIME_INVALID
7	XBR_CONFIG_ERROR
16	FLASH_WRITE_FAIL
17	FLASH_ERASE_FAIL
18	FLASH_JMP_FAIL
19	FLASH_PSP_TIMEOUT
20	FLASH_ABORT_RECEIVED
21	FLASH_PAGE_MISMATCH
22	FLASH_BLOCK_MISMATCH
23	FLASH_PAGE_NOT_IN_CODE_AREA
24	MEM_ILLEGAL_ADDRESS
25	FLASH_LOCKED
26	INVALID_BLOCK
27	FLASH_ILLEGAL_PAGE
28	FLASH_TOO_MANY_BYTES
29	FLASH_INVALID_STRING_NUM
48	STREAM_IS_ACTIVE
49	STREAM_TABLE_INVALID
50	STREAM_CONFIG_INVALID
51	STREAM_BAD_TRIGGER_SOURCE
52	STREAM_NOT_RUNNING
53	STREAM_INVALID_TRIGGER
54	STREAM_ADC0_BUFFER_OVERFLOW
55	STREAM_SCAN_OVERLAP
56	STREAM_SAMPLE_NUM_INVALID
57	STREAM_BIPOLAR_GAIN_INVALID
58	STREAM_SCAN_RATE_INVALID
64	TIMER_INVALID_MODE
65	TIMER_QUADRATURE_AB_ERROR
66	TIMER_QUAD_PULSE_SEQUENCE
67	TIMER_BAD_CLOCK_SOURCE
68	TIMER_STREAM_ACTIVE
69	TIMER_PWMSTOP_MODULE_ERROR
70	TIMER_SEQUENCE_ERROR
71	TIMER_LINE_SEQUENCE_ERROR
72	TIMER_SHARING_ERROR
80	EXT_OSC_NOT_STABLE
81	INVALID_POWER_SETTING
82	PLL_NOT_LOCKED
96	INVALID_PIN
97	PIN_CONFIGURED_FOR_ANALOG
98	PIN_CONFIGURED_FOR_DIGITAL
99	IOTYPE_SYNCH_ERROR
100	INVALID_OFFSET
101	IOTYPE_NOT_VALID

A. Specifications

Specifications at 25 degrees C and Vusb/Vext = 5.0V, except where noted.

Parameter	Conditions	Min	Typical	Max	Units
General					
USB Cable Length				5	meters
Supply Voltage		4.0	5.0	5.25	volts
Typical Supply Current (1)		30			mA
Operating Temperature		-40		85	°C
Clock Error	~ 25 °C			±30	ppm
	-10 to 60 °C			±50	ppm
	-40 to 85 °C			±100	ppm
Typ. Command Execution Time (2)	USB high-high	0.6			ms
	USB other	4			ms
Vs Outputs					
Typical Voltage (3)	Self-Powered	4.75	5.0	5.25	volts
	Bus-Powered	4.0	5.0	5.25	
Maximum Current (3)	Self-Powered		450		mA
	Bus-Powered		50		mA

(1) Typical current drawn by the U3 itself, not including any user connections. Minimum value is the typical current when the device is idle. Maximum value is the typical current when the device is very busy.

(2) Total typical time to execute a single Feedback function with no analog inputs. Measured by timing a Windows application that performs 1000 calls to the Feedback function. See Section 3.1 for more timing information.

(3) These specifications are related to the power provided by the host/hub. Self- and bus-powered describes the host/hub, not the U3. Self-powered would apply to USB hubs with a power supply, all known desktop computer USB hosts, and some notebook computer USB hosts. An example of bus-powered would be a hub with no power supply, or many PDA ports. The current rating is the maximum current that should be sourced through the U3 and out of the Vs terminals.

Parameter	Conditions	Min	Typical	Max	Units
Analog Inputs					
Typical Input Range (1)	Single-Ended	0		2.44	volts
	Differential	-2.44		2.44	volts
	Special 0-3.6	0		3.6	volts
Max AIN Voltage to GND (2)	Valid Operation	-0.3		3.6	volts
Max AIN Voltage to GND (3)	No Damage, FIO	-10		10	volts
	No Damage, EIO	-6		6	volts
Input Bias Current (4)			TBD		nA
Input Impedance (4)			TBD		MΩ
Source Impedance (5)				10	kΩ
Resolution (No Missing Codes)			12		bits
Absolute Accuracy	Single-Ended		±TBD		% FS
	Differential				
	Special 0-3.6				
Temperature Drift			15		ppm/°C
Noise	Single-Ended		±TBD		counts
	Differential		±TBD		counts
	Special 0-3.6		±TBD		counts
Integral Linearity Error			±0.05		% FS
Differential Linearity Error			±1		counts
Stream Data Buffer Size			N/A		
Stream Speed (6)				N/A	
Channel-to-Channel Delay (7)	Comm./Resp.				µs

(1) With DAC1 disabled.

(2) This is the maximum voltage on any AIN pin compared to ground for valid measurements. Note that a differential channel has a minimum voltage of -2.44 volts, meaning that the positive channel can be 2.44 volts less than the negative channel, but no AIN pin can go more than 0.3 volts below ground.

(3) Maximum voltage, compared to ground, to avoid damage to the device. Protection level is the same whether the device is powered or not.

(4) This is the steady state input bias current and impedance. When the analog input multiplexer changes from one channel to another at a different voltage, more current is briefly required to change the charge on the internal capacitance.

(5) To meet specifications, the impedance of the source signal should be kept at or below the specified value.

(6) Stream not supported at this time.

(7) This is the time between each sample within a scan.

Parameter	Conditions	Min	Typical	Max	Units
Analog Outputs					
Nominal Output Range (1)	No Load	0.04		4.95	volts
	@ ±2.5 mA				volts
Resolution			8		bits
Absolute Accuracy	5% to 95% FS		±0.1		% FS
Noise					
Integral Linearity Error			±1		counts
Differential Linearity Error			±1		counts
Error Due To Loading	@ 100 µA		0.1		%
	@ 1 mA		1		%
Source Impedance			50		Ω
Short Circuit Current	Max to GND				mA
Slew Rate					V/µs
Digital I/O					
Low Level Input Voltage		-0.3		0.8	volts
High Level Input Voltage		2		5.8	volts
Maximum Input Voltage (2)	FIO	-10		10	volts
	EIO/CIO	-6		6	volts
Output Low Voltage (3)	No Load		0		volts
FIO	Sinking 1 mA		0.55		volts
EIO/CIO/MIO	Sinking 1 mA		0.18		volts
EIO/CIO/MIO	Sinking 5 mA		0.9		volts
Output High Voltage (3)	No Load		3.3		volts
FIO	Sourcing 1 mA		2.75		volts
EIO/CIO/MIO	Sourcing 1 mA		3.12		volts
EIO/CIO/MIO	Sourcing 5 mA		2.4		volts
Short Circuit Current (3)	FIO		6		mA
	EIO/CIO		18		mA
Output Impedance (3)	FIO		550		Ω
	EIO/CIO		180		Ω
Counter Input Frequency (4)				TBD	MHz
Input Timer Total Edge Rate (5)	No Stream			TBD	edges/s
	While Streaming			N/A	edges/s

(1) Maximum and minimum analog output voltage is limited by the supply voltages (Vs and GND). The specifications assume Vs is 5.0 volts. Also, the ability of the DAC output buffer to drive voltages close to the power rails, decreases with increasing output current, but in most applications the output is not sinking/sourcing much current as the output voltage approaches GND.

(2) Maximum voltage to avoid damage to the device. Protection works whether the device is powered or not, but continuous voltages over 5.8 volts or less than -0.3 volts are not recommened when the U3 is unpowered, as the voltage will attempt to supply operating power to the U3 possibly causing poor start-up behavior.

(3) These specifications provide the answer to the question: "How much current can the digital I/O sink or source?". For instance, if EIO0 is configured as output-high and shorted to ground, the current sourced by EIO0 into ground will be about 18 mA (3.3/180). If connected to a load that draws 5 mA, EIO0 can provide that current but the voltage will droop to about 2.4 volts instead of the nominal 3.3 volts. If connected to a 180 ohm load to ground, the resulting voltage and current will be about 1.65 volts @ 9 mA.

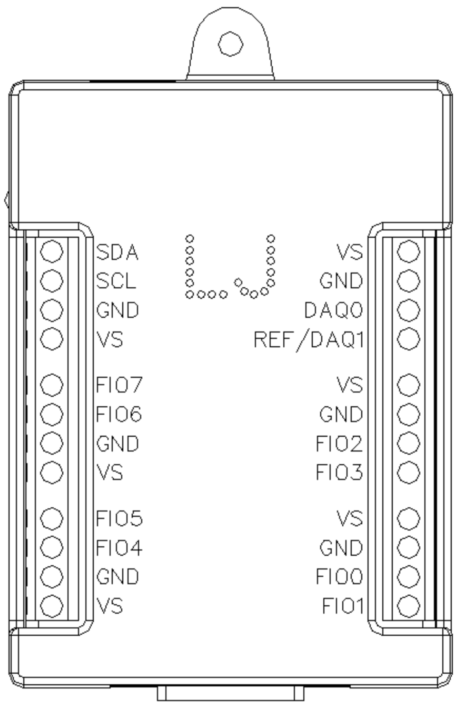
(4) Hardware counters. 0 to 3.3 volt square wave.

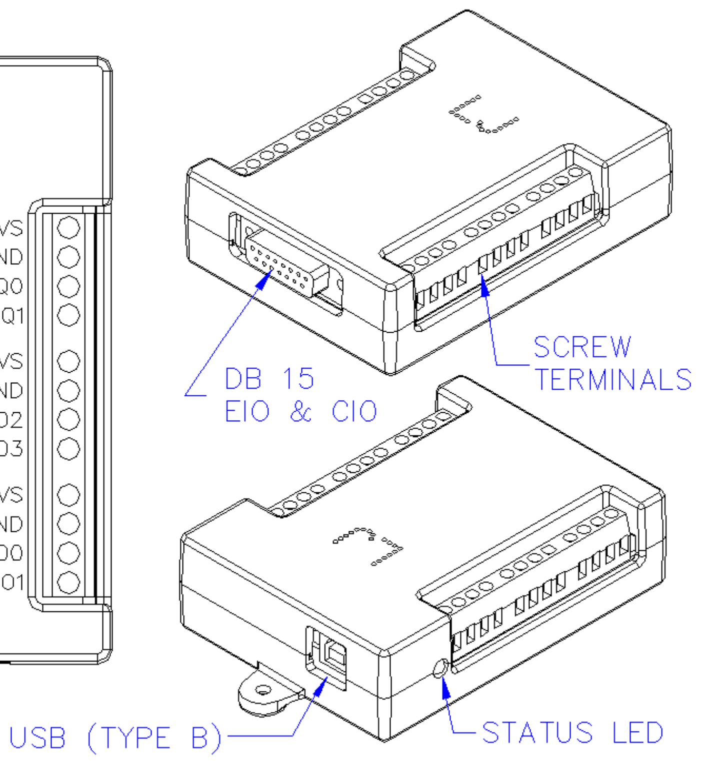
(5) To avoid missing edges, keep the total number of applicable edges on all applicable timers below this limit. See Section 2.9 for more information.

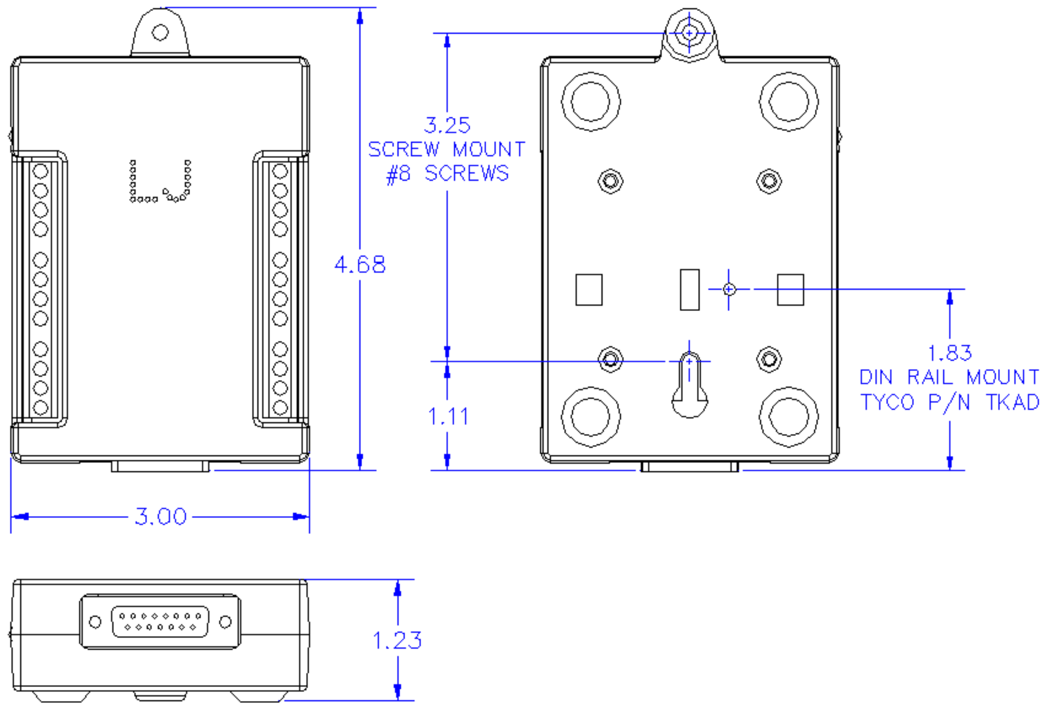
B. Enclosure & PCB Drawings

The square holes shown below are for a DIN rail mounting adapter: Tyco part #TKAD.

Units are inches.

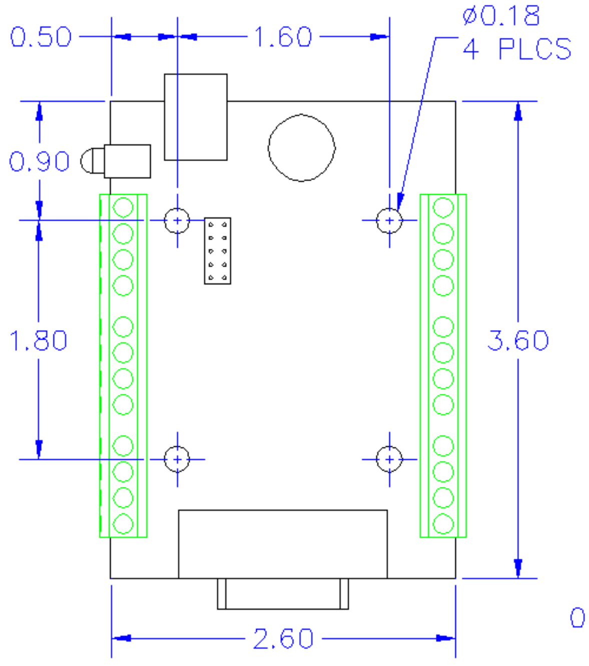






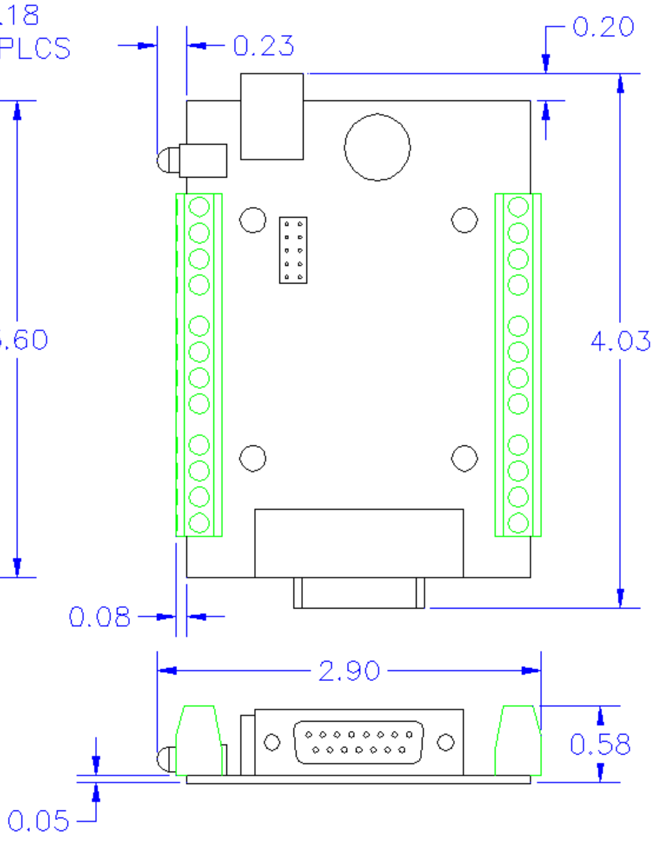
PCB MOUNTING HOLES

DIMENSIONS IN INCHES



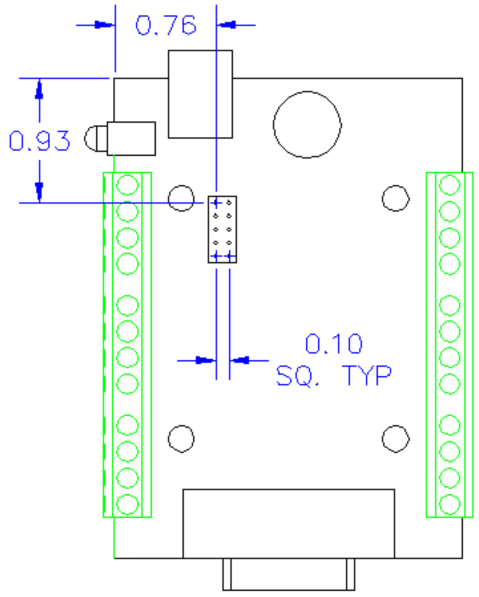
PCB AND CONNECTORS

DIMENSIONS IN INCHES



PIN HEADER

DIMENSIONS IN INCHES



0.76
0.93
0.10
SQ. TYP