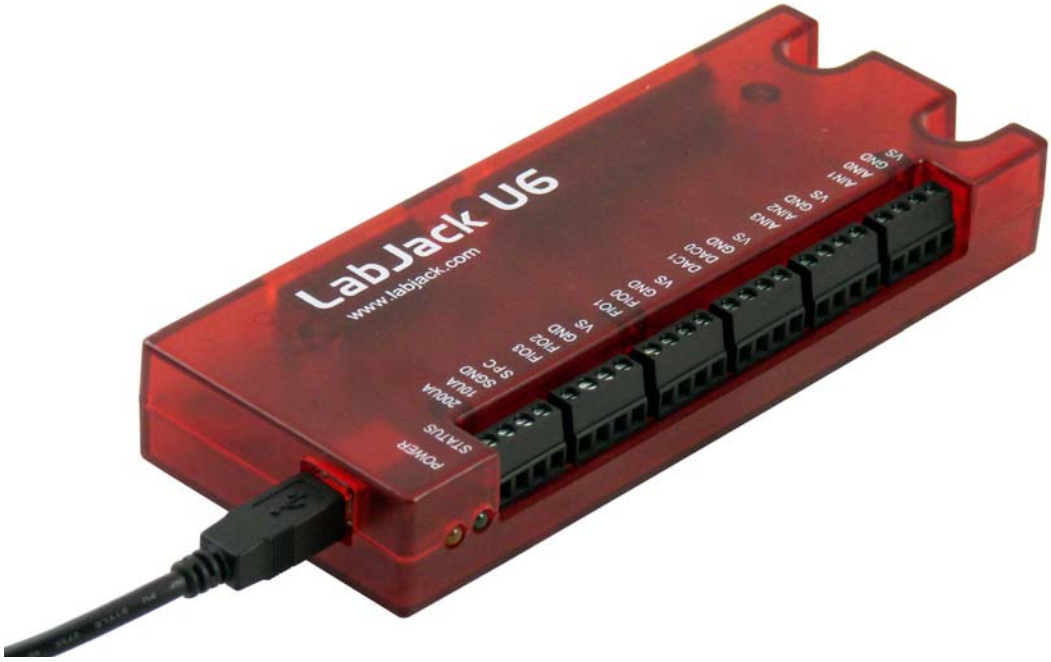
LabJack U6 (-Pro) User's Guide

Revision 0.99
November 3rd, 2009

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



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

LabJack designs and manufactures measurement and automation peripherals that enable the connection of a PC to the real-world. Although LabJacks have various redundant protection mechanisms, it is possible, in the case of improper and/or unreasonable use, to damage the LabJack and even the PC to which it is connected. LabJack Corporation will not be liable for any such damage.

Except as specified herein, LabJack Corporation makes no warranties, express or implied, including but not limited to any implied warranty or merchantability or fitness for a particular purpose. LabJack Corporation shall not be liable for any special, indirect, incidental or consequential damages or losses, including loss of data, arising from any cause or theory.

LabJacks and associated products are not designed to be a critical component in life support or systems where malfunction can reasonably be expected to result in personal injury. Customers using these products in such applications do so at their own risk and agree to fully indemnify LabJack Corporation for any damages resulting from such applications.

LabJack assumes no liability for applications assistance or customer product design. Customers are responsible for their applications using LabJack products. To minimize the risks associated with customer applications, customers should provide adequate design and operating safeguards.

Declaration of Conformity

Manufacturers Name: LabJack Corporation
Manufacturers Address: 3232 S Vance St STE 100, Lakewood, CO 80227, USA

Declares that the product

Product Name: LabJack U6 (-Pro)
Model Number: LJU6 (-Pro)

conforms to the following Product Specifications:

EMC Directive: 89/336/EEC

EN 55011 Class A
EN 61326-1: General Requirements

and is marked with CE

Warranty:

The LabJack U6 comes with a 1 year limited warranty from LabJack Corporation, covering this product and parts against defects in material or workmanship. The LabJack can be damaged by misconnection (such as connecting 120 VAC to any of the screw terminals), and this warranty does not cover damage obviously caused by the customer. If you have a problem, contact support@labjack.com for return authorization. In the case of warranty repairs, the customer is responsible for shipping to LabJack Corporation, and LabJack Corporation will pay for the return shipping.

LabJack U6 User's Guide Revision History

…

Table Of Contents

Table Of Figures

1. Installation on Windows

The UD driver requires a PC running Windows XP or Vista. For other operating systems, go to labjack.com for available support. Software will be installed to the LabJack directory which defaults to c:\Program Files\LabJack\.

Install the software first: Install the software using the CD or by downloading the latest UD installer from labjack.com. Although all necessary software is available at labjack.com, do not discard the CD as it includes a fully licensed copy of DAQFactory Express which is not available by download.

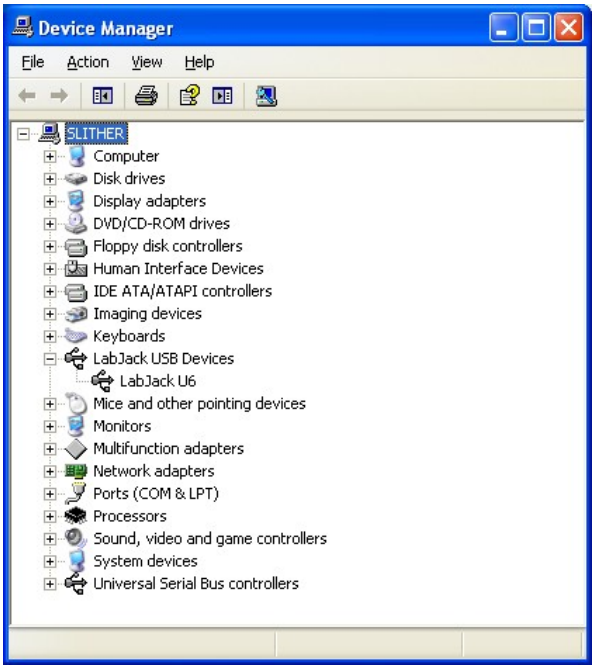
Connect the USB cable: The USB cable provides data and power. After the UD software installation is complete, connect the hardware and Windows should prompt with *"Found New Hardware"* and shortly after the Found New Hardware Wizard will open. When the Wizard appears allow Windows to install automatically by accepting all defaults.

Run LJControlPanel: From the Windows Start Menu, go to the LabJack group and run LJControlPanel. Click the "Find Devices" button, and an entry should appear for the connected U6 showing the serial number. Click on the "USB – 1" entry below the serial number to bring up the U6 configuration panel. Click on "Test" in the configuration panel to bring up the test panel where you can view and control the various I/O on the U6.

If LJControlPanel does not find the U6, check Windows Device Manager to see if the U6 installed correctly. One way to get to the Device Manager is:

Start => Control Panel => System => Hardware => Device Manager

The entry for the U6 should appear as in the following figure. If it has a yellow caution symbol or exclamation point symbol, right-click and select "Uninstall" or "Remove". Then disconnect and reconnect the U6 and repeat the Found New Hardware Wizard as described above.



1.1 Control Panel Application (LJControlPanel)

The LabJack Control Panel application (LJCP) handles configuration and testing of the U6. Click on the “Find Devices” button to search for connected devices.

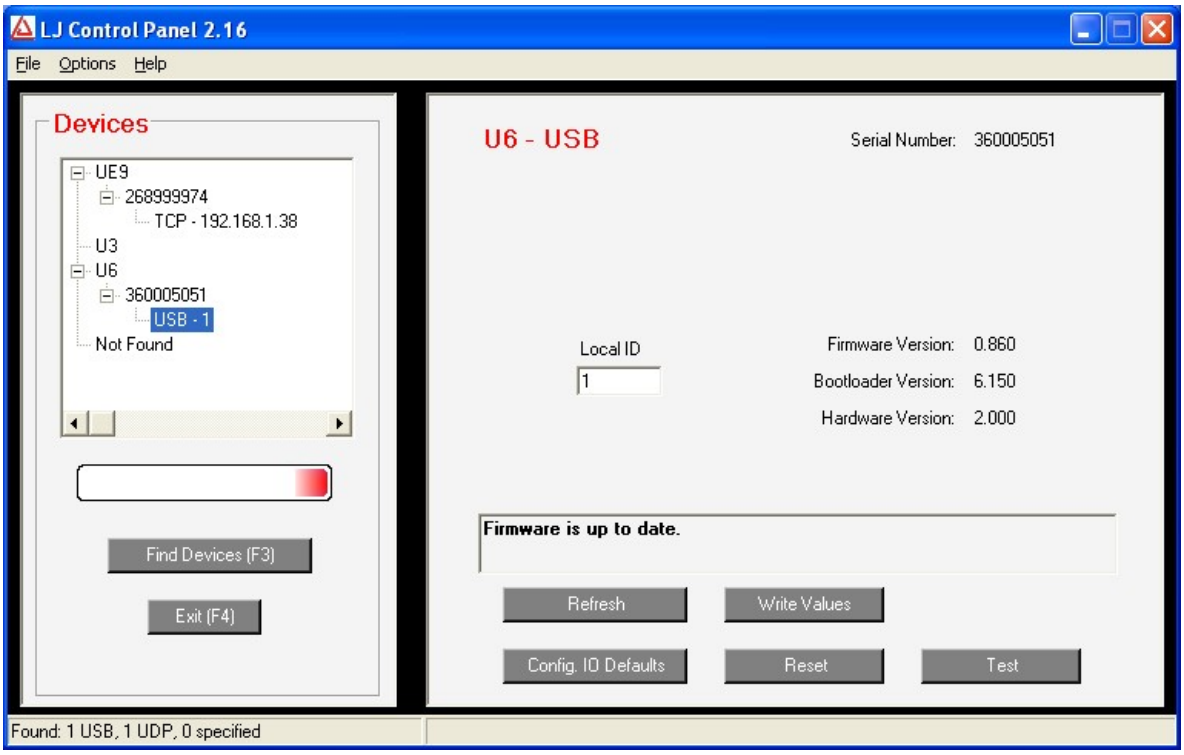


Figure 1-1. LJControlPanel Device Window

Figure 1-1 shows the results from a typical search. The application found a U6 connected by USB. The USB connection has been selected in Figure 1-1, bringing up the main device window on the right side.

- Refresh: Reload the window using values read from the device.
- Write Values: Write the Local ID from the window to the device.
- Config. IO Defaults: Opens the window shown in Figure 1-2.
- Reset: Click to reset the selected device.
- Test: Opens the window shown in Figure 1-3.

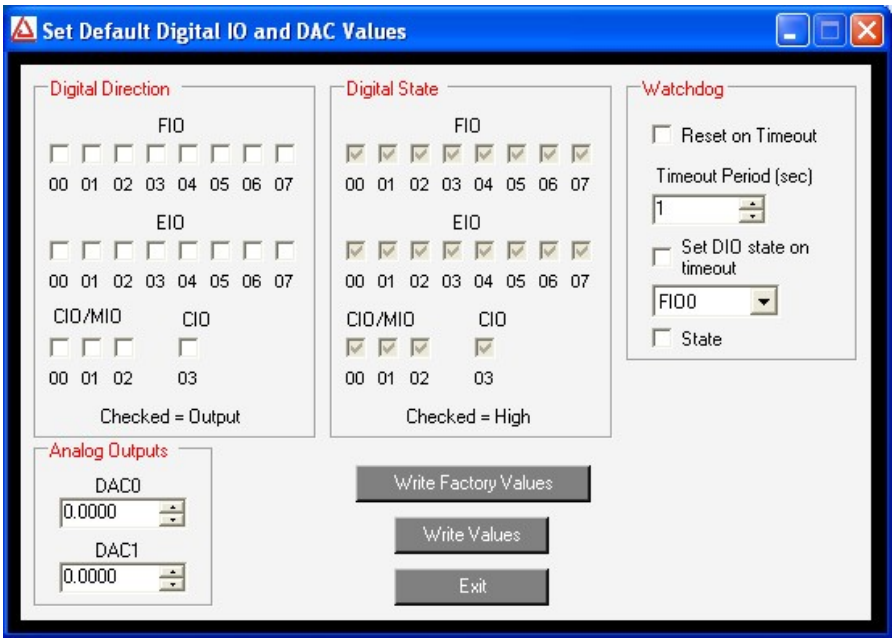


Figure 1-2. LJControlPanel U6 Configure Defaults Window

Figure 1-2 shows the configuration window for U6 defaults. These are the values that will be loaded by the U6 at power-up or reset. The factory defaults are shown above.

Figure 1-3 shows the U6 test window. This window continuously (once per second) writes to and reads from the selected LabJack.

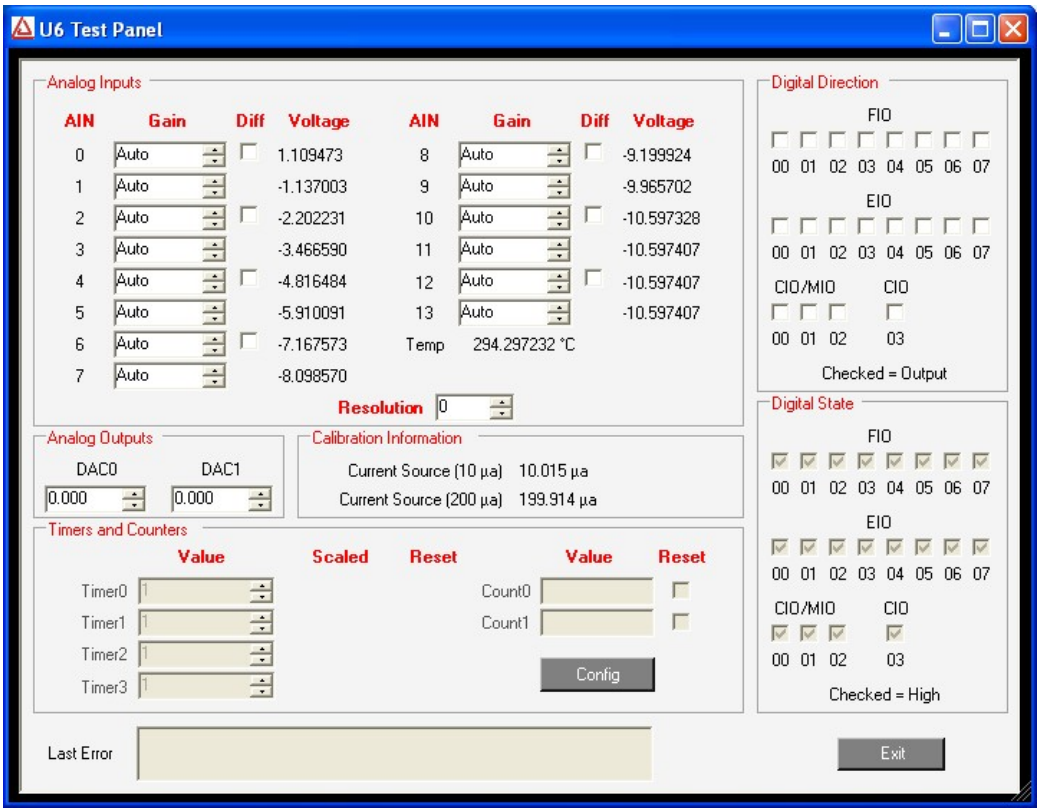


Figure 1-3. LJControlPanel U6 Test Window

Selecting Options=>Settings from the main LJControlPanel menu brings up the window shown in Figure 1-4. This window allows some features to of the LJControlPanel application to be customized.

Figure 1-4. 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 U6.
- Search for Ethernet devices using specified IP addresses: Does not apply to the U6.

1.2 Self-Upgrade Application (LJSelfUpgrade)

The processor in the U6 has field upgradeable flash memory. The self-upgrade application shown in Figure 1-5 programs the latest firmware onto the processor.

USB is the only interface on the U6, and first found is the only option for self-upgrading the U6, so no changes are needed in the "Connect by:" box. There must only be one U6 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.

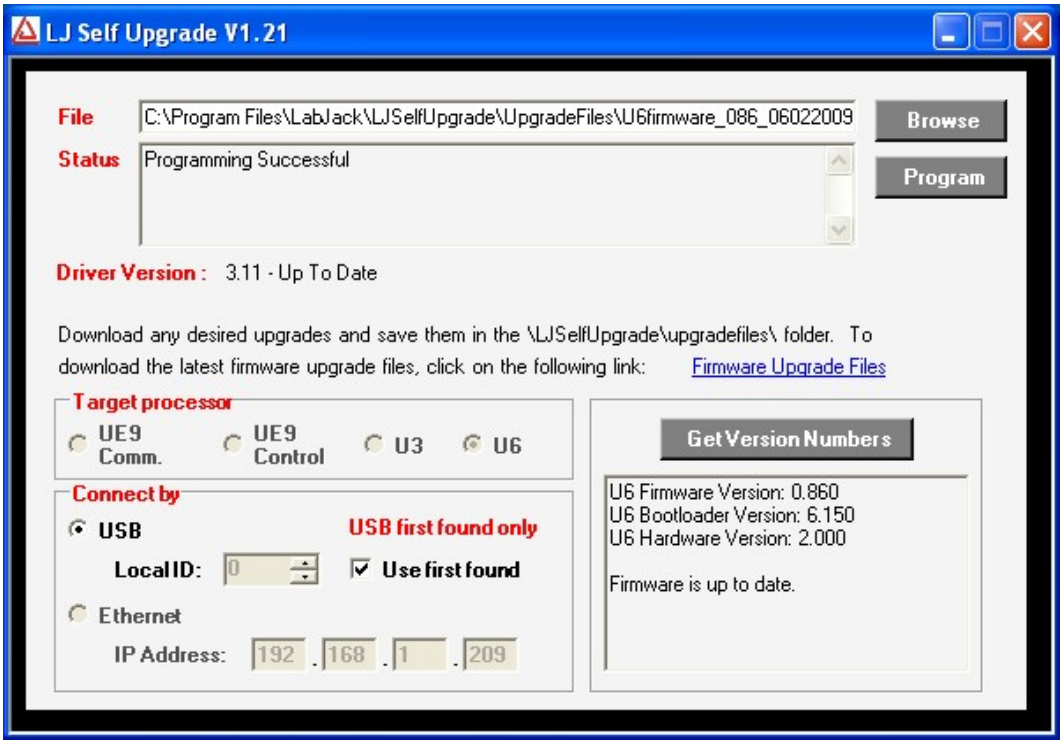


Figure 1-5. Self-Upgrade Application

If problems are encountered during programming, try the following:

1. Unplug the U6, wait 5 seconds then reconnect the U6. Click OK then press program again.

2. If step 1 does not fix the problem unplug the U6 and watch the LED while plugging the U6 back in. Follow the following steps based on the LED's activity.

    a. **If the LED is blinking continuously,** connect a jumper between FIO0 and SPC, then unplug the U6, wait 5 seconds and plug the U6 back in. Try programming again (disconnect the jumper before programming).
    b. **If the LED blinks several times and stays on,** connect a jumper between FIO1 and SPC, then unplug the U6, wait 5 seconds and plug the U6 back in. Try programming again (disconnect the jumper before programming).
    c. **If the LED blinks several times and stays off,** the U6 is not enumerating. Please restart your computer and try to program again.
    d. **If there is no LED activity,** connect a jumper between FIO1 and SPC, then unplug the U6, wait 5 seconds and plug the U6 back in. If the LED is blinking continuously click OK and program again (after removing the jumper). If the LED does not blink connect a jumper between FIO0 and SPC, then unplug the U6, wait 5 seconds and plug the U6 back in.

3. If there is no activity from the U6's LED after following the above steps, please contact support.

2. Hardware Description

The U6 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 4 analog inputs, both analog outputs, 4 flexible digital I/O (FIO), and both current sources. The screw terminals are arranged in blocks of 4, with each block consisting of Vs, GND, and two I/O. Also on this edge are two LEDs. One simply indicates power, while the other serves as a status indicator.

The DB Edge has 2 D-sub type connectors: a DB37 and DB15. The DB37 has some digital I/O and all the analog I/O. The DB15 has 12 additional digital I/O (3 are duplicates of DB37 I/O).

Figure 2-1. LabJack U6

2.1 USB

For information about USB installation, see Section 1.

The U6 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 U6 ground (GND), and USB ground is generally the same as the ground of the PC chassis and AC mains.

The details of the U6 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 LabJack vendor ID is 0x0CD5. The product ID for the U6 is 0x0006.

The USB interface consists of the normal bidirectional control endpoint (0 OUT & IN), 3 used bulk endpoints (1 OUT, 2 IN, 3 IN), and 1 dummy endpoint (3 OUT). Endpoint 1 consists of a 64 byte OUT endpoint (address = 0x01). Endpoint 2 consists of a 64 byte IN endpoint (address = 0x82). Endpoint 3 consists of a dummy OUT endpoint (address = 0x03) and a 64 byte IN endpoint (address = 0x83). Endpoint 3 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 always be on Endpoint 2. Endpoint 3 is only used to send stream data from the U6 to the host.

2.2 Power and Status LED

There is a yellow “Power” LED to indicate power, and a green “Status” LED controlled by the main processor.

The Power LED is connected to VS (with a series resistor). It indicates that some voltage is present on VS, but does not indicate whether the voltage is valid or not.

The Status LED flashes on reset and USB enumeration, then remains solid on with flashes to indicate communication (USB) traffic.

Normal Power-Up Status LED Behavior: When the USB cable is connected to the U6, the Status LED should blink a few times and then remain solid on.

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 near the upper-left of the device. 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 U6.

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. All Vs terminals are the same.

For information about powering the U6 from a source other than USB, see the OEM information in Section 2.13.

2.5 10UA and 200UA

The U6 has 2 fixed current source terminals useful for measuring resistance (thermistors, RTDs, resistors). The 10UA terminal provides about 10 µA and the 200UA terminal provides about 200 µA.

The actual value of each current source is noted during factory calibration and stored with the calibration constants on the device. These can be viewed using the test panel in LJControlPanel, or read programmatically.

The current sources can drive about 3 volts max, thus limiting the maximum load resistance to about 300 kΩ (10UA) and 15 kΩ (200UA).

Multiple resistances can be measured by putting them in series with a single current source and using differential analog inputs to measure the voltage across each resistance.

The following charts show the typical tempco of the current sources over temperature. The 10UA current source has a very low tempco across temperature. The 200 UA current source has a good tempco from about 0-50 degrees C, and outside of that range the effect of tempco will be more noticeable.

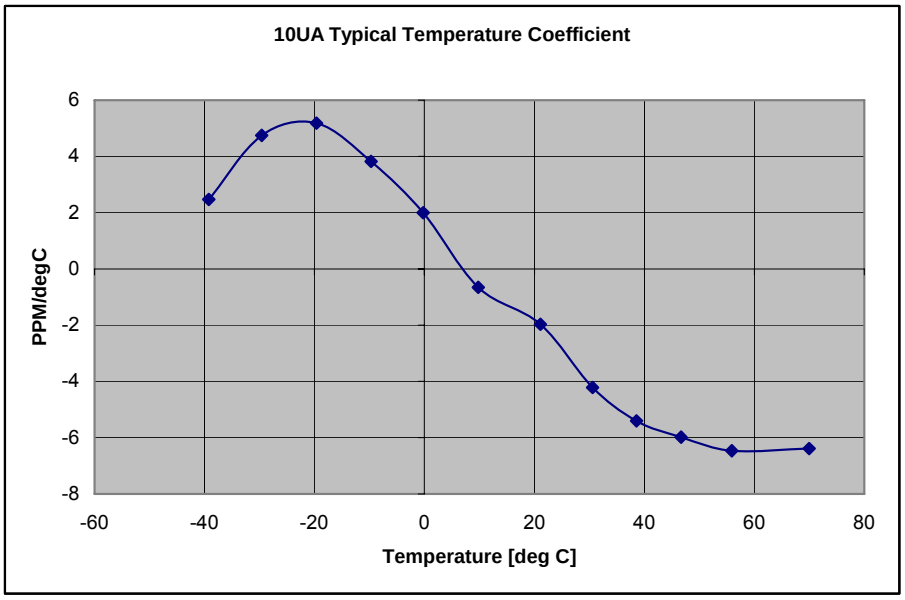


Figure 2-2. Typical Temperature Coefficient of the 10UA Source

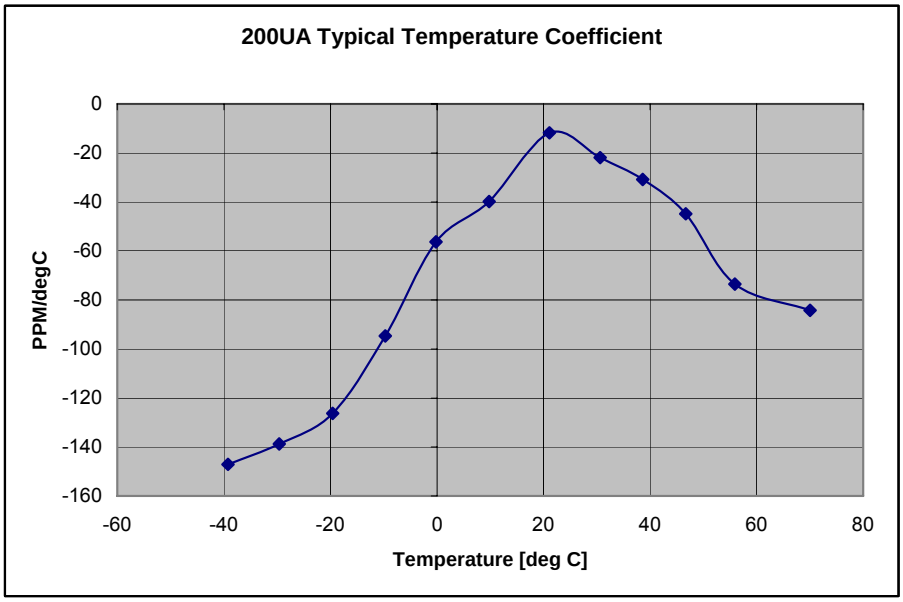


Figure 2-3. Typical Temperature Coefficient of the 200UA Source

2.6 AIN

The LabJack U6 has 14 user accessible analog inputs built-in. All the analog inputs are available on the DB37 connector, and the first 4 are also available on the built-in screw terminals.

The analog inputs have variable resolution, where the time required per sample increases with increasing resolution. The value passed for resolution is from 0-8, where 0 corresponds to default resolution, 1 is roughly 16-bit resolution (RMS or effective), and 8 is roughly 19-bit resolution. The U6-Pro has additional resolution settings 9-12 that use the alternate high-resolution converter (24-bit sigma-delta) and correspond to roughly 19-bit to 22-bit resolution.

The analog inputs are connected to a high impedance instrumentation amplifier. The inputs are not pulled to 0.0 volts, as that would reduce the input impedance, so readings obtained from floating channels will generally not be 0.0 volts. The readings from floating channels depend on adjacent channels and sample rate. See Section 2.6.3.8.

When scanning multiple channels, the nominal channel-to-channel delay is specified in Appendix A, and includes enough settling time to meet the specified performance. Some signal sources could benefit from increased settling, so a settling time parameter is available that adds extra delay between configuring the multiplexers and acquiring a sample. This extra delay will impact the maximum possible data rates.

2.6.1 Channel Numbers

The LabJack U6 has 16 total built-in analog inputs. Two of these are connected internally (AIN14/AIN15), leaving 14 user accessible analog inputs (AIN0-AIN13). The first 4 analog inputs, AIN0-AIN3, appear both on the screw terminals and on the DB37 connector. There is about 4.4 kΩ of resistance between the duplicated connections, so connecting signals to both will not short-circuit the signals but they will contend with each other.

Positive Channel #	
0-13	Single-Ended
0,2,4,6,8,10,12	Differential
14	Temp Sensor (deg K)
15	GND

Table 2-1. Positive Channel Numbers

Negative Channel #	
1,3,5,7,9,11,13	Differential
0,15,199	Single-Ended (GND)

Table 2-2. Negative Channel Numbers

The DB37 connector has 3 MIO lines (shared with CIO0-CIO2) designed to address expansion multiplexer ICs (integrated circuits), allowing for up to 112 total external analog inputs. The DG408 from Intersil is a recommended multiplexer, and a convenient ±12 volt power supply is available so the multiplexers can pass bipolar signals (see Vm+/Vm- discussion in Section 2.11). Figure 2-4 shows the typical connections for a pair of multiplexers.

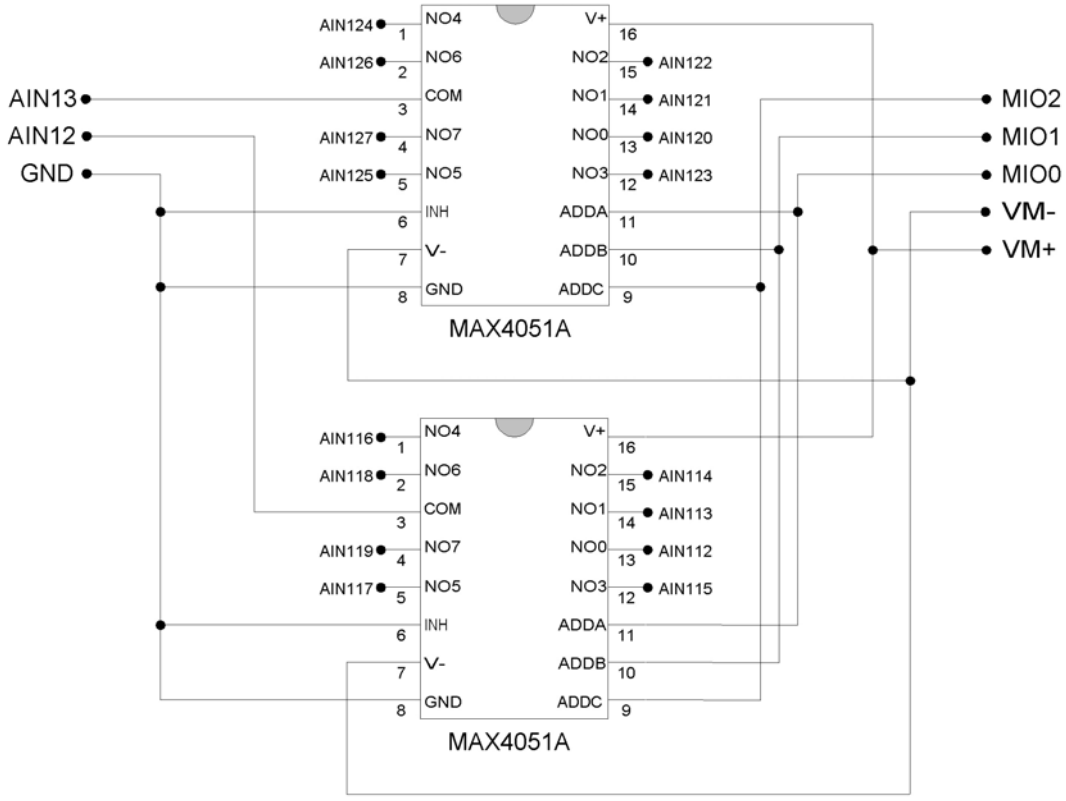


Figure 2-4. Typical External Multiplexer Connections (Wrong Mux Shown!)

To make use of external multiplexers, the user must be comfortable reading a simple schematic (such as Figure 2-4) and making basic connections on a solderless breadboard (such as the

EB37). Initially, it is recommended to test the basic operation of the multiplexers without the MIO lines connected. Simply connect different voltages to NO0 and NO1, connect ADDA/ADDB/ADDC to GND, and the NO0 voltage should appear on COM. Then connect ADDA to VS and the NO1 voltage should appear on COM.

If any of the AIN channel numbers passed to a U6 function are in the range 16-127 (extended channels), the MIO lines will automatically be set to output and the correct state while sampling that channel. For instance, a channel number of 28 will cause the MIO to be set to b100 and the ADC will sample AIN1. Channel number besides 16-127 will have no affect on the MIO. The extended channel number mapping is shown in Table 2-2.

For differential inputs, the positive channel must map to an even channel from 0-12, and the negative channel must be 8 higher.

In command/response mode, after sampling an extended channel the MIO lines remain in that same condition until commanded differently by another extended channel or another function. When streaming with any extended channels, the MIO lines are all set to output-low for any non extended analog channels. For special channels (digital/timers/counters), the MIO are driven to unspecified states. Note that the StopStream can occur during any sample within a scan, so the MIO lines will wind up configured for any of the extended channels in the scan. If a stream does not have any extended channels, the MIO lines are not affected.

U6 Channel	MIO Multiplexed Channels
0	16-23
1	24-31
2	32-39
3	40-47
4	48-55
5	56-63
6	64-71
7	72-79
8	80-87
9	88-95
10	96-103
11	104-111
12	112-119
13	120-127

Table 2-3. Expanded Channel Mapping

2.6.2 Converting Binary Readings to Voltages

Following are the nominal input voltage ranges for the analog inputs.

	Gain	Max V	Min V
Bipolar	1	10.1	-10.6
Bipolar	10	1.01	-1.06
Bipolar	100	0.101	-0.106
Bipolar	1000	0.0101	-0.0106

Table 2-4. Nominal Analog Input Voltage Ranges

The readings returned by the analog inputs are raw binary values (low-level functions). An approximate voltage conversion can be performed as:

Volts(uncalibrated) = (Bits/65536)*Span

Where span is the maximum voltage minus the minimum voltage from the table above. For a proper voltage conversion, though, use the calibration values (Slope and Offset) stored in the internal flash on the Control processor.

Volts = (Slope * Bits) + Offset

In both cases, "Bits" is always aligned to 16-bits, so if the raw binary value is 24-bit data it must be divided by 256 before converting to voltage. Binary readings are always unsigned integers.

Since the U6 uses multiplexers, all channels have the same calibration for a given input range.

Go to labjack.com for details about the location of the U6 calibration constants.

2.6.3 Typical Analog Input Connections

A common question is "can this sensor/signal be measured with the U6". Unless the signal has a voltage (referred to U6 ground) beyond the limits in Appendix A, it can be connected without damaging the U6, but more thought is required to determine what is necessary to make useful measurements with the U6 or any measurement device.

Voltage (versus ground): The single-ended analog inputs on the U6 measure a voltage with respect to U6 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 U6, 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 U6, 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 U6 analog input will cause noticeable voltage errors due to the impedance of the source. See Appendix A for the recommended maximum source impedance.

Resolution (and Accuracy): Based on the selected input range and resolution of the U6, the resolution can be determined in terms of voltage or engineering units. For example, assume some temperature sensor provides a 0-10 mV signal, corresponding to 0-100 degrees C. Samples are then acquired with the U6 using the ±10 volt input range and 16-bit resolution, resulting in a voltage resolution of about 20/65536 = 305 μV. That means there will be about 33 discrete steps across the 10 mV span of the signal, and the overall resolution is 0.03 degrees C. Accuracy (which is different than resolution) 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 U6.

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 U6 itself, is with an analog output. All I/O on the U6 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. 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 thermocouple or photocell where the sensor leads are not shorted to any external voltages. Such a sensor typically has two leads. The positive lead connects to an AINx terminal and the negative lead connects to a GND terminal.

An exception might be a thermocouple housed in a metal probe where the negative lead of the thermocouple is shorted to the metal probe housing. If this probe is put in contact with something (engine block, pipe, …) that is connected to ground or some other external voltage, care needs to be taken to insure valid measurements and prevent damage.

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 AINx 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 AINx and a 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. An instrumentation amplifier is required to convert the differential signal to signal-ended, and probably also to amplify the signal.

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. Such a case is obviously isolated and easy to keep isolated. In practical applications, though, signals thought to be isolated are often not at all, or perhaps are isolated at some time but the isolation is easily lost at another time.

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 nanoamps for the U6.

The SGND terminal 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 U6. If no large voltages are noted, connect the ground to U6 SGND with a 100 Ω series resistor. Then again use the DMM to measure the voltage of each signal wire before connecting to the U6.

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 U6. 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 U6 ground.

2.6.3.5 Amplifying small signal voltages

This section has general information about external signal amplification. The U6 has an outstanding amplifier built-in. Combined with the high resolution capability of the U6, an external amplifier is seldom needed, and in many cases will actually degrade noise and accuracy performance.

For a do-it-yourself solution, the following figure shows an operational amplifier (op-amp) configured as non-inverting:

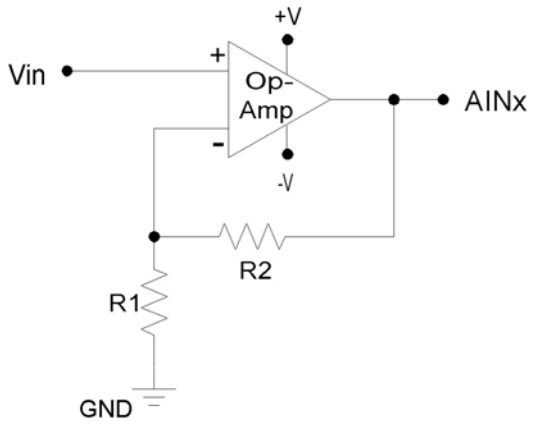


Figure 2-5. Non-Inverting Op-Amp Configuration

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 max supply rating for the OPA344 is 5.5 volts, so for applications using Vm+/Vm- (±12 volts), the LT1490A from Linear Technologies (linear.com) might be a good option.

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 ±10 volts (and resistance measurement)

The nominal maximum analog input voltage range for the U6 is ±10 volts. The basic way to handle higher voltages is with a resistive voltage divider. The following figure shows the resistive voltage divider assuming that the source voltage (Vin) is referred to the same ground as the U6 (GND).

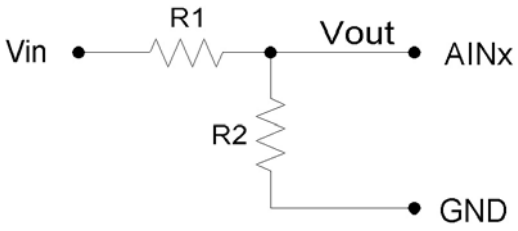


Figure 2-6. Voltage Divider Circuit

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, R1 = R2 = 10 kΩ provides a divide by 2, so a ±20 volt input will be scaled to ±10 volts and a 0-20 volt input will be scaled to 0-10 volts.

The divide by 2 configuration where R1 = R2 = 10 kΩ, presents a 20 kΩ load to the source, meaning that a ±10 volt signal will have to be able to source/sink up to ±500 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).

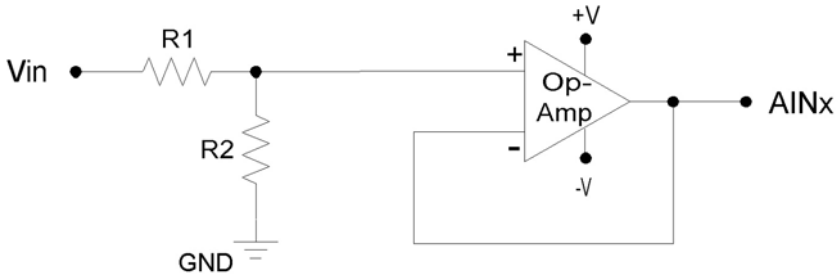


Figure 2-7. Buffered Voltage Divider Circuit

The op-amp is chosen to have low input bias currents so that large resistors can be used in the voltage divider. The LT1490A from Linear Technologies (linear.com) is a good choice for dual-supply applications. The LT1490A only draws 40 µA of supply current, thus many of these amps can be powered from the Vm+/Vm- supply on the U6, and can pass signals in the ±10 volt range. Since the input bias current is only -1 nA, large divider resistors such as R1 = R2 = 470 kΩ will only cause an offset of about -470 µV, and yet present a load to the source of about 1 megaohm.

For 0-5 volt applications, where the amp will be powered from Vs and GND, the LT1490A is not the best choice. When the amplifier input voltage is within 800 mV of the positive supply, the bias current jumps from -1 nA to +25 nA, which with R1 = 470 kΩ will cause the offset to change from -470 µV to +12 mV. A better choice in this case 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.

Another option is the LJTick-Divider which plugs into the U6 screw-terminals. It is similar to the buffered divider shown in Figure 2-7. More information is available at labjack.com.

The information above also applies to resistance measurement. A common way to measure resistance is to build a voltage divider as shown in Figure 2-6, where one of the resistors is known and the other is the unknown. If Vin is known and Vout is measured, the voltage divider equation can be rearranged to solve for the unknown resistance.

A great way to measure resistance is using the current sources on the U6. By sending this known current through the resistance and measuring the voltage that results across, the value of the resistance can be calculated. Common resistive sensors are thermistors and RTDs.

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 (loop-powered) current loop sensor. The current shunt shown in the figure is simply a resistor.

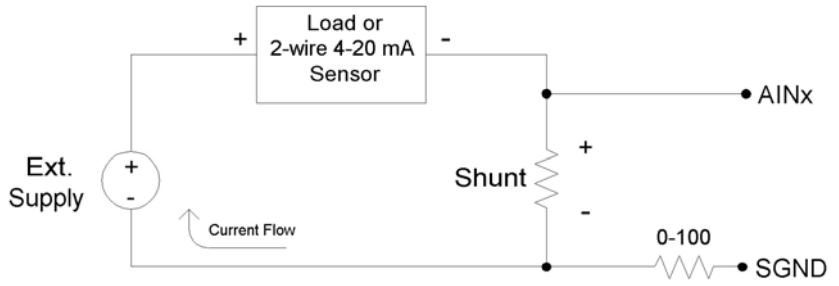


Figure 2-8. Current Measurement With Arbitrary Load or 2-Wire 4-20 mA Sensor

When measuring a 4-20 mA signal, a typical value for the shunt would be 240 Ω. This results in a 0.96 to 4.80 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 9.8 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 2.5 volts of drop is the most that can be tolerated without affecting the load, a 2.4 Ω resistor could be used. That equates to 2.4 watts, though, which would require a special high wattage resistor. A better solution would be to use a lower resistance shunt, and rely on the outstanding performance of the U6 to resolve the smaller signal. 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 240 Ω which results in 0.96 to 4.80 volts.

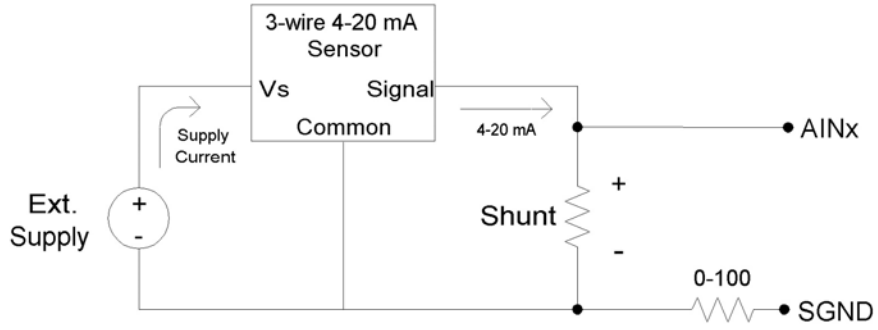


Figure 2-9. Current Measurement With 3-Wire 4-20 mA (Sourcing) Sensor

The sensor shown in Figure 2-9 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 U6 ground, the sinking type of sensor presents a couple of problems, as the voltage across the shunt resistor is differential (neither side is at ground) and at least one side of the resistor has a high common mode voltage (equal to the positive sensor supply). If the sensor and/or U6 are isolated, a possible solution is to connect the sensor signal or positive sensor supply to U6 ground (instead of sensor ground). This requires a good understanding of grounding and isolation in the system. The LJTick-CurrentShunt is often a simple solution.

Both Figure 2-8 and 2-9 show a 0-100 Ω resistor in series with SGND, which is discussed in general in Section 2.6.3.4. In this case, if SGND is used (rather than GND), a direct connection (0 Ω) should be good.

The best way to handle 4-20 mA signals is with the LJTick-CurrentShunt, which is a two channel active current to voltage converter module that plugs into the U6 screw-terminals. More information is available at labjack.com.

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, ...), but obviously that degrades the input impedance. For the specific case of pulling a floating channel to 0 volts at gain=1 and resolution=1, a 100 kΩ resistor to GND can typically be used to provide analog input readings within 100 mV of ground.

2.6.4 Internal Temperature Sensor

The U6 has an internal temperature sensor. The sensor is physically located near the AIN3 screw-terminal. It is labeled U17 on the PCB, and can be seen through the gap between the AIN3 terminal and adjacent VS terminal.

The U6 enclosure typically makes a 1 degree C difference in the temperature at the internal sensor. Calibrated readings are typically 0.5 degrees high with the enclosure installed, and 0.5 degrees low with the PCB in free air.

The screw terminals AIN0-AIN3 are typically 3 degrees C above ambient with the enclosure installed, so when the internal temperature sensor is used for thermocouple cold junction compensation on AIN0-AIN3, it is recommended to add 2.5 degrees C to the readings.

With the UD driver, the internal temperature sensor is read by acquiring analog input channel 14 and returns degrees K.

2.7 DAC

There are two DACs (digital-to-analog converters or analog outputs) on the U6. Each DAC can be set to a voltage between about 0.02 and 5 volts with 12-bits of resolution.

Although the DAC values are based on an absolute reference voltage, and not the supply voltage, the DAC output buffers are powered internally by Vs and thus the maximum output is limited to slightly less than Vs.

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/4.86)*65536

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

Bits = (Slope * Volts) + Offset

The DACs appear both on the screw terminals and on the DB37 connector. These connections are electrically the same.

The power-up condition of the DACs can be configured by the user. From the factory, the DACS default to enabled at minimum voltage (~0 volts). Note that even if the power-up default for a line is changed to a different voltage or disabled, there is a delay of about 100 ms at power-up where the DACs are in the factory default condition.

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.

There is an accessory available from LabJack called the LJTick-DAC that provides a pair of 14-bit analog outputs with a range of ±10 volts. The LJTick-DAC plugs into any digital I/O block, and thus up to 10 of these can be used per U6 to add 20 analog outputs.

2.7.1 Typical Analog Output Connections

2.7.1.1 High Current Output

The DACs on the U6 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 2-5. 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 0.02 to 5 volts. For other unipolar ranges, an op-amp in the non-inverting configuration (Figure 2-5) can be used to provide the desired gain. For example, to increase the maximum output from 4.86 volts to 10.0 volts, a gain of 2.06 is required. If R2 (in Figure 2-5) is chosen as 100 kΩ, then an R1 of 93.1 kΩ is the closest 1% resistor that provides a gain greater than 2.06. 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). For example, the EB37 experiment board provides power supplies that are typically ±9.5 volts. If these supplies are used to power the LT1490A op-amp (linear.com), which has rail-to-rail capabilities, the outputs could be driven very close to ±9.5 volts. If ±12 or ±15 volt supplies are available, then the op-amp might not need rail-to-rail capabilities to achieve the desired output range.

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.

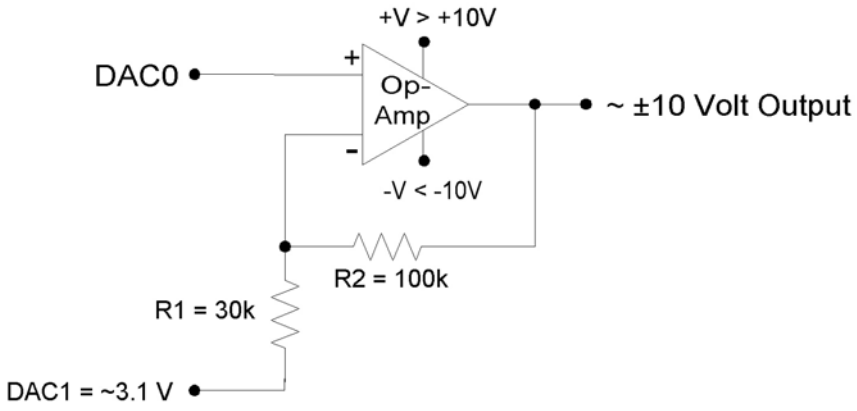


Figure 2-10. ±10 Volt DAC Output Circuit

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 U6 has 20 digital I/O. 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
20-22 MIO0-MIO2

Note that on the U6 CIO0-CIO2 are shared with MIO0-MIO2. That is, CIO0 is shorted to MIO0, CIO1 is shorted to MIO1, and CIO2 is shorted to MIO2.

Some signals appear in multiple locations. Outputs might use both locations at the same time, but inputs should only have a connection to one location at a time. FIO0-FIO3 appear on both the DB37 connector and screw terminals with 940 Ω between the duplicate connections. MIO/CIO0-MIO/CIO2 appear on both the DB15 connector and DB37 connector with 0 Ω between the duplicate connections.

The U6 has 8 FIO. The first 4 lines, FIO0-FIO3, appear both on the screw terminals and on the DB37 connector. The upper 4 lines appear only on the DB37 connector. By default, the FIO

lines are digital I/O, but they can also be configured as up to 4 timers and 2 counters (see Timers/Counters Section of this User's Guide).

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.

Up to 6 of the FIO/EIO lines can be configured as timers and counters. These are enabled sequential starting from FIO0-EIO0 (determined by pin offset). Thus, any sequential block of 1-6 digital I/O, starting from FIO0 to EIO0, can be configured as up to 4 timers and up to 2 counters.

MIO are standard digital I/O that also have a special multiplexer control function described in Section 2.6 above (AIN). The MIO are addressed as digital I/O bits 20-22 by the Windows driver. The MIO hardware (electrical specifications) is the same as the EIO/CIO hardware.

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 U6 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 100 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. See Section 3.1 for timing information. For information about using the digital I/O under the Windows LabJackUD driver, see Section 4.3.5.

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 function ControlConfig, the parameter FIODir is a single byte (8 bits) that writes/reads the power-up direction of each of the 8 FIO lines:

- if FIODir is 0, all FIO lines are input,
- if FIODir is 1 (2^0), FIO0 is output, FIO1-FIO7 are input,
- if FIODir is 5 ($2^0 + 2^2$), FIO0 and FIO2 are output, all other FIO lines are input,
- if FIODir 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 U6 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 U6 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 22 kΩ series resistor, about 19 volts will be dropped across the resistor, resulting in a current of 1.1 mA, which is no problem for the U6. The series resistor should be 22 kΩ or less, to make sure the voltage on the I/O line when low is pulled below 0.8 volts.

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 U6, then no additional ground connection is used. If the signal is known to not have a common ground with the U6, then the signal ground can simply be connected to U6 GND. If there is uncertainty about the relationship between signal ground and U6 ground (e.g. possible common ground through AC mains), then a ground connection with a 100 Ω series resistor is generally recommended (see Section 2.6.3.4).

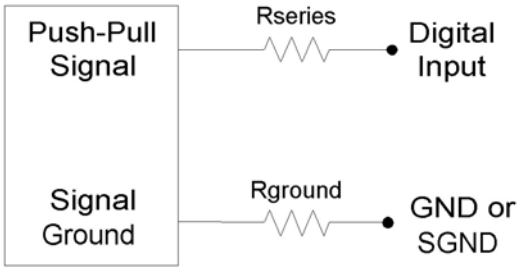


Figure 2-11. Driven Signal Connection To Digital Input

Figure 2-11 shows typical connections. Rground is typically 0-100 Ω. Rseries is typically 0 Ω (short-circuit) for 3.3/5 volt logic, or 22 kΩ (max) 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 U6 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 or NPN) 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 U6 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.

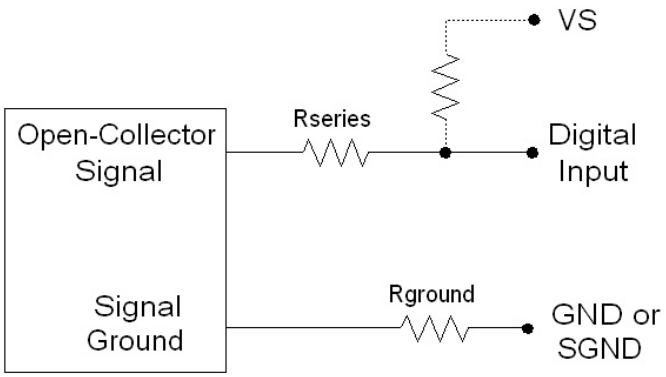


Figure 2-12. Open-Collector (NPN) Connection To Digital Input

Figure 2-12 shows typical connections. Rground is typically 0-100 Ω, Rseries is typically 0 Ω, and the external pull-up resistor is generally not required. If there is some uncertainty about whether the signal is really open-collector or could drive a voltage beyond 5.8 volts, use an Rseries of 22 kΩ as discussed in Section 2.8.1.1, and the input should be compatible with an open-collector signal or a driven signal up to at least 48 volts. 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 U6 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 U6 ground and the other side to a digital input. The behavior is very similar to the open-collector described above.

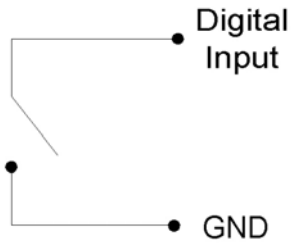


Figure 2-13. Basic Mechanical Switch Connection To Digital Input

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 2.9.1 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.

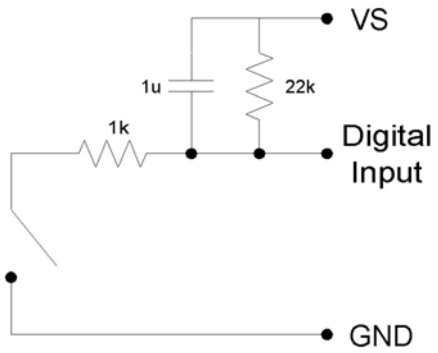


Figure 2-14. Passive Hardware Debounce

Figure 2-14 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, but solid-state relays (SSRs) can usually be controlled directly by the digital I/O. The SSR is connected as shown in the following diagram, where VS (~5 volts) connects to the positive control input and the digital I/O line connects to the negative control input (sinking configuration).

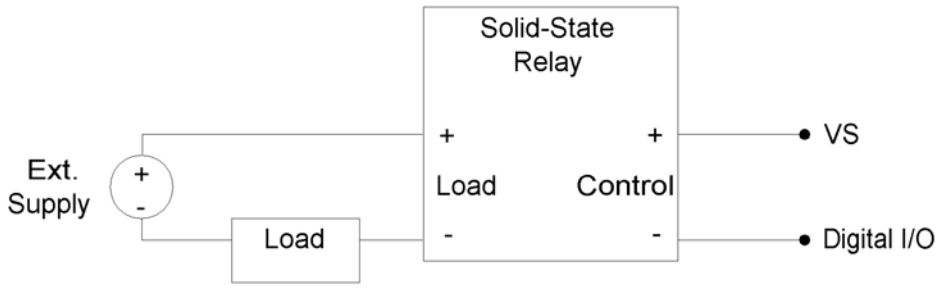


Figure 2-15. Relay Connections (Sinking Control, High-Side Load Switching)

When the digital line is set to output-low, control current flows and the relay turns on. When the digital line is set to input, control current does not flow and the relay turns off. When the digital

line is set to output-high, some current flows, but whether the relay is on or off depends on the specifications of a particular relay. It is recommended to only use output-low and input.

For example, the Series 1 (D12/D24) or Series T (TD12/TD24) relays from Crydom specify a max turn-on of 3.0 volts, a min turn-off of 1.0 volts, and a nominal input impedance of 1500 Ω.

- When the digital line is set to output-low, it is the equivalent of a ground connection with 180 Ω (EIO/CIO/MIO) or 550 Ω (FIO) in series. When using an EIO/CIO/MIO line, the resulting voltage across the control inputs of the relay will be about 5*1500/(1500+180) = 4.5 volts (the other 0.5 volts is dropped across the internal resistance of the EIO/CIO/MIO line). With an FIO line the voltage across the inputs of the relay will be about 5*1500/(1500+550) = 3.7 volts (the other 1.3 volts are dropped across the internal resistance of the FIO line). Both of these are well above the 3.0 volt threshold for the relay, so it will turn on.

- When the digital line is set to input, it is the equivalent of a 3.3 volt connection with 100 kΩ in series. The resulting voltage across the control inputs of the relay will be close to zero, as virtually all of the 1.7 volt difference (between VS and 3.3) is dropped across the internal 100 kΩ resistance. This is well below the 1.0 volt threshold for the relay, so it will turn off.

- When the digital line is set to output-high, it is the equivalent of a 3.3 volt connection with 180 Ω (EIO/CIO/MIO) or 550 Ω (FIO) in series. When using an EIO/CIO/MIO line, the resulting voltage across the control inputs of the relay will be about 1.7*1500/(1500+180) = 1.5 volts. With an FIO line the voltage across the inputs of the relay will be about 1.7*1500/(1500+550) = 1.2 volts. Both of these in the 1.0-3.0 volt region that is not defined for these example relays, so the resulting state is unknown.

Most mechanical relays require more control current than SSRs, and cannot be controlled directly by the digital I/O on the U6. To control higher currents with the digital I/O, some sort of buffer is used. Some options are a discrete transistor (e.g. 2N2222), a specific chip (e.g. ULN2003), or an op-amp.

Note that the U6 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 U6 and accepts up to 12 industry standard I/O modules (designed for Opto22 G4 modules and similar).

Another accessory available from LabJack is the LJTick-RelayDriver. This is a two channel module that plugs into the U6 screw-terminals, and allows two digital lines to each hold off up to 50 volts and sink up to 200 mA. This allows control of virtually any solid-state or mechanical relay.

2.9 Timers/Counters

The U6 has 4 timers (Timer0-Timer3) 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, Timer2, Timer3, Counter0, then Counter1), starting with FIO0+TimerCounterPinOffset. Some examples:

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

FIO0=Timer0

1 Timer enabled, Counter0 disabled, Counter1 enabled, and TimerCounterPinOffset=0:
FIO0=Timer0
FIO1=Counter1

2 Timers enabled, Counter0 enabled, Counter1 enabled, and TimerCounterPinOffset=8:
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 FIO1 to EIO1, depending on TimerCounterPinOffset. On the other hand, FIO2 (for example), is always on the screw terminal labeled FIO2, and AIN3 is always on the screw terminal labeled FIO3.

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. An error will result if an attempt is made to enable Counter0 when one of these frequencies is configured. Similarly, an error will result if an attempt is made to configure one of these frequencies when Counter0 is enabled.

Applicable digital I/O are automatically configured as input or output as needed when timers and counters are enabled, and stay that way when the timers/counters are disabled.

See Section 2.8.1 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-Timer3) have various modes available:

Timer Modes

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	4 MHz
1	12 MHz
2	48 MHz (Default)
3	1 MHz /Divisor
4	4 MHz /Divisor
5	12 MHz /Divisor
6	48 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. The divisor has a range of 0-255, where 0 corresponds to a division of 256.

2.9.1 Timer Mode Descriptions

2.9.1.1 PWM Output (16-Bit, 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 Ranges	
TimerBaseClock		Divisor=1	Divisor=256
0	4 MHz	61.04	N/A
1	12 MHz	183.11	N/A
2	48 MHz (Default)	732.42	N/A
3	1 MHz /Divisor	15.26	0.060
4	4 MHz /Divisor	61.04	0.238
5	12 MHz /Divisor	183.11	0.715
6	48 MHz /Divisor	732.42	2.861

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.

PWM output starts by setting the digital line to output-low for the specified amount of time. The output does not necessarily start instantly, but rather waits for the internal clock to roll. For example, if the PWM frequency is 100 Hz, that means the period is 10 milliseconds, and thus after the command is received by the device it could be anywhere from 0 to 10 milliseconds before the start of the PWM output.

2.9.1.2 PWM Output (8-Bit, 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 Ranges	
TimerBaseClock		Divisor=1	Divisor=256
0	4 MHz	15625.00	N/A
1	12 MHz	46875.00	N/A
2	48 MHz (Default)	187500.00	N/A
3	1 MHz /Divisor	3906.25	15.259
4	4 MHz /Divisor	15625.00	61.035
5	12 MHz /Divisor	46875.00	183.105
6	48 MHz /Divisor	187500.00	732.422

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.

PWM output starts by setting the digital line to output-low for the specified amount of time. The output does not necessarily start instantly, but rather waits for the internal clock to roll. For example, if the PWM frequency is 100 Hz, that means the period is 10 milliseconds, and thus after the command is received by the device it could be anywhere from 0 to 10 milliseconds before the start of the PWM output.

2.9.1.3 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 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 the other timer, delays of a few microseconds are possible.
- If a stream is in progress, every sample is acquired in a high-priority interrupt. These interrupts could cause delays on the order of 10 microseconds.
- The always active U6 system timer causes an interrupt 61 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.

Note that the minimum measurable period is limited by the edge rate limit discussed in Section 2.9.2.

See Section 3.2.1 for a special condition if stream mode is used to acquire timer data in this mode.

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.

2.9.1.4 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. Note that a duty cycle of 0% or 100% does not have any edges.

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.

Note that the minimum measurable high/low time is limited by the edge rate limit discussed in Section 2.9.2.

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. The duty cycle reset is special, in that if the signal is low at the time of reset, the high-time/low-time registers are set to 0/65535, but if the signal is high at the time of reset, the high-time/low-time registers are set to 65535/0. Thus if no edges occur before the next read, it is possible to tell if the duty cycle is 0% or 100%.

2.9.1.5 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.

2.9.1.6 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.

The debounce period is set by writing the timer value. The low byte of the timer value is a number from 0-255 that specifies a debounce period in 16 ms increments (plus an extra 0-16 ms of variability):

Debounce Period = (0-16 ms) + (TimerValue * 16 ms)

In the high byte (bits 8-16) 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 16-32 ms and negative edges will be counted. When the input detects a negative edge, it increments the count by 1, and then waits 16-32 ms before re-arming the edge detector. Any negative edges within the debounce period are ignored. This is good behavior for a normally-high signal where the switch closure causes a brief low signal. The debounce period can be set long enough so that bouncing on both the switch closure and switch open is ignored.

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.

2.9.1.7 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 Frequency Ranges	
		Divisor=1	Divisor=1
TimerBaseClock		Value=1	Value=256
0	4 MHz	2000000.0	7812.50
1	12 MHz	6000000.0	23437.50
2	48 MHz (Default)	24000000.0	93750.00
		Divisor=1	Divisor=256
		Value=1	Value=256
3	1 MHz /Divisor	500000.0	7.629
4	4 MHz /Divisor	2000000.0	30.518
5	12 MHz /Divisor	6000000.0	91.553
6	48 MHz /Divisor	24000000.0	366.211

The frequency output has a -3 dB frequency of about 10 MHz on the FIO lines. Accordingly, at high frequencies the output waveform will get less square and the amplitude will decrease.

The output does not necessarily start instantly, but rather waits for the internal clock to roll. For example, if the output frequency is 100 Hz, that means the period is 10 milliseconds, and thus after the command is received by the device it could be anywhere from 0 to 10 milliseconds before the start of the frequency output.

2.9.1.8 Quadrature Input (Mode 8)

Requires 2 timer channels used in adjacent pairs (0/1 or 2/3). Even timers will be quadrature channel A, and odd timers will be quadrature channel B. Timer#Value passed has no effect. The U6 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 even and odd 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.

2.9.1.9 Timer Stop Input (Mode 9)

This mode should only be assigned to Timer1. On every rising edge seen by the external pin, this mode increments a 16-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 MSW of the read from this timer mode returns the number of edges counted, but does not increment past the stop count value. The LSW of the read returns edges waiting for.

2.9.1.10 System Timer Low/High Read (Modes 10 & 11)

The LabJack U6 has a free-running internal 64-bit system timer with a frequency of 4 MHz. 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. This system timer cannot be reset, and is not affected by the timer clock.

If using both modes 10 & 11, read both in the same low-level command and read 10 before 11.

Mode 11, the upper 32 bits of the system timer, is not available for stream reads. Note that when streaming on the U6, the timing is known anyway (elapsed time = scan rate * scan number) and it does not make sense to stream the system timer modes 10 or 11.

2.9.1.11 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.

Note that the minimum measurable period is limited by the edge rate limit discussed in Section 2.9.2.

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 U6 operations do not affect the output.

The edge-detecting timer input modes do require U6 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 30,000 per second. That means that in the case of a single timer, there should be no more than 1 edge per 33 µs. For multiple timers, all can process an edge simultaneously, but if for instance both timers get an edge at the same time, 66 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 (7,000 per second), and since each edge requires processing time the sustainable stream rates can also be reduced.

2.10 SPC (or VSPC)

The SPC (possibly labeled VSPC) terminal is use for manually resetting default values or jumping in/out of flash programming mode.

2.11 DB37

The DB37 connector brings out analog inputs, analog outputs, FIO, and other signals. Some signals appear on both the DB37 connector and screw terminals, so care must be taken to avoid a short circuit.

DB37 Pinouts

1	GND	14	AIN9	27	Vs
2	PIN2 (200uA)	15	AIN7	28	Vm+
3	FIO6	16	AIN5	29	DAC1
4	FIO4	17	AIN3	30	GND
5	FIO2	18	AIN1	31	AIN12
6	FIO0	19	GND	32	AIN10
7	MIO1/CIO1	20	PIN20 (10uA)	33	AIN8
8	GND	21	FIO7	34	AIN6
9	Vm-	22	FIO5	35	AIN4
10	GND	23	FIO3	36	AIN2
11	DAC0	24	FIO1	37	AIN0
12	AIN13	25	MIO0/CIO0		
13	AIN11	26	MIO2/CIO2		

Some signals appear in multiple locations. Outputs might use both locations at the same time, but inputs should only have a connection to one location at a time. AIN0-AIN3 appear on both the DB37 connector and screw terminals with 4.4 kΩ between the duplicate connections. FIO0-FIO3 appear on both the DB37 connector and screw terminals with 940 Ω between the duplicate connections. 200uA and 10uA appear on both the DB37 connector and screw terminals with 0 Ω between the duplicate connections. DAC0-DAC1 appear on both the DB37 connector and screw terminals with 0 Ω between the duplicate connections. MIO/CIO0-MIO/CIO2 appear on both the DB15 connector and DB37 connector with 0 Ω between the duplicate connections.

Ground, Vs, AIN, DAC, FIO, and MIO are all described in earlier sections.

Vm+/Vm- are bipolar power supplies intended to power external multiplexer ICs such as the DG408 from Intersil. The multiplexers can only pass signals within their power supply range, so Vm+/Vm- can be used to pass bipolar signals. Nominal voltage is ±12 volts at no load. Both

lines have a 100 ohm source impedance, and are designed to provide 2.5 mA or less. This is the same voltage supply used internally by the U6 to bias the analog input amplifier and multiplexers. If this supply is loaded more than 2.5 mA, the voltage can droop to the point that the maximum analog input range is reduced. If this supply is severely overloaded (e.g. short circuited), then damage could eventually occur.

If Vm+/Vm- are used to power multiplexers, series diodes are recommended as shown in Figure 9 of the Intersil DG408 datasheet. Not so much to protect the mux chips, but to prevent current from going back into Vm+/Vm-. Use Schottky diodes to minimize voltage drop.

On the U6, PIN2/PIN20 bring out the 200uA/10uA current sources.

2.11.1 CB37 Terminal Board

The CB37 terminal board from LabJack connects to the U6's DB37 connector and provides convenient screw terminal access to all lines. The CB37 is designed to connect directly to the U6, but can also connect via a 37-line 1:1 male-female cable.

When using the analog connections on the CB37, the effect of ground currents should be considered, particularly when a cable is used and substantial current is sourced/sunk through the CB37 terminals.

For instance, a test was done with a 3 foot cable between the CB37 and U6, and a 100 ohm load placed from Vs to GND on the CB37 (~50 mA load). A single-ended measurement of AIN4 shorted to CB37 GND returned about 2100 µV. Even with just the 5 mA ground current due to the LED on the CB37, about 200 uV of offset was noted. With a direct connection of the CB37 to the U6, these offsets dropped to about 170 uV and 8 uV. A measurement of AIN5 shorted to AGND resulted in no noticeable offset in all cases.

When any sizeable cable lengths are involved, a good practice is to separate current carrying ground from ADC reference ground. An easy way to do this on the CB37 is to use GND as the current source/sink, and use AGND as the reference ground. This works well for passive sensors (no power supply), such as a thermocouple, where the only ground current is the return of the input bias current of the analog input.

2.11.2 EB37 Experiment Board

The EB37 experiment board connects to the LabJack U6's DB37 connector and provides convenient screw terminal access. Also provided is a solderless breadboard and useful power supplies. The EB37 is designed to connect directly to the LabJack, but can also connect via a 37-line 1:1 male-female cable.

2.12 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 U6 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		

2.12.1 CB15 Terminal Board

The CB15 terminal board connects to the LabJack U6'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.

2.12.2 RB12 Relay Board

The RB12 provides a convenient interface for the U6 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.

2.13 OEM Connector Options

As of this writing, the U6 is only produced in the normal form factor with screw-terminals and DB connectors, but the PCB does have alternate holes available for 0.1” pin-header installation.

Connectors J2 and J3 provide pin-header alternatives to the DB15 and DB37 connectors. The J2 and J3 holes are always present, but are obstructed when the DB15 and DB37 are installed:

J2

1	GND	2	VS
3	CIO0	4	CIO1
5	CIO2	6	CIO3
7	GND	8	EIO0
9	EIO1	10	EIO2
11	EIO3	12	EIO4
13	EIO5	14	EIO6
15	EIO7	16	GND

J3

1	GND	2	GND	3	PIN20 (10uA)
4	PIN2 (200uA)	5	FIO7	6	FIO6
7	FIO5	8	FIO4	9	FIO3
10	FIO2	11	FIO1	12	FIO0
13	MIO0/CIO0	14	MIO1/CIO1	15	MIO2/CIO2
16	GND	17	Vs	18	Vm-
19	Vm+	20	GND	21	DAC1
22	DAC0	23	GND	24	AIN13
25	AIN12	26	AIN11	27	AIN10
28	AIN9	29	AIN8	30	AIN7
31	AIN6	32	AIN5	33	AIN4
34	AIN3	35	AIN2	36	AIN1
37	AIN0	38	GND	39	GND
40	GND				

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 U6.

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 U6. The Windows UD 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 command/response mode. The time varies primarily with the number of analog inputs requested, and is not noticeably affected by the number of digital I/O, DAC, timer, and counter operations, except when the packet size is big enough that multiple low-level commands must be used.

These times were measured using the example program “allio.c” (VC6_LJUD). The program executes a loop 1000 times and divides the total time by 1000, and thus includes everything (Windows latency, UD driver overhead, communication time, U6 processing time, etc.).

Following is the milliseconds for a single channel command/response AIN read at the different resolution index values.

Res Index	Res (bits)	ms
1	15.9	0.6
2	16.4	0.6
3	16.9	0.6
4	17.5	0.6
5	17.9	0.7
6	18.4	0.8
7	18.8	1.2
8	19.1	1.9
9	19.6	4.1
10	20.6	14.2
11	21.6	68
12	21.9	161

Res is the effective (RMS) resolution. +/-10 volt range used for this test.

Table 3-1. Typical Feedback Function Execution Times (+/-10 volt range)

A “USB high-high” configuration means the U6 is connected to a high-speed USB2 hub which is then connected to a high-speed USB2 host. Even though the U6 is not a high-speed USB device, such a configuration does provide improved performance.

3.2 Stream Mode

The highest input data rates are obtained in stream mode. Stream is a continuous hardware timed input mode where a list of channels is scanned at a specified scan rate. The scan rate specifies the interval between the beginning of each scan. The samples within each scan are acquired as fast as possible.

As samples are collected, they are placed in a small FIFO buffer on the U6, until retrieved by the host. The buffer typically holds 984 samples, but the size ranges from 512 to 984 depending on the number of samples per packet. Each data packet has various measures to ensure the integrity and completeness of the data received by the host.

The U6 uses a feature called auto-recovery. If the buffer overflows, the U6 will continue streaming but discard data until the buffer is emptied, and then data will be stored in the buffer again. The U6 keeps track of how many packets are discarded and reports that value. Based on the number of packets discarded, the UD driver adds the proper number of dummy samples (-9999.0) such that the correct timing is maintained.

The table below shows various stream performance parameters. Some systems might require a USB high-high configuration to obtain the maximum speed. A “USB high-high” configuration means the U6 is connected to a high-speed USB2 hub which is then connected to a high-speed USB2 host. Even though the U6 is not a high-speed USB device, such a configuration does often provide improved performance.

Stream data rates over USB can also be limited by other factors such as speed of the PC and program design. One general technique for robust continuous streaming would be increasing the priority of the stream process.

A sample is defined as a single conversion of a single channel, while a scan is defined as a single conversion of all channels being acquired. That means the maximum scan rate for a stream of five channels is 50k/5 = 10 kscans/second.

Res Index	Max Stream (Samples/s)	ENOB (RMS)	ENOB (Noise-Free)	Noise (16-bit Counts)	Interchannel Delay (μs)
1	50000	15.8	13.4	±3.0	
2	30000	16.2*	14.0	±2.0	
3	16000	16.6*	14.4	±1.5	
4	8800	17.0*	14.4	±1.5	
5	4500	17.1*	15.0	±1.0	
6	2200	17.3*	16.0	±0.5	
7	1000	17.7*	16.0	±0.5	
8	500	18.7*	16.0	±0.5	

Table 3-4. Stream Performance (+/-10 volt range)

***Note**: Stream mode literally returns only 16-bits of binary data per sample. The RMS resolution values exceeding 16.0 reflect the low noise of the 16-bit data.

ENOB stands for effective number of bits. The first ENOB column is the commonly used “effective” resolution, and can be thought of as the resolution obtained by most readings. This data is calculated by collecting 1000 samples and evaluating the standard deviation (RMS noise). The second ENOB column is the noise-free resolution, and is the resolution obtained by all readings. This data is calculated by collecting 1000 samples and evaluating the maximum

value minus the minimum value (peak-to-peak noise). Similarly, the Noise Counts column is the peak-to-peak noise based on counts from a 16-bit reading.

Interchannel delay is the time between successive channels within a stream scan.

3.2.1 Streaming Digital Inputs, Timers, and Counters

There are special channel numbers that allow digital inputs, timers, and counters, to be streamed in with analog input data.

Channel#	
193	EIO_FIO
200	Timer0
201	Timer1
210	Counter0
211	Counter1
224	TC_Capture

Table 3-5. Special Stream Channels

Channel number 193 returns the input states of 16 bits of digital I/O. FIO is the lower 8 bits and EIO is the upper 8 bits.

Channels 200-201 and 210-211 retrieve the least significant word (LSW, lower 2 bytes) of the specified timer/counter. At the same time that any one of these is sampled, the most significant word (MSW, upper 2 bytes) of that particular timer/counter is stored in an internal capture register (TC_Capture), so that the proper value can be sampled later in the scan. For any timer/counter where the MSW is wanted, channel number 224 must be sampled after that channel and before any other timer/counter channel. For example, a scan list of {200,224,201,224} would get the LSW of Timer0, the MSW of Timer0, the LSW of Timer1, and the MSW of Timer1. A scan list of {200,201,224} would get the LSW of Timer0, the LSW of Timer1, and the MSW of Timer1 (MSW of Timer0 is lost).

Adding these special channels to the stream scan list does not configure those inputs. If any of the FIO or EIO lines have been configured as outputs, timers, or counters, a channel 193 read will still be performed without error but the values from those bits should be ignored. The timers/counters (200-224) must be configured before streaming using normal timer/counter configuration commands.

The timing for these special channels is the same as for normal analog channels. For instance, a stream of the scan list {0,1,200,224,201,224} counts as 6 channels, and the maximum scan rate is determined by taking the maximum sample rate at the specified resolution and dividing by 6.

Special care must be taken when streaming timers configured in mode 2 or 3 (32-bit period measurement). It is possible for the LSW to roll, but the MSW be captured before it is incremented. That means the resulting value will be low by 65536 clock ticks, which is easy to detect in many applications, but if this is an unacceptable situation then only the LSW or MSW should be used and not both.

Mode 11, the upper 32 bits of the system timer, is not available for stream reads. Note that when streaming on the U6, the timing is known anyway (elapsed time = scan rate * scan number) and it does not make sense to stream the system timer modes 10 or 11.

4. LabJackUD High-Level Driver

The low-level U6 functions are described in Section 5, but most Windows applications will use the LabJackUD driver instead.

The driver requires a PC running Windows XP or Vista. 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\System32\.

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 (Add).
- Execute the list (Go).
- Read the result of each request (Get).

At the core, the UD driver only has 4 basic functions: Open, AddRequest, Go, and GetResult. Then with these few functions, there are many constants used to specify the desired actions. When programming in any language, it is recommended to have the header file handy, so that constants can be copied and pasted into the code.

The first type of constant is an IOType, which is always passed in the IOType parameter of a function call. One example of an IOType is the constant *LJ_ioPUT_DAC,* which is used to update the value of an analog output (DAC).

The second type of constant is a Channel Constant, also called a Special Channel. These constants are always passed in the Channel parameter of a function call. For the most part, these are used when a request is not specific to a particular channel, and go with the configuration IOTypes (*LJ_ioPUT_CONFIG* or *LJ_ioGET_CONFIG*). One example of a Special Channel is the constant *LJ_chLOCALID*, which is used to write or read the local ID of the device.

The third major type of constant used by the UD driver is a Value Constant. These constants are always passed in the Value parameter of a function call. One example of a Value Constant is the constant *LJ_tmPWM8*, which specifies a timer mode. This constant has a numeric value of 1, which could be passed instead, but using the constant *LJ_tmPWM8* makes for programming code that is easier to read.

Following is pseudocode that performs various actions. First, a call is done to open the device. The primary work done with this call is finding the desired device and creating a handle that points to the device for further function calls. In addition, opening the device performs various configuration and initialization actions, such as reading the calibration constants from the device:

```
//Use the following line to open the first found LabJack U6
//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 U6 over USB.
lngErrorcode = OpenLabJack (LJ_dtU6, LJ_ctUSB, "1", TRUE, &lngHandle);
```

Second, a list of requests is built in the UD driver using AddRequest calls. This does not involve any low-level communication with the device, and thus the execution time is relatively instantaneous:

```
//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 single-ended read from AIN3.
lngErrorcode = AddRequest (lngHandle, LJ_ioGET_AIN, 3, 0, 0, 0);
```

Third, the list of requests is processed and executed using a Go call. In this step, the driver determines which low-level commands must be executed to process all the requests, calls those low-level functions, and stores the results. This example consists of two requests, one analog input read and one analog output write, which can both be handled in a single low-level Feedback call (Section 5.2.5):

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

Finally, GetResult calls are used to retrieve the results (errorcodes and values) that were stored by the driver during the Go call. This does not involve any low-level communication with the device, and thus the execution time is relatively instantaneous:

```
//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 U6 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 U6, 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 ePut/eGet method, and they are generally interchangeable. See Section 4.3 for more pseudocode examples.

All the request and result 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 ?? 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 errorcode. 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_dtU6" 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_dtU6” 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 –** Pass the local ID or serial number 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 errorcode.

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 Section 4.3.
- **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 eAddGoGet()

This function passes multiple requests via arrays, then executes a GoOne() and returns all the results via the same arrays.

The parameters that start with "*a" are arrays, and all must be initialized with at least a number of elements equal to NumRequests.

Declaration:

```
LJ_ERROR _stdcall eAddGoGet (   LJ_HANDLE Handle,
                                long NumRequests,
                                long *aIOTypes,
                                long *aChannels,
                                double *aValues,
                                long *ax1s,
                                long *aRequestErrors,
                                long *GoError,
                                long *aResultErrors)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **Handle –** Handle returned by OpenLabJack().
- **NumRequests –** This is the number of requests that will be made, and thus the number of results that will be returned. All the arrays must be initialized with at least this many elements.
- **aIOTypes –** An array which is the list of IOTypes.
- **aChannels –** An array which is the list of Channels.
- **aValues –** An array which is the list of Values to write.
- **ax1s –** An array which is the list of x1s.

Outputs:

- **aValues –** An array which is the list of Values read.
- **aRequestErrors –** An array which is the list of errorcodes from each AddRequest().
- **GoError –** The errorcode returned by the GoOne() call.
- **aResultErrors –** An array which is the list of errorcodes from each GetResult().

4.2.5 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, …).

Declaration:

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

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **Handle –** Handle returned by OpenLabJack().
- **IOType –** The type of request. See Section 4.3.
- **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**

4.2.6 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.

Declaration:
LJ_ERROR _stdcall Go()

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

- **None**

Outputs:

- **None**

4.2.7 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.

Declaration:
LJ_ERROR _stdcall GoOne(LJ_HANDLE Handle)

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

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

Outputs:

- **None**

4.2.8 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 Section 4.3.
- **Channel –** The channel number of the particular IOType.

Outputs:

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

4.2.9 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.

4.2.10 DoubleToStringAddress()

Some special-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.

4.2.11 StringToDoubleAddress()

Some special-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.

4.2.12 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**

4.2.13 ErrorToString()

Outputs a string describing the given errorcode 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.

4.2.14 GetDriverVersion()

Returns the version number of this Windows LabJack driver.

Declaration:
double _stdcall GetDriverVersion();

Parameter Description:
Returns: Driver version.
Inputs:

- **None**

Outputs:

- **None**

4.2.15 TCVoltsToTemp()

A utility function to convert thermocouple voltage readings to temperature.

Declaration:

```
LJ_ERROR _stdcall TCVoltsToTemp (   long TCType,
                                    double TCVolts,
                                    double CJTempK,
                                    double *pTCTempK)
```

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

- **TCType –** A constant that specifies the thermocouple type, such as LJ_ttK.
- **TCVolts –** The thermocouple voltage.
- **CJTempK –** The temperature of the cold junction in degrees K.

Outputs:

- **pTCTempK –** Returns the calculated thermocouple temperature.

4.2.16 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 might 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**

4.2.17 eAIN()

An “easy” function that returns a reading from one analog input. This is a simple alternative to the very flexible IOType based method normally used by this driver.

When needed, this function automatically configures the specified channel(s) for analog input.

Declaration:

```
LJ_ERROR _stdcall eAIN (    LJ_HANDLE Handle,
                            long ChannelP,
                            long ChannelN,
                            double *Voltage,
                            long Range,
                            long Resolution,
                            long Settling,
                            long Binary,
                            long Reserved1,
                            long Reserved2)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **Handle –** Handle returned by OpenLabJack().
- **ChannelP –** The positive AIN channel to acquire.
- **ChannelN –** The negative AIN channel to acquire. For differential readings on the U6, this should be and odd number equal to ChannelP+1. For single-ended readings on the U6, this parameter should be 0 or 15.
- **Range –** Pass a range constant.
- **Resolution –** Pass a resolution index.
- **Settling –** Pass a settling factor.
- **Binary –** If this is nonzero (True), the Voltage parameter will return the raw binary value.
- **Reserved (1&2) –** Pass 0.

Outputs:

- **Voltage –** Returns the analog input reading, which is generally a voltage.

4.2.18 eDAC()

An “easy” function that writes a value to one analog output. This is a simple alternative to the very flexible IOType based method normally used by this driver.

Declaration:

```
LJ_ERROR _stdcall eDAC (    LJ_HANDLE Handle,
                            long Channel,
                            double Voltage,
                            long Binary,
```

long Reserved1,
long Reserved2)

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

- **Handle –** Handle returned by OpenLabJack().
- **Channel –** The analog output channel to write to.
- **Voltage –** The voltage to write to the analog output.
- **Binary –** If this is nonzero (True), the value passed for Voltage should be binary.
- **Reserved (1&2) –** Pass 0.

4.2.19 eDI()

An "easy" function that reads the state of one digital input. This is a simple alternative to the very flexible IOType based method normally used by this driver.

When needed, this function automatically configures the specified channel as a digital input.

Declaration:
LJ_ERROR _stdcall eDI (LJ_HANDLE Handle,
long Channel,
long *State)

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

- **Handle –** Handle returned by OpenLabJack().
- **Channel –** The channel to read. 0-22 corresponds to FIO0-MIO2.

Outputs:

- **State –** Returns the state of the digital input. 0=False=Low and 1=True=High.

4.2.20 eDO()

An "easy" function that writes the state of one digital output. This is a simple alternative to the very flexible IOType based method normally used by this driver.

When needed, this function automatically configures the specified channel as a digital output.

Declaration:
LJ_ERROR _stdcall eDO (LJ_HANDLE Handle,
long Channel,
long State)

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

- **Handle –** Handle returned by OpenLabJack().
- **Channel –** The channel to write to. 0-22 corresponds to FIO0-MIO2.
- **State –** The state to write to the digital output. 0=False=Low and 1=True=High.

4.2.21 eTCConfig()

An “easy” function that configures and initializes all the timers and counters. This is a simple alternative to the very flexible IOType based method normally used by this driver.

When needed, this function automatically configures the needed lines as digital.

Declaration:

```
LJ_ERROR _stdcall eTCConfig (   LJ_HANDLE Handle,
                                long *aEnableTimers,
                                long *aEnableCounters,
                                long TCPinOffset,
                                long TimerClockBaseIndex,
                                long TimerClockDivisor,
                                long *aTimerModes,
                                double *aTimerValues,
                                long Reserved1,
                                long Reserved2)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **Handle –** Handle returned by OpenLabJack().
- **aEnableTimers –** An array where each element specifies whether that timer is enabled. Timers must be enabled in order starting from 0, so for instance, Timer1 cannot be enabled without enabling Timer0 also. A nonzero value for an array element specifies to enable that timer. For the U6, this array must always have at least 4 elements.
- **aEnableCounters –** An array where each element specifies whether that counter is enabled. Counters do not have to be enabled in order starting from 0, so Counter1 can be enabled when Counter0 is disabled. A nonzero value for an array element specifies to enable that counter. For the U6, this array must always have at least 2 elements.
- **TCPinOffset –** Value from 0-8 specifies where to start assigning timers and counters.
- **TimerClockBaseIndex –** Pass a constant to set the timer base clock. The default is LJ_tc48MHZ.
- **TimerClockDivisor –** Pass a divisor from 0-255 where 0 is a divisor of 256.
- **aTimerModes –** An array where each element is a constant specifying the mode for that timer. For the U6, this array must always have at least 4 elements.
- **aTimerValues –** An array where each element is specifies the initial value for that timer. For the U6, this array must always have at least 4 elements.
- **Reserved (1&2) –** Pass 0.

4.2.22 eTCValues()

An “easy” function that updates and reads all the timers and counters. This is a simple alternative to the very flexible IOType based method normally used by this driver.

Declaration:

```
LJ_ERROR _stdcall eTCValues (   LJ_HANDLE Handle,
                                long *aReadTimers,
                                long *aUpdateResetTimers,
                                long *aReadCounters,
                                long *aResetCounters,
                                double *aTimerValues,
                                double *aCounterValues,
                                long Reserved1,
                                long Reserved2)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **Handle –** Handle returned by OpenLabJack().
- **aReadTimers –** An array where each element specifies whether to read that timer. A nonzero value for an array element specifies to read that timer. For the U6, this array must always have at least 4 elements.
- **aUpdateResetTimers –** An array where each element specifies whether to update/reset that timer. A nonzero value for an array element specifies to update/reset that timer. For the U6, this array must always have at least 4 elements.
- **aReadCounters –** An array where each element specifies whether to read that counter. A nonzero value for an array element specifies to read that counter. For the U6, this array must always have at least 2 elements.
- **aResetCounters –** An array where each element specifies whether to reset that counter. A nonzero value for an array element specifies to reset that counter. For the U6, this array must always have at least 2 elements.
- **aTimerValues –** An array where each element is the new value for that timer. Each value is only updated if the appropriate element is set in the aUpdateResetTimers array. For the U6, this array must always have at least 4 elements.
- **Reserved (1&2) –** Pass 0.

Outputs:

- **aTimerValues –** An array where each element is the value read from that timer if the appropriate element is set in the aReadTimers array.
- **aCounterValues –** An array where each element is the value read from that counter if the appropriate element is set in the aReadCounters array.

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 U6 is:

```
LJ_dtU6
```

There is only one valid ConnectionType for the U6:

```
LJ_ctUSB
```

Following is example pseudocode to open a U6 over USB:

```
//Open the first found LabJack U6 over USB.
OpenLabJack (LJ_dtU6, 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

There are two IOTypes used to write or read general U6 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 0/15.
LJ_ioGET_AIN_DIFF   //Specify negative channel in x1.
```

The following are IOTypes used to configure (or read) the input range of a particular analog input channel:

```
LJ_ioPUT_AIN_RANGE
LJ_ioGET_AIN_RANGE
```

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

```
LJ_rgAUTO           // LabJackUD Default
LJ_rgBIP10V         // +/- 10V
LJ_rgBIP1V          // +/- 1V
LJ_rgBIPP1V         // +/- 0.1V
LJ_rgBIPP01V        // +/- 0.01V
```

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

```
LJ_chAIN_RESOLUTION       //0=default, 1-8=high-speed ADC, 9-13=high-res ADC
LJ_chAIN_SETTLING_TIME    //0=5us, 1=10us, 2=100us, 3=1ms, 4=10ms
LJ_chAIN_BINARY
```

Following is example pseudocode to read analog inputs:

```
//Configure all analog inputs for max resolution.  Like most
//settings, this will apply to all further measurements until
//the parameter is changed or the DLL unloaded.
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 13, 0, 0);

//Configure AIN1 for +/- 10 volt range.
AddRequest (lngHandle, LJ_ioPUT_AIN_RANGE, 1, LJ_rgBIP10V, 0, 0);

//Configure AIN2 for +/- 1 volt range.  This applies to any
//reading, single-ended or differential, where the positive
//channel is AIN2.
AddRequest (lngHandle, LJ_ioPUT_AIN_RANGE, 2, LJ_rgBIP1V, 0, 0);

//Request a single-ended read from AIN1.
AddRequest (lngHandle, LJ_ioGET_AIN, 1, 0, 0, 0);

//Request a differential read of AIN2-AIN3.
AddRequest (lngHandle, LJ_ioGET_AIN_DIFF, 2, 0, 3, 0);

//Request a single-ended read of AIN2.  Here we use the DIFF
//IOType, but pass x1=0 which does a single-ended measurement.
AddRequest (lngHandle, LJ_ioGET_AIN_DIFF, 2, 0, 0, 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 AIN1 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 AIN2-AIN3 voltage.
GetNextResult (lngHandle, 0, 0, &dblValue, 0, 0);

//Get the AIN2.
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 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
20-22 MIO0-MIO2

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:

```
//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 timer and counter 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       //0-4
LJ_chTIMER_CLOCK_BASE            //Value constants below
LJ_chTIMER_CLOCK_DIVISOR         //0-255, where 0=256
LJ_chTIMER_COUNTER_PIN_OFFSET    //0-8
```

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

```
LJ_tc4MHZ          //4 MHz clock base
LJ_tc12MHZ         //12 MHz clock base
LJ_tc48MHZ         //48 MHz clock base
LJ_tc1MHZ_DIV      //1 MHz clock base w/ divisor (no Counter0)
LJ_tc4MHZ_DIV      //4 MHz clock base w/ divisor (no Counter0)
LJ_tc12MHZ_DIV     //12 MHz clock base w/ divisor (no Counter0)
LJ_tc48MHZ_DIV     //48 MHz clock base w/ divisor (no Counter0)
LJ_tcSYS           //Equivalent to LJ_tc48MHZ
```

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

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

//Set the pin offset to 0, which causes the timers to start on FIO0.
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 0, 0, 0);

//Enable 2 timers.  They will use FIO0-FIO1
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, FIO2.
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 48MHZ_DIV, meaning that the clock divisor
//is supported and Counter0 is not available.
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tc48MHZ_DIV, 0, 0);

//Set the timer clock divisor to 48, creating a 1 MHz timer clock.
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 48, 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 simple ePut/eGet 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

The highest input data rates are obtained in stream mode. The following IOTypes are used to control streaming:

```
LJ_ioCLEAR_STREAM_CHANNELS
LJ_ioADD_STREAM_CHANNEL
LJ_ioADD_STREAM_CHANNEL_DIFF     //Put negative channel in x1.
LJ_ioSTART_STREAM                //Value returns actual scan rate.
LJ_ioSTOP_STREAM
LJ_ioGET_STREAM_DATA
```

The following constant is passed in the Channel parameter with the get stream data IOType to specify a read returning all scanned channels, rather than retrieving each scanned channel separately:

```
LJ_chALL_CHANNELS          //Used with LJ_ioGET_STREAM_DATA.
```

The following are special channels, used with the get/put config IOTypes, to write or read various stream values:

```
LJ_chSTREAM_SCAN_FREQUENCY
LJ_chSTREAM_BUFFER_SIZE          //UD driver stream buffer size in samples.
LJ_chSTREAM_WAIT_MODE
LJ_chSTREAM_DISABLE_AUTORECOVERY
LJ_chSTREAM_BACKLOG_COMM         //Read-only.  0=0% and 256=100%.
LJ_chSTREAM_BACKLOG_UD           //Read-only.  Number of samples.
LJ_chSTREAM_SAMPLES_PER_PACKET   //Default 25. Range 1-25.
LJ_chSTREAM_READS_PER_SECOND     //Default 25.
```

With the wait mode special channel above, the following constants are passed in the value parameter to select the behavior when reading data:

```
LJ_swNONE           //No wait.  Immediately return available data.
LJ_swALL_OR_NONE    //No wait.  Immediately return requested amount, or none.
LJ_swPUMP           //Advanced message pump wait mode.
LJ_swSLEEP          //Wait until requested amount available.
```

The backlog special channels return information about how much data is left in the stream buffer on the U6 or in the UD driver. These parameters are updated whenever a stream packet is read by the driver, and thus might not exactly reflect the current state of the buffers, but can be useful to detect problems.

When streaming, the processor acquires data at precise intervals, and transfers it to a buffer on the U6 itself. The U6 has a small buffer (512-984 samples) for data waiting to be transferred to the host. The *LJ_chSTREAM_BACKLOG_COMM* special channel specifies how much data is left in the U6 buffer (*COMM* or *CONTROL* are the same thing on the U6), where 0 means 0% full and 256 would mean 100% full. The UD driver retrieves stream data from the U6 in the background, but if the computer or communication link is too slow for some reason, the driver might not be able to read the data as fast as the U6 is acquiring it, and thus there will be data left over in the U6 buffer.

To obtain the maximum stream rates documented in Section 3.2, the data must be transferred between host and U6 in large chunks. The amount of data transferred per low-level packet is controlled by *LJ_chSTREAM_SAMPLES_PER_PACKET* . The driver will use the parameter *LJ_chSTREAM_READS_PER_SECOND* to determine how many low-level packets to retrieve per read.

The size of the UD stream buffer on the host is controlled by *LJ_chSTREAM_BUFFER_SIZE* . The application software on the host must read data out of the UD stream buffer fast enough to prevent overflow. After each read, use *LJ_chSTREAM_BACKLOG_UD* to determine how many samples are left in the buffer.

Since the data buffer on the U6 is fairly small a feature called auto-recovery is used. If the buffer overflows, the U6 will continue streaming but discard data until the buffer is emptied, and then data will be stored in the buffer again. The U6 keeps track of how many packets are discarded and reports that value. Based on the number of packets discarded, the UD driver adds the proper number of dummy samples (-9999.0) such that the correct timing is maintained. Auto-recovery will generally occur when the U6 buffer is 90-95% full.

In stream mode the LabJack acquires inputs at a fixed interval, controlled by the hardware clock on the device itself, and stores the data in a buffer. The LabJackUD driver automatically reads

data from the hardware buffer and stores it in a PC RAM buffer until requested. The general procedure for streaming is:

- Update configuration parameters.
- Build the scan list.
- Start the stream.
- Periodically retrieve stream data in a loop.
- Stop the stream.

Following is example pseudocode to configure a 2-channel stream.

```
//Set the scan rate.
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chSTREAM_SCAN_FREQUENCY, scanRate, 0, 0);

//Give the UD driver a 5 second buffer (scanRate * 2 channels * 5 seconds).
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chSTREAM_BUFFER_SIZE, scanRate*2*5, 0, 0);

//Configure reads to wait and retrieve the desired amount of data.
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chSTREAM_WAIT_MODE, LJ_swSLEEP, 0, 0);

//Define the scan list as singled ended AIN2 then differential AIN0-AIN1.
AddRequest (lngHandle, LJ_ioCLEAR_STREAM_CHANNELS, 0, 0, 0, 0);
AddRequest (lngHandle, LJ_ioADD_STREAM_CHANNEL, 2, 0, 0, 0);
AddRequest (lngHandle, LJ_ioADD_STREAM_CHANNEL_DIFF, 0, 0, 1, 0);

//Execute the requests.
GoOne (lngHandle);
```

Next, start the stream:

```
//Start the stream.
eGet(lngHandle, LJ_ioSTART_STREAM, 0, &dblValue, 0);

//The actual scan rate is dependent on how the desired scan rate divides into
//the LabJack clock.  The actual scan rate is returned in the value parameter
//from the start stream command.
actualScanRate = dblValue;
actualSampleRate = 2*dblValue;
```

Once a stream is started, the data must be retrieved periodically to prevent the buffer from overflowing. To retrieve data, add a request with IOType *LJ_ioGET_STREAM_DATA*. The Channel parameter should be *LJ_chALL_CHANNELS* or a specific channel number (ignored for a single channel stream). The Value parameter should be the number of scans (all channels) or samples (single channel) to retrieve. The x1 parameter should be a pointer to an array that has been initialized to a sufficient size. Keep in mind that the required number of elements if retrieving all channels is number of scans * number of channels.

Data is stored interleaved across all streaming channels. In other words, if two channels are streaming, 0 and 1, and *LJ_chALL_CHANNELS* is the channel number for the read request, the data will be returned as Channel0, Channel1, Channel0, Channel1, etc. Once the data is read it is removed from the internal buffer, and the next read will give new data.

If multiple channels are being streamed, data can be retrieved one channel at a time by passing a specific channel number in the request. In this case the data is not removed from the internal

buffer until the last channel in the scan is requested. Reading the data from the last channel (not necessarily all channels) is the trigger that causes the block of data to be removed from the buffer. This means that if three channels are streaming, 0, 1 and 2 (in that order in the scan list), and data is requested from channel 0, then channel 1, then channel 0 again, the request for channel 0 the second time will return the same data as the first request. New data will not be retrieved until after channel 2 is read, since channel 2 is last in the scan list. If the first get stream data request is for 10 samples from channel 1, the reads from channels 0 and 2 also must be for 10 samples. Note that when reading stream data one channel at a time (not using *LJ_chALL_CHANNELS*), the scan list cannot have duplicate channel numbers.

There are three basic wait modes for retrieving the data:

- LJ_swNONE: The Go call will retrieve whatever data is available at the time of the call up to the requested amount of data. A Get command should be called to determine how many scans were retrieved. This is generally used with a software timed read interval. The number of samples read per loop iteration will vary, but the time per loop iteration will be pretty consistent. Since the LabJack clock could be faster than the PC clock, it is recommended to request more scans than are expected each time so that the application does not get behind.
- LJ_swSLEEP: This makes the Go command a blocking call. The Go command will loop until the requested amount of is retrieved or no new data arrives from the device before timeout. In this mode, the hardware dictates the timing of the application. The time per loop iteration will vary, but the number of samples read per loop will be the same every time. A Get command should be called to determine whether all the data was retrieved, or a timeout condition occurred and none of the data was retrieved.
- LJ_swALL_OR_NONE: If available, the Go call will retrieve the amount of data requested, otherwise it will retrieve no data. A Get command should be called to determine whether all the data was returned or none. This could be a good mode if hardware timed execution is desirable, but without the application continuously waiting in SLEEP mode.

The following pseudocode reads data continuously in SLEEP mode as configured above:

```
//Read data until done.
while(!done)
{
	//Must set the number of scans to read each iteration, as the read
	//returns the actual number read.
	numScans = 1000;

	//Read the data.  Note that the array passed must be sized to hold
	//enough SAMPLES, and the Value passed specifies the number of SCANS
	//to read.
	eGet(lngHandle, LJ_ioGET_STREAM_DATA, LJ_chALL_CHANNELS, &numScans, array);
	actualNumberRead = numScans;

	//When all channels are retrieved in a single read, the data
	//is interleaved in a 1-dimensional array.  The following lines
	//get the first sample from each channel.
	channelA = array[0];
	channelB = array[1];

	//Retrieve the current U6 backlog.  The UD driver retrieves
	//stream data from the U6 in the background, but if the computer
	//is too slow for some reason the driver might not be able to read
	//the data as fast as the U6 is acquiring it, and thus there will
	//be data left over in the U6 buffer.
	eGet(lngHandle, LJ_ioGET_CONFIG, LJ_chSTREAM_BACKLOG_COMM, &dblCommBacklog, 0);
```

```
        //Retrieve the current UD driver backlog.  If this is growing, then
        //the application software is not pulling data from the UD driver
        //fast enough.
        eGet(lngHandle, LJ_ioGET_CONFIG, LJ_chSTREAM_BACKLOG_UD, &dblUDBacklog, 0);
}
```

Finally, stop the stream:

```
//Stop the stream.
errorcode = ePut (Handle, LJ_ioSTOP_STREAM, 0, 0, 0);
```

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, or when making OS portable code.

```
LJ_ioRAW_OUT
LJ_ioRAW_IN
```

When using these IOTypes, channel # specifies the desired communication pipe. For the U6, 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 Easy Functions

The easy functions are simple alternatives to the very flexible IOType based method normally used by this driver. There are 6 functions available:

```
eAIN()       //Read 1 analog input.
eDAC()       //Write to 1 analog output.
eDI()        //Read 1 digital input.
eDO()        //Write to 1 digital output.
eTCConfig()  //Configure all timers and counters.
eTCValues()  //Update/reset and read all timers and counters.
```

In addition to the basic operations, these functions also automatically handle configuration as needed. For example, eDO() sets the specified line to digital output if previously configured as analog and/or input, and eAIN() sets the line to analog if previously configured as digital.

The first 4 functions should not be used when speed is critical with multi-channel reads. These functions use one low-level function per operation, whereas using the normal Add/Go/Get method with IOTypes, many operations can be combined into a single low-level call. With single channel operations, however, there will be little difference between using an easy function or Add/Go/Get.

The last two functions handle almost all functionality related to timers and counters, and will usually be as efficient as any other method. These easy functions are recommended for most timer/counter applications.

Following is example pseudocode:

```
//Take a single-ended measurement from AIN3.
//eAIN (Handle, ChannelP, ChannelN, *Voltage, Range, Resolution,
//     Settling, Binary,  Reserved1, Reserved2)
//
eAIN(lngHandle, 3, 15, &dblVoltage, LJ_rgAUTO, 0, 0, 0, 0, 0);
printf("AIN3 value = %.3f\n",dblVoltage);

//Set DAC0 to 3.1 volts.
//eDAC (Handle, Channel, Voltage, Binary, Reserved1, Reserved2)
//
eDAC(lngHandle, 0, 3.1, 0, 0, 0);

//Read state of FIO2.
//eDI  (Handle, Channel, *State)
//
eDI(lngHandle, 2, &lngState);
printf("FIO2 state = %.0f\n",lngState);

//Set FIO3 to output-high.
//eDO  (Handle, Channel, State)
//
eDO(lngHandle, 3, 1);


//Enable and configure 1 output timer and 1 input timer, and enable Counter0.
//Fill the arrays with the desired values, then make the call.
alngEnableTimers = {1,1,0,0}; //Enable Timer0-Timer1
alngTimerModes = {LJ_tmPWM8,LJ_tmRISINGEDGES32,0,0}; //Set timer modes
adblTimerValues = {16384,0,0,0}; //Set PWM8 duty-cycle to 75%.
alngEnableCounters = {1,0}; //Enable Counter0
//
//eTCConfig  (Handle, *aEnableTimers, *aEnableCounters, TCPinOffset,
//            TimerClockBaseIndex, TimerClockDivisor, *aTimerModes,
//            *aTimerValues, Reserved1, Reserved2);
//
eTCConfig(lngHandle, alngEnableTimers, alngEnableCounters, 4, LJ_tc48MHZ, 0,
alngTimerModes, adblTimerValues, 0, 0);


//Read and reset the input timer (Timer1), read and reset Counter0, and update
//the value (duty-cycle) of the output timer (Timer0).
//Fill the arrays with the desired values, then make the call.
alngReadTimers = {0,1,0,0}; //Read Timer1
alngUpdateResetTimers = {1,1,0,0}; //Update Timer0 and reset Timer1
alngReadCounters = {1,0}; //Read Counter0
alngResetCounters = {1,0}; //Reset Counter0
adblTimerValues = {32768,0,0,0}; //Change Timer0 duty-cycle to 50%
//
//eTCValues  (Handle, *aReadTimers, *aUpdateResetTimers, *aReadCounters,
//            *aResetCounters, *aTimerValues, *aCounterValues, Reserved1,
//            Reserved2);
//
```

```
eTCValues(lngHandle, alngReadTimers, alngUpdateResetTimers, alngReadCounters,
alngResetCounters, adblTimerValues, adblCounterValues, 0, 0);
printf("Timer1 value = %.0f\n",adblTimerValues[1]);
printf("Counter0 value = %.0f\n",adblCounterValues[0]);
```

4.3.10 SPI Serial Communication

The U6 supports Serial Peripheral Interface (SPI) communication as the master only. SPI is a synchronous serial protocol typically used to communicate with chips that support SPI as slave devices.

This serial link is not an alternative to the USB connection. Rather, the host application will write/read data to/from the U6 over USB, and the U6 communicates with some other device using the serial protocol. Using this serial protocol is considered an advanced topic. A good knowledge of the protocol is recommended, and a logic analyzer or oscilloscope might be needed for troubleshooting.

There is one IOType used to write/read data over the SPI bus:

```
LJ_ioSPI_COMMUNICATION     // Value= number of bytes (1-50). x1= array.
```

The following are special channels, used with the get/put config IOTypes, to configure various parameters related to the SPI bus. See the low-level function description in Section 5.2.17 for more information about these parameters:

```
LJ_chSPI_AUTO_CS
LJ_chSPI_DISABLE_DIR_CONFIG
LJ_chSPI_MODE
LJ_chSPI_CLOCK_FACTOR
LJ_chSPI_MOSI_PIN_NUM
LJ_chSPI_MISO_PIN_NUM
LJ_chSPI_CLK_PIN_NUM
LJ_chSPI_CS_PIN_NUM
```

Following is example pseudocode to configure SPI communication:

```
//First, configure the SPI communication.

//Enable automatic chip-select control.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSPI_AUTO_CS,1,0,0);

//Do not disable automatic digital i/o direction configuration.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSPI_DISABLE_DIR_CONFIG,0,0,0);

//Mode A:  CPHA=1, CPOL=1.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSPI_MODE,0,0,0);

//Maximum clock rate (~100kHz).
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSPI_CLOCK_FACTOR,0,0,0);

//Set MOSI to FIO2.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSPI_MOSI_PIN_NUM,2,0,0);

//Set MISO to FIO3.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSPI_MISO_PIN_NUM,3,0,0);

//Set CLK to FIO0.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSPI_CLK_PIN_NUM,0,0,0);

//Set CS to FIO1.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSPI_CS_PIN_NUM,1,0,0);
```

```
//Execute the configuration requests.
GoOne(lngHandle);
```

Following is pseudocode to do the actual SPI communication:

```
//Transfer the data.
eGet(lngHandle, LJ_ioSPI_COMMUNICATION, 0, &numBytesToTransfer, array);
```

4.3.11 I²C Serial Communication

The U6 supports Inter-Integrated Circuit (I²C or I2C) communication as the master only. I²C is a synchronous serial protocol typically used to communicate with chips that support I²C as slave devices. Any 2 digital I/O lines are used for SDA and SCL. Note that the I²C bus generally requires pull-up resistors of perhaps 4.7 kΩ from SDA to Vs and SCL to Vs, and also note that the screw terminals labeled SDA and SCL (if present) are not used for I²C.

This serial link is not an alternative to the USB connection. Rather, the host application will write/read data to/from the U6 over USB, and the U6 communicates with some other device using the serial protocol. Using this serial protocol is considered an advanced topic. A good knowledge of the protocol is recommended, and a logic analyzer or oscilloscope might be needed for troubleshooting.

There is one IOType used to write/read I²C data:

```
LJ_ioI2C_COMMUNICATION
```

The following are special channels used with the I²C IOType above:

```
LJ_chI2C_READ       // Value= number of bytes (0-52). x1= array.
LJ_chI2C_WRITE      // Value= number of bytes (0-50). x1= array.
LJ_chI2C_GET_ACKS
```

The following are special channels, used with the get/put config IOTypes, to configure various parameters related to the I²C bus. See the low-level function description in Section 5.2.21 for more information about these parameters:

```
LJ_chI2C_ADDRESS_BYTE
LJ_chI2C_SCL_PIN_NUM      // 0-19. Pull-up resistor usually required.
LJ_chI2C_SDA_PIN_NUM      // 0-19. Pull-up resistor usually required.
LJ_chI2C_OPTIONS
LJ_chI2C_SPEED_ADJUST
```

The LJTick-DAC is an accessory from LabJack with an I²C 24C01C EEPROM chip. Following is example pseudocode to configure I²C to talk to that chip:

```
//The AddressByte of the EEPROM on the LJTick-DAC is 0xA0 or decimal 160.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chI2C_ADDRESS_BYTE,160,0,0);

//SCL is FIO0
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chI2C_SCL_PIN_NUM,0,0,0);

//SDA is FIO1
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chI2C_SDA_PIN_NUM,1,0,0);

//See description of low-level I2C function.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chI2C_OPTIONS,0,0,0);
```

```
//See description of low-level I2C function.  0 is max speed of about 150 kHz.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chI2C_SPEED_ADJUST,0,0,0);

//Execute the configuration requests.
GoOne(lngHandle);
```

Following is pseudocode to read 4 bytes from the EEPROM:

```
//Initial read of EEPROM bytes 0-3 in the user memory area.
//We need a single I2C transmission that writes the address and then reads
//the data.  That is, there needs to be an ack after writing the address,
//not a stop condition.  To accomplish this, we use Add/Go/Get to combine
//the write and read into a single low-level call.
numWrite = 1;
array[0] = 0;  //Memory address.  User area is 0-63.
AddRequest(lngHandle, LJ_ioI2C_COMMUNICATION, LJ_chI2C_WRITE, numWrite, array, 0);

numRead = 4;
AddRequest(lngHandle, LJ_ioI2C_COMMUNICATION, LJ_chI2C_READ, numRead, array, 0);

//Execute the requests.
GoOne(lngHandle);
```

For more example code, see the I2C.cpp example in the VC6_LJUD archive.

4.3.12 Asynchronous Serial Communication

The U6 has a UART available that supports asynchronous serial communication. The TX (transmit) and RX (receive) lines appear on FIO/EIO after any timers and counters, so with no timers/counters enabled, and pin offset set to 0, TX=FIO0 and RX=FIO1.

Communication is in the common 8/n/1 format. Similar to RS232, except that the logic is normal CMOS/TTL. Connection to an RS232 device will require a converter chip such as the MAX233, which inverts the logic and shifts the voltage levels.

This serial link is not an alternative to the USB connection. Rather, the host application will write/read data to/from the U6 over USB, and the U6 communicates with some other device using the serial protocol. Using this serial protocol is considered an advanced topic. A good knowledge of the protocol is recommended, and a logic analyzer or oscilloscope might be needed for troubleshooting.

There is one IOType used to write/read asynchronous data:

```
LJ_ioASYNCH_COMMUNICATION
```

The following are special channels used with the asynch IOType above:

```
LJ_chASYNCH_ENABLE  // Enables UART to begin buffering rx data.
LJ_chASYNCH_RX      // Value= returns pre-read buffer size. x1= array.
LJ_chASYNCH_TX      // Value= number to send (0-56), number in rx buffer. x1= array.
LJ_chASYNCH_FLUSH   // Flushes the rx buffer.  All data discarded. Value ignored.
```

When using *LJ_chASYNCH_RX*, the Value parameter returns the size of the Asynch buffer before the read. If the size is 32 bytes or less, that is how many bytes were read. If the size is more than 32 bytes, then the call read 32 this time and there are still bytes left in the buffer.

When using *LJ_chASYNCH_TX*, specify the number of bytes to send in the Value parameter. The Value parameter returns the size of the Asynch read buffer.

The following is a special channel, used with the get/put config IOTypes, to specify the baud rate for the asynchronous communication:

```
LJ_chASYNCH_BAUDFACTOR    // 16-bit value for hardware V1.30.  8-bit for V1.21.
```

With hardware revision 1.30 this is a 16-bit value that sets the baud rate according the following formula: BaudFactor16 = 2^16 - 48000000/(2 * Desired Baud). For example, a BaudFactor16 = 63036 provides a baud rate of 9600 bps. With hardware revision 1.21, the value is only 8-bit and the formula is BaudFactor8 = 2^8 – TimerClockBase/(Desired Baud).

Following is example pseudocode for asynchronous communication:

```
//Set data rate for 9600 bps communication.
ePut(lngHandle, LJ_ioPUT_CONFIG, LJ_chASYNCH_BAUDFACTOR, 63036, 0);

//Enable UART.
ePut(lngHandle, LJ_ioASYNCH_COMMUNICATION, LJ_chASYNCH_ENABLE, 1, 0);

//Write data.
eGet(lngHandle, LJ_ioASYNCH_COMMUNICATION, LJ_chASYNCH_TX, &numBytes, array);

//Read data.  Always initialize array to 32 bytes.
eGet(lngHandle, LJ_ioASYNCH_COMMUNICATION, LJ_chASYNCH_RX, &numBytes, array);
```

4.3.13 Watchdog Timer

The U6 has firmware based watchdog capability. Unattended systems requiring maximum up-time might use this capability to reset the U6 or the entire system. When any of the options are enabled, an internal timer is enabled which resets on any incoming USB communication. If this timer reaches the defined TimeoutPeriod before being reset, the specified actions will occur. Note that while streaming, data is only going out, so some other command will have to be called periodically to reset the watchdog timer.

Timeout of the watchdog on the U6 can be specified to cause a device reset, update the state of 1 digital I/O (must be configured as output by user), or both.

Typical usage of the watchdog is to configure the reset defaults as desired, and then use the watchdog simply to reset the device on timeout.

Note that some USB hubs do not like to have any USB device repeatedly reset. With such hubs, the operating system will quit reenumerating the device on reset and the computer will have to be rebooted, so avoid excessive resets with hubs that seem to have this problem.

If the watchdog is accidentally configured to reset the processor with a very low timeout period (such as 1 second), it could be difficult to establish any communication with the device. In such a case, the reset-to-default jumper can be used to turn off the watchdog. Power up the U6 with a short from FIO2 to SPC, then remove the jumper and power cycle the device again. This resets all power-up settings to factory default values.

There is one IOType used to configure and control the watchdog:

```
LJ_ioSWDT_CONFIG    // Channel is enable or disable constant.
```

The watchdog settings are stored in non-volatile flash memory (and reloaded at reset), so every request with this IOType causes a flash erase/write. The flash has a rated endurance of at least 20000 writes, which is plenty for reasonable operation, but if this IOType is called in a high-speed loop the flash could be damaged.

The following are special channels used with the watchdog config IOType above:

```
LJ_chSWDT_ENABLE    // Value is timeout in seconds (1-65535).
LJ_chSWDT_DISABLE
```

The following are special channels, used with the put config IOType, to configure watchdog options. These parameters cause settings to be updated in the driver only. The settings are not actually sent to the hardware until the *LJ_ioSWDT_CONFIG* IOType (above) is used:

```
LJ_chSWDT_RESET_DEVICE
LJ_chSWDT_UDPATE_DIOA
LJ_chSWDT_DIOA_CHANNEL
LJ_chSWDT_DIOA_STATE
```

Following is example pseudocode to configure and enable the watchdog:

```
//Initialize EIO2 to output-low, which also forces the direction to output.
//It would probably be better to do this by configuring the power-up defaults.
AddRequest(lngHandle, LJ_ioPUT_DIGITAL_BIT, 10,0,0,0);

//Specify that the device should be reset on timeout.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSWDT_RESET_DEVICE,1,0,0);

//Specify that the state of the digital line should be updated on timeout.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSWDT_UDPATE_DIOA,1,0,0);

//Specify that EIO2 is the desired digital line.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSWDT_DIOA_CHANNEL,10,0,0);

//Specify that the digital line should be set high.
AddRequest(lngHandle, LJ_ioPUT_CONFIG, LJ_chSWDT_DIOA_STATE,1,0,0);

//Enable the watchdog with a 60 second timeout.
AddRequest(lngHandle, LJ_ioSWDT_CONFIG, LJ_chSWDT_ENABLE,60,0,0);

//Execute the requests.
GoOne(lngHandle);
```

Following is pseudocode to disable the watchdog:

```
//Disable the watchdog.
ePut(lngHandle, LJ_ioSWDT_CONFIG, LJ_chSWDT_DISABLE,0,0);
```

4.3.14 Miscellaneous

The following are special channels, used with the get/put config IOTypes, to read/write the calibration memory and user memory:

```
LJ_chCAL_CONSTANTS
LJ_chUSER_MEM
```

For more information, see the low-level descriptions in Sections 5.2.6-5.2.8, and see the Memory example in the VC6_LJUD archive.

The following wait IOType is used to create a delay between other actions:

```
LJ_ioPUT_WAIT       // Channel ignored. Value = 0-8388480 microseconds. Actual
resolution is 128 microseconds.
```

Any value (in microseconds) from 0-8388480 can be passed, but the actual resolution is 128 microseconds.

This is typically used to put a small delay between two actions that will execute in the same low-level Feedback command. It is useful when the desired delay is less than what can be accomplished through software.

For example, a 1.024 millisecond pulse can be created by executing a single Add/Go/Get block that sequentially requests to set FIO4 to output-high, wait 1024 microseconds, then set FIO4 to output-low.

4.4 Errorcodes

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

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	
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	Overrun of the UD stream buffer.
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	Stream packet received out of sequence.
16	LJE_AD_PIN_CONFIGURATION_ERROR	Analog request on a digital pin, or vice versa.
17	LJE_REQUEST_NOT_PROCESSED	Previous request had an error.
19	LJE_SCRATCH_ERROR	
20	LJE_DATA_BUFFER_OVERFLOW	
21	LJE_ADC0_BUFFER_OVERFLOW	
22	LJE_FUNCTION_INVALID	
23	LJE_SWDT_TIME_INVALID	
24	LJE_FLASH_ERROR	
25	LJE_STREAM_IS_ACTIVE	
26	LJE_STREAM_TABLE_INVALID	
27	LJE_STREAM_CONFIG_INVALID	
28	LJE_STREAM_BAD_TRIGGER_SOURCE	
30	LJE_STREAM_INVALID_TRIGGER	
31	LJE_STREAM_ADC0_BUFFER_OVERFLOW	
33	LJE_STREAM_SAMPLE_NUM_INVALID	
34	LJE_STREAM_BIPOLAR_GAIN_INVALID	
35	LJE_STREAM_SCAN_RATE_INVALID	

Table 4-1. Request Level Errorcodes (Part 1)

Errorcode	Name	Description
36	LJE_TIMER_INVALID_MODE	
37	LJE_TIMER_QUADRATURE_AB_ERROR	
38	LJE_TIMER_QUAD_PULSE_SEQUENCE	
39	LJE_TIMER_BAD_CLOCK_SOURCE	
40	LJE_TIMER_STREAM_ACTIVE	
41	LJE_TIMER_PWMSTOP_MODULE_ERROR	
42	LJE_TIMER_SEQUENCE_ERROR	
43	LJE_TIMER_SHARING_ERROR	
44	LJE_TIMER_LINE_SEQUENCE_ERROR	
45	LJE_EXT_OSC_NOT_STABLE	
46	LJE_INVALID_POWER_SETTING	
47	LJE_PLL_NOT_LOCKED	
48	LJE_INVALID_PIN	
49	LJE_IOTYPE_SYNCH_ERROR	
50	LJE_INVALID_OFFSET	
51	LJE_FEEDBACK_IOTYPE_NOT_VALID	
52	LJE_SHT_CRC	
53	LJE_SHT_MEASREADY	
54	LJE_SHT_ACK	
55	LJE_SHT_SERIAL_RESET	
56	LJE_SHT_COMMUNICATION	
57	LJE_AIN_WHILE_STREAMING	AIN not available to command/response functions while the UE9 is streaming.
58	LJE_STREAM_TIMEOUT	
60	LJE_STREAM_SCAN_OVERLAP	New scan started before the previous scan completed. Scan rate is too high.
61	LJE_FIRMWARE_VERSION_IOTYPE	IOType not supported with this firmware.
62	LJE_FIRMWARE_VERSION_CHANNEL	Channel not supported with this firmware.
63	LJE_FIRMWARE_VERSION_VALUE	Value not supported with this firmware.
64	LJE_HARDWARE_VERSION_IOTYPE	IOType not supported with this hardware.
65	LJE_HARDWARE_VERSION_CHANNEL	Channel not supported with this hardware.
66	LJE_HARDWARE_VERSION_VALUE	Value not supported with this hardware.
67	LJE_CANT_CONFIGURE_PIN_FOR_ANALOG	
68	LJE_CANT_CONFIGURE_PIN_FOR_DIGITAL	
70	LJE_TC_PIN_OFFSET_MUST_BE_4_TO_8	

Table 4-2. Request Level Errorcodes (Part 2)

Errorcode	Name	Description
1000	LJE_MIN_GROUP_ERROR	Errors above this number stop all requests.
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	GetResult() 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	
1013	LJE_INVALID_CONNECTION_TYPE	
1014	LJE_INVALID_MODE	

Table 4-3. Group Level Errorcodes

The first two tables list errors which are specific to a request. For example, LJE_INVALID_CHANNEL_NUMBER. If this error occurs, other requests are not affected. The last table lists errors which cause all pending requests for a particular Go() to fail with the same error. If this type of error is received the state of any of the request is not known. For example, if requests are executed with a single Go() to set the AIN range and read an AIN, and the read fails with an LJE_COMM_FAILURE, it is not known whether the AIN range was set to the new value or whether it is still set at the old value.

5. Low-Level Function Reference

This section describes the low level functions of the U6. These are commands sent over USB directly to the processor on the U6.

The majority of Windows users will use the high-level UD driver rather than these low-level functions.

5.1 General Protocol

Following is a description of the general U6 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". Both the 8-bit and 16-bit checksum are unsigned. Sum all applicable bytes in an accumulator, 1 at a time. Each time another byte is added, 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.

Destination bit:

This bit specifies whether the command is destined for the local or remote target. This bit is ignored on the U6.

Multi-byte parameters:

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).

Masks:

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.

Binary Encoded Parameters:

Many parameters in the following functions use 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 function ControlConfig, the parameter FIODir is a single byte (8 bits) that writes/reads the direction of each of the 8 FIO lines:

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

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 ConfigU6

Writes the Local ID, and reads some hardware information. The old U6 version of this function used to write and read power-up defaults for many parameters, but that functionality has been moved to new functions.

If WriteMask is nonzero, some or all default values are written to flash. The U6 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.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x0A
3	0x08
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	WriteMask0 Bit 3: LocalID
7	Reserved
8	LocalID
9-25	Reserved

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
22-36	Reserved
37	VersionInfo

- 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.
- LocalID: If the WriteMask bit 3 is set, the value passed become the default value, meaning it is 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.

- FirmwareVersion: Fixed parameter specifies the version number of the main firmware. A firmware upgrade will generally cause this parameter to change. The lower byte is the integer portion of the version and the higher byte is the fractional portion of the version.
- BootloaderVersion: Fixed parameter specifies the version number of the bootloader. The lower byte is the integer portion of the version and the higher byte is the fractional portion of the version.
- HardwareVersion: Fixed parameter specifies the version number of the hardware. The lower byte is the integer portion of the version and the higher byte is the fractional portion of the version.
- SerialNumber: Fixed parameter that is unique for every LabJack.
- ProductID: (6) Fixed parameter identifies this LabJack as a U6.
- VersionInfo: Bit 0 specifies U3B. Bit 1 specifies U3C and if set then bit 4 specifies -HV version. Bit 2 species U6 and bit 3 specifies U6-Pro.

5.2.3 ConfigIO

Writes and reads the current IO configuration.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x05
3	0x0B
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	WriteMask Bit 0: TimerCounterConfig
7	NumberTimersEnabled
8	CounterEnable Bit 1: Enable Counter1 Bit 0: Enable Counter0
9	TimerCounterPinOffset
10-15	Reserved

Response:

Byte	
0	Checksum8
1	0xF8
2	0x05
3	0x0B
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	Reserved
8	NumberTimersEnabled
9	CounterEnable
10	TCPinOffset
10-15	Reserved

- WriteMask: Has a bit that determines if new timer/counter settings are written.
- NumberTimersEnabled: 0-4. Used to enable/disable timers. Timers will be assigned to IO pins starting with FIO0 plus TimerCounterPinOffset. Timer0 takes the first IO pin, then Timer1, and so on. 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.
- TimerCounterPinOffset: 0-8. Timers/counters are assigned terminals starting from here.

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: 4 MHz b001: 12 MHz b010: 48 MHz (Default) b011: 1 MHz /Divisor b100: 4 MHz /Divisor b101: 12 MHz /Divisor b110: 48 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 5.1 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 PortStateRead (d26) is passed, byte 2 would be equal to 1 for the command and 3 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
AIN24	2	4	3
AIN24AR	3	4	5
WaitShort	5	2	0
WaitLong	6	2	0
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 (8-bit)	34	2	0
DAC1 (8-bit)	35	2	0
DAC0 (16-bit)	38	3	0
DAC1 (16-bit)	39	3	0
Timer0	42	4	4
Timer0Config	43	4	0
Timer1	44	4	4
Timer1Config	45	4	0
Timer2	46	4	4
Timer2Config	47	4	0
Timer3	48	4	4
Timer3Config	49	4	0
Counter0	54	2	4
Counter1	55	2	4

5.2.5.1 AIN: IOType=1

AIN, 3 Command Bytes:

0	IOType=1
1	PositiveChannel
2	Reserved

2 Response Bytes:

0	AIN LSB
1	AIN MSB

This is the U3 style IOType to get a single analog input reading. It is available for compatibility, but to make full use of the U6 the new AIN24 IOType should be used.

- PositiveChannel: 0-143 for AIN0-AIN143. 14 is the internal temperature sensor and 15 is internal GND.
- NegativeChannel: 0, 15, or 199 signifies a single-ended reading. For differential readings this should be positive channel plus one, where the positive channel is an even number from 0-142.
- AIN LSB & MSB: Analog input reading is returned as a 16-bit value (always unsigned).

5.2.5.2 AIN24: IOType=2

AIN24, 4 Command Bytes:

0	IOType=2
1	PositiveChannel
2	Bits 0-3: ResolutionIndex Bits 4-7: GainIndex
3	Bits 0-2: SettlingFactor Bit 7: Differential

3 Response Bytes:

0	AIN LSB
1	AIN Bits 8-15
2	AIN MSB

This IOType returns a single analog input reading. If using the autorange feature, the AIN24AR IOType in the following Section should be used instead.

- PositiveChannel: 0-143 for AIN0-AIN143. 14 is the internal temperature sensor and 15 is internal GND.
- ResolutionIndex: 0=default, 1-8 for high-speed ADC, 9-13 for high-res ADC on U6-Pro.
- GainIndex: 0=x1, 1=x10, 2=x100, 3=x1000, 15=autorange.
- SettlingFactor: 0=5us, 1=10us, 2=100us, 3=1ms, 4=10ms.
- Differential: If this bit is set, a differential reading is done where the negative channel is PositiveChannel+1.
- AIN bytes: Analog input reading is returned as a 24-bit value (always unsigned).

5.2.5.3 AIN24AR: IOType=3

AIN24AR, 4 Command Bytes:

0	IOType=3
1	PositiveChannel
2	Bits 0-3: ResolutionIndex Bits 4-7: GainIndex
3	Bits 0-2: SettlingFactor Bit 7: Differential

5 Response Bytes:

0	AIN LSB
1	AIN Bits 8-15
2	AIN MSB
3	Bits 0-3: ResolutionIndex Bits 4-7: GainIndex
4	Status

This IOType returns a single analog input reading. Also returns the actual resolution and gain used for the reading.

- PositiveChannel: 0-143 for AIN0-AIN143. 14 is the internal temperature sensor and 15 is internal GND.
- ResolutionIndex: 0=default, 1-8 for high-speed ADC, 9-13 for high-res ADC on U6-Pro. Value in the response is the actual resolution setting used for the reading.
- GainIndex: 0=x1, 1=x10, 2=x100, 3=x1000, 15=autorange. Value in the response is the actual gain setting used for the reading.
- SettlingFactor: 0=5us, 1=10us, 2=100us, 3=1ms, 4=10ms.
- Differential: If this bit is set, a differential reading is done where the negative channel is PositiveChannel+1.
- AIN bytes: Analog input reading is returned as a 24-bit value (always unsigned).
- Status: Reserved for future use.

5.2.5.4 WaitShort: IOType=5

WaitShort, 2 Command Bytes:

0	IOType=5
1	Time (*128 us)

0 Response Bytes:

This IOType provides a way to add a delay during execution of the Feedback function. The typical use would be putting this IOType in between IOTypes that set a digital output line high and low, thus providing a simple way to create a pulse. Note that this IOType uses the same internal timer as stream mode, so cannot be used while streaming.

- Time: This value (0-255) is multiplied by 128 microseconds to determine the delay.

5.2.5.5 WaitLong: IOType=6

WaitLong, 2 Command Bytes:
0 IOType=6
1 Time (*32 ms)

0 Response Bytes:

This IOType provides a way to add a delay during execution of the Feedback function. The typical use would be putting this IOType in between IOTypes that set a digital output line high and low, thus providing a simple way to create a pulse. Note that this IOType uses the same internal timer as stream mode, so cannot be used while streaming.

- Time: This value (0-255) is multiplied by 32 milliseconds to determine the delay.

5.2.5.6 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.

5.2.5.7 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.

5.2.5.8 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. The direction of the specified line is forced to output.

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

5.2.5.9 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.

5.2.5.10 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.

5.2.5.11 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 (b00000001000001110000111), State=d67335.

5.2.5.12 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. The direction of the selected lines is forced to 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 (b00000001000001110000111), State=d67335.

5.2.5.13 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 (b00000001000001110000111), Direction=d67335.

5.2.5.14 PortDirWrite: IOType=29

PortDirWrite, 7 Command Bytes:

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

0 Response Bytes:

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 (b00000001000011100000111), Direction=d67335.

5.2.5.15 DAC# (8-bit): IOType=34,35

DAC# (8-bit), 2 Command Bytes:

0	IOType=34,35
1	Value

0 Response Bytes:

This IOType controls a single analog output.

- Value: 0=Minimum, 255=Maximum.

5.2.5.16 DAC# (16-bit): IOType=38,39

DAC# (16-bit), 3 Command Bytes:

0	IOType=38,39
1	Value LSB
2	Value MSB

0 Response Bytes:

This IOType controls a single analog output.

- Value: 0=Minimum, 65535=Maximum.

5.2.5.17 Timer#: IOType=42,44,46,48

Timer#, 4 Command Bytes:

0	IOType=42,44,46,48
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).

5.2.5.18 Timer#Config: IOType=43,45,47,49

Timer#Config, 4 Command Bytes:

0	IOType=43,45,47,49
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: The meaning of this parameter varies with the timer mode.

5.2.5.19 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: Setting this bit resets the counter to 0 after reading.
- Counter: Returns the current count from the counter if enabled. This is the value before reset (if reset was done).

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 U6 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 SetDefaults (SetToFactoryDefaults)

Executing this function causes the current or last used values (or the factory defaults) to be stored in flash as the power-up defaults.

The U6 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 the flash could eventually be damaged.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x01
3	0x0E
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	0xBA (0x82)
7	0x26 (0xC7)

Response:

Byte	
0	Checksum8
1	0xF8
2	0x01
3	0x0E
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	0x00

5.2.10 ReadDefaults

Reads the power-up defaults from flash.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x01
3	0x0E
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	0x00
7	BlockNum (0-7)

Response:

Byte	
0	Checksum8
1	0xF8
2	0x11
3	0x0E
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	0x00
8-39	Data

Power-Up Defaults Address Map

Block #	Byte #	Name	Factory
0	4	FIODirection	0
0	5	FIOState	0
0	8	EIODirection	0
0	9	EIOState	0
0	12	CIODirection	0
0	13	CIOState	0

… YTBD

5.2.11 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

5.2.12 StreamConfig

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 + 4
3	0x11
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	NumChannels
7	ResolutionIndex
8	SamplesPerPacket (1-25)
9	Reserved
10	SettlingFactor
11	ScanConfig Bit 3: Internal stream clock frequency. b0: 4 MHz b1: 48 MHz Bit 1: Divide Clock by 256
12-13	Scan Interval (1-65535)
14	ChannelNumber (Positive)
15	ChannelOptions Bit 7: Differential Bits 5-4: GainIndex

Repeat 14-15 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 (1-25).
- SamplesPerPacket: Specifies how many samples will be pulled out of the U6 FIFO buffer and returned per data read packet. For faster stream speeds, 25 samples per packet are required for data transfer efficiency. A small number of samples per packet would be desirable for low-latency data retrieval. Note that this parameter is not necessarily the same as the number of channels per scan. Even if only 1 channel is

being scanned, SamplesPerPacket will usually be set to 25, so there are usually multiple scans per packet.

- ScanConfig: Has bits to specify the stream clock.
- ScanInterval: (1-65535) This value divided by the clock frequency defined in the ScanConfig parameter, gives the interval (in seconds) between scans.
- ChannelNumber: This is the positive channel number. 0-143 for analog input channels or 193-224 for digital/timer/counter channels.
- ChannelOptions: If bit 7 is set, a differential reading is done rather than single-ended. Bits 4-5 specify the gain:

	GainIndex	Gain	Max V	Min V
Bipolar	b00 (d0)	1	10.1	-10.6
Bipolar	b01 (d1)	10	1.01	-1.06
Bipolar	b10 (d2)	100	0.101	-0.106
Bipolar	b11 (d3)	1000	0.0101	-0.0106

5.2.13 StreamStart

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

5.2.14 StreamData

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
62 (max)	Backlog
63 (max)	0x00

- SamplesPerPacket: From StreamConfig function.
- TimeStamp: Not currently implemented during normal operation, but after auto-recovery this reports the number of packets missed (1-65535).
- 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. The 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. The value ranges from 0-255, where 256 would equal 100% full.

Stream mode on the U6 uses a feature called auto-recovery. If the stream buffer gets too full, the U6 will go into auto-recovery mode. In this mode, the U6 no longer stores new scans in the buffer, but rather new scans are discarded. Data already in the buffer will be sent until the buffer contains less samples than SamplesPerPacket, and every StreamData packet will have errorcode 59. Once the stream buffer contains less samples than SamplesPerPacket, the U6 will start to buffer new scans again. The next packet returned will have errorcode 60. This packet will have 1 dummy scan where each sample is 0xFFFF, and this scan separates new data from any pre auto-recovery data. Note that the dummy scan could be at the beginning, middle, or end of this packet, and can even extend to following packets. Also, the TimeStamp parameter in this packet contains the number of scans that were discarded, allowing correct time to be calculated. The dummy scan counts as one of the missing scans included in the TimeStamp value.

5.2.15 StreamStop

Command:

Byte	
0	0xB0
1	0xB0

Response:

Byte	
0	Checksum8
1	0xB1
2	Errorcode
3	0x00

5.2.16 Watchdog

Controls a firmware based watchdog timer. Unattended systems requiring maximum up-time might use this capability to reset the U6 or the entire system. When any of the options are enabled, an internal timer is enabled which resets on any incoming USB communication. If this timer reaches the defined TimeoutPeriod before being reset, the specified actions will occur. Note that while streaming, data is only going out, so some other command will have to be called periodically to reset the watchdog timer.

If the watchdog is accidentally configured to reset the processor with a very low timeout period (such as 1 second), it could be difficult to establish any communication with the device. In such a case, the reset-to-default jumper can be used to turn off the watchdog (sets bytes 7-10 to 0). Power up the U6 with a short from FIO2 to SPC (or VSPC), then remove the jumper and power cycle the device again. This also affects the parameters in the DefaultConfig?? function.

The watchdog settings (bytes 7-10) are stored in non-volatile flash memory, so every call to this function where settings are written causes a flash erase/write. The 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 the flash could be damaged.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x05
3	0x09
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	WriteMask Bit 0: Write
7	WatchdogOptions Bit 5: Reset on Timeout Bit 4: Set DIO State on Timeout
8-9	TimeoutPeriod
10	DIOConfig Bit 7: State Bit 0-4: DIO#
11	Reserved
12	Reserved
13	Reserved
14	Reserved
15	Reserved

Response:

Byte	
0	Checksum8
1	0xF8
2	0x05
3	0x09
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	WatchdogOptions
8-9	TimeoutPeriod
10	DIOConfig
11	Reserved
12	Reserved
13	Reserved
14	Reserved
15	Reserved

- WatchdogOptions: The watchdog is enabled when this byte is nonzero. Set the appropriate bits to reset the device and/or update the state of 1 digital output.
- TimeoutPeriod: The watchdog timer is reset to zero on any incoming USB communication. Note that most functions consist of a write and read, but StreamData is outgoing only and does not reset the watchdog. If the watchdog timer is not reset before it counts up to TimeoutPeriod, the actions specified by WatchdogOptions will occur. The watchdog timer has a clock rate of about 1 Hz, so a TimeoutPeriod range of 1-65535 corresponds to about 1 to 65535 seconds.
- DIOConfig: Determines which digital I/O is affected by the watchdog, and the state it is set to. The specified DIO must have previously been configured for output. DIO# is a value from 0-19 according to the following:

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

5.2.17 SPI

Sends and receives serial data using SPI synchronous communication.

Command:

Byte	
0	Checksum8
1	0xF8
2	4 + NumSPIWords
3	0x3A
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	SPIOptions Bit 7: AutoCS Bit 6: DisableDirConfig Bits 1-0: SPIMode (0=A, 1=B, 2=C, 3=D)
7	SPIClockFactor
8	Reserved
9	CSPinNum
10	CLKPinNum
11	MISOPinNum
12	MOSIPinNum
13	NumSPIBytesToTransfer
14	SPIByte0
…	…

Response:

Byte	
0	Checksum8
1	0xF8
2	1 + NumSPIWords
3	0x3A
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	NumSPIBytesTransferred
8	SPIByte0
…	…

- NumSPIWords: This is the number of SPI bytes divided by 2. If the number of SPI bytes is odd, round up and add an extra zero to the packet.
- SPIOptions: If AutoCS is true, the CS line is automatically driven low during the SPI communication and brought back high when done. If DisableDirConfig is true, this function does not set the direction of the lines, whereas if it is false the lines are configured as CS=output, CLK=output, MISO=input, and MOSI=output. SPIMode specifies the standard SPI mode as discussed below.
- SPIClockFactor: Sets the frequency of the SPI clock according the following approximate formula: Frequency = 1000000/(10+10*(256-SPIClockFactor), where passing a value of 0 corresponds to a factor of 256, and thus a maximum frequency of about 100 kHz.

- CS/CLK/MISO/MOSI -PinNum: Assigns which digital I/O line is used for each SPI line. Value passed is 0-19 corresponding to the normal digital I/O numbers as specified in Section 2.8.
- NumSPIBytesToTransfer: Specifies how many SPI bytes will be transferred (1-50).

The initial state of SCK is set properly (CPOL), by this function, before CS (chip select) is brought low (final state is also set properly before CS is brought high again). If CS is being handled manually, outside of this function, care must be taken to make sure SCK is initially set to CPOL before asserting CS.

All standard SPI modes supported (A, B, C, and D).

Mode A: CPHA=1, CPOL=1
Mode B: CPHA=1, CPOL=0
Mode C: CPHA=0, CPOL=1
Mode D: CPHA=0, CPOL=0

If Clock Phase (CPHA) is 1, data is valid on the edge going to CPOL. If CPHA is 0, data is valid on the edge going away from CPOL. Clock Polarity (CPOL) determines the idle state of SCK.

Up to 50 bytes can be written/read. Communication is full duplex so 1 byte is read at the same time each byte is written.

5.2.18 AsynchConfig

Configures the U6 UART for asynchronous communication. The TX (transmit) and RX (receive) lines appear on FIO/EIO after any timers and counters, so with no timers/counters enabled, and pin offset set to 0, TX=FIO0 and RX=FIO1. Communication is in the common 8/n/1 format. Similar to RS232, except that the logic is normal CMOS/TTL. Connection to an RS232 device will require a converter chip such as the MAX233, which inverts the logic and shifts the voltage levels.

AsynchConfig

Command:

Byte	
0	Checksum8
1	0xF8
2	0x02
3	0x14
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	0x00
7	AsynchOptions Bit 7: Update Bit 6: UARTEnable Bit 5: Reserved
8-9 (8)	BaudFactor16 (BaudFactor8 for hardware 1.21)

Response:

Byte	
0	Checksum8
1	0xF8
2	0x02
3	0x14
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	AsynchOptions
8-9 (8)	BaudFactor16 (BaudFactor8 for hardware 1.21)

- AsynchOptions: If Update is true, the new parameters are written (otherwise just a read is done). If UARTEnable is true, the UART is enabled and the RX line will start buffering any incoming bytes.
- BaudFactor16 (BaudFactor8): This 16-bit value sets the baud rate according the following formula: BaudFactor16 = 2^16 - 48000000/(2 * Desired Baud). For example, a BaudFactor16 = 63036 provides a baud rate of 9600 bps. (With hardware revision 1.21, the value is only 8-bit and the formula is BaudFactor8 = 2^8 – TimerClockBase/(Desired Baud)).

5.2.19 AsynchTX

Sends bytes to the U6 UART which will be sent asynchronously on the transmit line.

Command:

Byte	
0	Checksum8
1	0xF8
2	1 + NumAsynchWords
3	0x15
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	0x00
7	NumAsynchBytesToSend
8	AsynchByte0
…	…

Response:

Byte	
0	Checksum8
1	0xF8
2	0x02
3	0x15
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	NumAsynchBytesSent
8	NumAsynchBytesInRXBuffer
9	0x00

- NumAsynchWords: This is the number of asynch data bytes divided by 2. If the number of bytes is odd, round up and add an extra zero to the packet.
- NumAsynchBytesToSend: Specifies how many bytes will be sent (0-56).
- NumAsynchBytesInRXBuffer: Returns how many bytes are currently in the RX buffer.

5.2.20 AsynchRX

Reads the oldest 32 bytes from the U6 UART RX buffer (received on receive terminal). The buffer holds 256 bytes.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x01
3	0x16
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	0x00
7	Flush

Response:

Byte	
0	Checksum8
1	0xF8
2	0x11
3	0x16
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	NumAsynchBytesInRXBuffer
8	AsynchByte0
…	…
39	AsynchByte31

- Flush: If nonzero, the entire 256-byte RX buffer is emptied. If there are more than 32 bytes in the buffer that data is lost.
- NumAsynchBytesInRXBuffer: Returns the number of bytes in the buffer before this read.
- AsynchByte#: Returns the 32 oldest bytes from the RX buffer.

5.2.21 I2C

Sends and receives serial data using I^2C (I2C) synchronous communication.

Command:

Byte	
0	Checksum8
1	0xF8
2	4+NumI2CWordsSend
3	0x3B
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	I2COptions Bits 7-5: Reserved Bit 4: Enable clock stretching. Bit 2: No stop when restarting. Bit 1: ResetAtStart Bit 0: Reserved
7	SpeedAdjust
8	SDAPinNum
9	SCLPinNum
10	AddressByte
11	Reserved
12	NumI2CBytesToSend
13	NumI2CBytesToReceive
14	I2CByte0
…	…

Response:

Byte	
0	Checksum8
1	0xF8
2	3+NumI2CWordsReceive
3	0x3B
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	Reserved
8	AckArray0
9	AckArray1
10	AckArray2
11	AckArray3
12	I2CByte0
…	…

- NumI2CWordsSend: This is the number of I2C bytes to send divided by 2. If the number of bytes is odd, round up and add an extra zero to the packet. This parameter is actually just to specify the size of this packet, as the NumI2CbytesToSend parameter below actually specifies how many bytes will be sent.
- I2COptions: If ResetAtStart is true, an I2C bus reset will be done before communicating.

- SpeedAdjust: Allows the communication frequency to be reduced. 0 is the maximum speed of about 150 kHz. 20 is a speed of about 70 kHz. 255 is the minimum speed of about 10 kHz.
- SDAP/SCLP -PinNum: Assigns which digital I/O line is used for each I2C line. Value passed is 0-19 corresponding to the normal digital I/O numbers as specified in Section 2.8. Note that the screw terminals labeled “SDA” and “SCL” on hardware revision 1.20 or 1.21 are not used for I2C. Note that the I2C bus generally requires pull-up resistors of perhaps 4.7 kΩ from SDA to Vs and SCL to Vs.
- AddressByte: This is the first byte of data sent on the I2C bus. The upper 7 bits are the address of the slave chip and bit 0 is the read/write bit. Note that the read/write bit is controlled automatically by the LabJack, and thus bit 0 is ignored.
- NumI2CBytesToSend: Specifies how many I2C bytes will be sent (0-50).
- NumI2CBytesToReceive: Specifies how many I2C bytes will be read (0-52).
- I2Cbyte#: In the command, these are the bytes to send. In the response, these are the bytes read.
- NumI2CWordsReceive: This is the number of I2C bytes to receive divided by 2. If the number of bytes is odd, the value is rounded up and an extra zero is added to the packet. This parameter is actually just to specify the size of this packet, as the NumI2CbytesToReceive parameter above actually specifies how many bytes to read.
- AckArray#: Represents a 32-bit value where bits are set if the corresponding I2C write byte was ack’ed. Useful for debugging up to the first 32 write bytes of communication. Bit 0 corresponds to the last data byte, bit 1 corresponds to the second to last data byte, and so on up to the address byte. So if n is the number of data bytes, the ACKs value should be (2^(n+1))-1.

5.2.22 SHT1X

Reads temperature and humidity from a Sensirion SHT1X sensor (which is used by the EI-1050). For more information, see the EI-1050 datasheet from labjack.com, and the SHT1X datasheet from sensirion.com.

Command:

Byte	
0	Checksum8
1	0xF8
2	0x02
3	0x39
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	DataPinNum (0-19)
7	ClockPinNum (0-19)
8	Reserved
9	Reserved

Response:

Byte	
0	Checksum8
1	0xF8
2	0x05
3	0x39
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	0x00
8	StatusReg
9	StatusRegCRC
10-11	Temperature
12	TemperatureCRC
13-14	Humidity
15	HumidityCRC

- Data/Clock -PinNum: Assigns which digital I/O line is used for each SPI line. Value passed is 0-7 corresponding to FIO0-FIO7. State and direction are controlled automatically for the specified lines.
- StatusReg: Returns a read of the SHT1X status register.
- Temperature: Returns the raw binary temperature reading.
- Humidity: Returns the raw binary humidity reading.
- #CRC: Returns the CRC values from the sensor.

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
40	SHT1x_COMM_TIME_OUT
41	SHT1x_NO_ACK
42	SHT1x_CRC_FAILED
43	SHT1X_TOO_MANY_W_BYTES
44	SHT1X_TOO_MANY_R_BYTES
45	SHT1X_INVALID_MODE
46	SHT1X_INVALID_LINE
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
59	STREAM_AUTORECOVER_ACTIVE
60	STREAM_AUTORECOVER_REPORT
63	STREAM_AUTORECOVER_OVERFLOW

Errorcodes (Continued):

Code	
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
102	TC_PIN_OFFSET_MUST_BE_4-8

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		4.75	5.0	5.25	volts
Supply Current (1)			100		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	4.8	5.0	5.25	
Maximum Current (3)	Self-Powered		400		mA
	Bus-Powered		0		mA
Vm+/Vm- Outputs					
Typical Voltage	No-load		±13		volts
	@ 2.5 mA		±12		volts
Maximum Current			2.5		mA
10UA & 200UA Current Outputs					
Absolute Accuracy (4)	~ 25 °C		±0.1	±0.2	%
Temperature Coefficient	See Section 2.5				ppm/°C
Maximum Voltage			VS - 2.0		volts

(1) Typical current drawn by the U6 itself, not including any user connections.

(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 U6. 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 U6 and out of the Vs terminals.

(4) This is compared to the value stored during factory calibration.

Parameter	Conditions	Min	Typical	Max	Units
Analog Inputs					
Typical Input Range (1)	Gain=1	-10.5		10.1	volts
Max AIN Voltage to GND (2)	Valid Readings	-11.8		11.3	volts
Max AIN Voltage to GND (3)	No Damage	YTBD		YTBD	volts
Input Bias Current (4)			20		nA
Input Impedance (4)			1		GΩ
Source Impedance (4)			1		kΩ
Integral Linearity Error			±YTBD		% FS
Differential Linearity Error			±YTBD		counts
Absolute Accuracy	Gain=1,10,100		±0.01		% FS
	Gain=1000		±YTBD		% FS
Temperature Drift			YTBD		ppm/°C
Noise (Peak-To-Peak) (5)	See Appendix B				
Effective Resolution (RMS) (6)	Gain=1		16-22		bits
Noise-Free Resolution (5)					bits
					mV
Command/Response Speed	See Section 3.1				
Stream Performance	See Section 3.2				

(1) Differential or single-ended.

(2) This is the maximum voltage on any AIN pin compared to ground for valid measurements. Single-ended inputs are limited by the input range above, but these numbers apply to differential inputs.

(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) The low-voltage analog inputs essentially connect directly to a SAR ADC on the U3, presenting a capacitive load to the signal source. The high-voltage inputs connect first to a resistive level-shifter/divider. The key specification in both cases is the maximum source impedance. As long as the source impedance is not over this value, there will be no substantial errors due to impedance problems.

(5) Measurements taken with AIN connected to a 2.048 reference (REF191 from Analog Devices) or GND. All "counts" data are aligned as 12-bit values. Noise-free data is determined by taking 128 readings and subtracting the minimum value from the maximum value.

(6) Effective (RMS) data is determined from the standard deviation of 128 readings. In other words, this data represents *most* readings, whereas noise-free data represents *all* readings.

Parameter	Conditions	Min	Typical	Max	Units
Analog Outputs (DAC)					
Nominal Output Range (1)	No Load	0.04		4.95	volts
	@ ±2.5 mA	0.225		4.775	volts
Resolution			12		bits
Absolute Accuracy	5% to 95% FS		±0.1		% FS
Integral Linearity Error			±YTBD		counts
Differential Linearity Error			±YTBD		counts
Error Due To Loading	@ 100 μA		0.1		%
	@ 1 mA		1		%
Source Impedance			50		Ω
Short Circuit Current (2)	Max to GND		YTBD		mA
Slew Rate			YTBD		V/ms
Digital I/O, Timers, Counters					
Low Level Input Voltage		-0.3		0.8	volts
High Level Input Voltage		2		5.8	volts
Maximum Input Voltage (3)	FIO	-10		10	volts
	EIO/CIO	-6		6	volts
Output Low Voltage (4)	No Load		0		volts
FIO	Sinking 1 mA		0.55		volts
EIO/CIO	Sinking 1 mA		0.18		volts
EIO/CIO	Sinking 5 mA		0.9		volts
Output High Voltage (4)	No Load		3.3		volts
FIO	Sourcing 1 mA		2.75		volts
EIO/CIO	Sourcing 1 mA		3.12		volts
EIO/CIO	Sourcing 5 mA		2.4		volts
Short Circuit Current (4)	FIO		6		mA
	EIO/CIO		18		mA
Output Impedance (4)	FIO		550		Ω
	EIO/CIO		180		Ω
Counter Input Frequency (5)				8	MHz
Input Timer Total Edge Rate (6)	No Stream			30000	edges/s
	While Streaming			7000	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) Continuous short circuit will not cause damage.

(3) 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.

(4) 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.

(5) Hardware counters. 0 to 3.3 volt square wave. Limit about 2 MHz with older hardware versions.

(6) 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. Limit about 10000 with older hardware versions.

B. Noise & Resolution Tables

The following tables provide typical noise levels of the U6 under ideal conditions. The resulting voltage resolution is then calculated based on the noise levels.

Measurements were taken with AIN0 connected to GND with a short jumper wire, or from internal ground channel #15.

All "counts" data are aligned as 24-bit values. To equate to counts at a particular resolution (Res) use the formula counts/(2^(24-Res)). For instance, with the U6 set to resolution=1 and the ±10 volt range, there are 1024 counts of noise when looking at 24-bit values. To equate this to 16-bit data, we take 1024/(2^8) which equals 4 counts of noise when looking at 16-bit values.

Noise-free data is determined by taking 128 readings and subtracting the minimum value from the maximum value.

RMS and Effective data are determined from the standard deviation of 128 readings. In other words, the RMS data represents most readings, whereas noise-free data represents all readings.

Resolution Index = 1

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	1024	14.0	1300.0	234	16.1	290.0
±1	1792	13.2	220.0	359	15.5	44.0
±0.1	6143	11.4	76.0	1116	13.9	14.0
±0.01	19466	9.8	24.0	3834	12.1	4.7

Resolution Index = 2

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	896	14.2	1100.0	192	16.4	240.0
±1	1280	13.7	160.0	274	15.9	34.0
±0.1	4223	12.0	52.0	856	14.3	11.0
±0.01	13319	10.3	16.0	2700	12.6	3.4

Resolution Index = 3

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	640	14.7	790.0	138	16.9	170.0
±1	1024	14.0	126.0	185	16.5	23.0
±0.1	3136	12.4	39.0	588	14.8	7.2
±0.01	10309	10.7	13.0	2118	13.0	2.6

Resolution Index = 4

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	512	15.0	632.0	94	17.5	116.0
±1	640	14.7	79.0	123	17.1	15.0
±0.1	2112	13.0	26.0	422	15.3	5.2
±0.01	7613	11.1	9.4	1455	13.5	1.8

Resolution Index = 5

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	320	15.7	395.0	67	17.9	82.0
±1	448	15.2	55.0	93	17.5	11.0
±0.1	1344	13.6	17.0	296	15.8	3.7
±0.01	5117	11.7	6.3	1055	14.0	1.3

Resolution Index = 6

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	256	16.0	316.0	50	18.4	61.0
±1	320	15.7	39.0	59	18.1	7.3
±0.1	1088	13.9	13.0	189	16.4	2.3
±0.01	4030	12.0	5.0	735	14.5	0.9

Resolution Index = 7

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	192	16.4	237.0	38	18.8	46.0
±1	256	16.0	32.0	51	18.3	6.3
±0.1	768	14.4	9.4	149	16.8	1.8
±0.01	2877	12.5	3.5	560	14.9	0.7

Resolution Index = 8

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	128	17.0	160.0	33	19.0	41.0
±1	192	16.4	24.0	39	18.7	4.8
±0.1	512	15.0	6.3	108	17.2	1.3
±0.01	1985	13.0	2.4	438	15.2	0.5

Res Index = 9 (-Pro only)

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	96	17.4	118.0	20	19.7	24.0
±1	124	17.0	15.0	21	19.6	2.6
±0.1	232	16.1	2.9	43	18.6	0.5
±0.01	1596	13.4	2.0	329	15.6	0.4

Res Index = 10 (-Pro only)

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	54	18.2	67.0	10	20.6	13.0
±1	65	18.0	8.0	13	20.3	1.7
±0.1	124	17.0	1.5	27	19.3	0.3
±0.01	1108	13.9	1.4	216	16.2	0.3

Res Index = 11 (-Pro only)

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	30	19.1	35.0	6	21.3	7.0
±1	30	19.1	3.5	6	21.3	0.8
±0.1	97	17.4	1.2	20	19.7	0.2
±0.01	1040	14.0	1.3	197	16.4	0.2

Res Index = 12 (-Pro only)

Range	Peak-To-Peak Noise	Noise-Free Resolution	Noise-Free Resolution	RMS Noise	Effective Resolution	Effective Resolution
volts	24-bit counts	bits	µV	counts	bits	µV
±10	21	19.6	26.0	4.2	22.0	5.0
±1	25	19.4	3.1	4.7	21.8	0.6
±0.1	97	17.4	1.2	20	19.7	0.2
±0.01	730	14.5	0.9	145	16.8	0.2

C. Enclosure & PCB Drawings

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

Units are inches.

YTBD: Very similar to UE9.