

U3 User's Guide

The complete user's guide for the U3, including documentation for the LabJackUD driver. Covers hardware versions 1.20, 1.21, and 1.30 (LV/HV).

To make a PDF of the whole manual, click "Export all" towards the upper-right of this page. Doing so converts these pages to a PDF on-the-fly, using the latest content, and can take 20-30 seconds. If it is not working for you try the "Print all" option instead.

If you are looking at a PDF or hardcopy, realize that the original is an online document at <http://labjack.com/support/u3/users-guide>.

Rather than using a PDF, though, we encourage you to use this web-based documentation. Some advantages:

- We can quickly change or update content.
- The site search includes the user's guide, forum, and all other resources at labjack.com. When you are looking for something try using the site search.
- For support, try going to the applicable user's guide page and post a comment. When appropriate we can then immediately add/change content on that page to address the question.

One other trick worth mentioning, is to browse the table of contents to the left. Rather than clicking on all the links to browse, you can click on the small black triangles to expand without reloading the whole page.

User's Guide

1 - Installation on Windows

The LJUD driver requires a PC running Windows. 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 by going to labjack.com/support/u3.

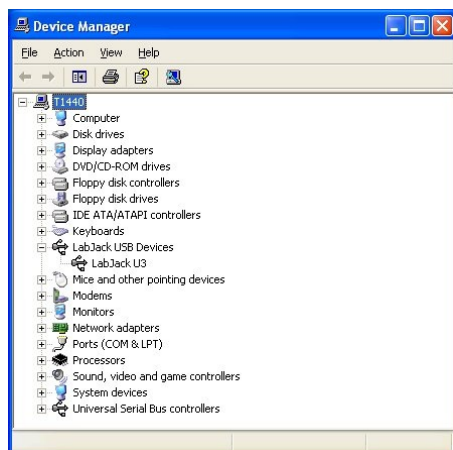
Connect the USB cable: The USB cable provides data and power. After the UD software installation is complete, connect the hardware and Windows should prompt with "Found New Hardware" and shortly after the Found New Hardware Wizard will open. When the Wizard appears allow Windows to install automatically 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 U3 showing the serial number. Click on the "USB - 1" entry below the serial number to bring up the U3 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 U3.

If LJControlPanel does not find the U3, check Windows Device Manager to see if the U3 installed correctly. One way to get to the Device Manager is:

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

The entry for the U3 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 U3 and repeat the Found New Hardware Wizard as described above.



Correctly Functioning U3 in Windows Device Manager

1.1 - Control Panel Application (LJControlPanel)

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

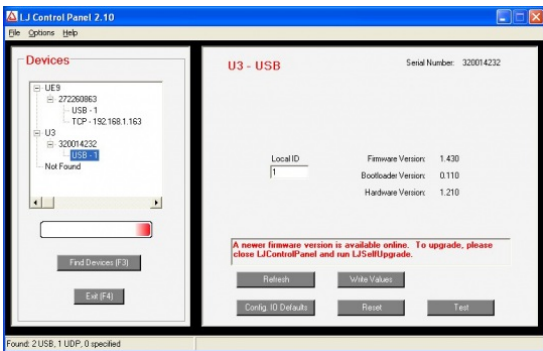


Figure 1-1. LJControlPanel Main Window

Figure 1-1 shows the results from a typical search. The application found one U3 connected by USB. The USB connection has been selected in Figure 1-1, bringing up the configuration window on the right side.

- Refresh: Reload the window using values read from the device.
- Write 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.

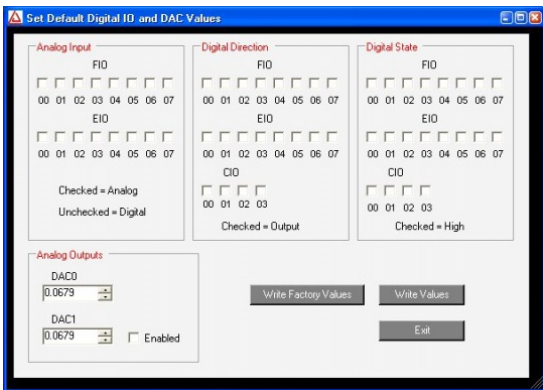


Figure 1-2. LJControlPanel U3 Configure Defaults Window

Figure 1-2 shows the configuration window for U3 defaults. These are the values that will be loaded by the U3 at power-up or reset. The factory defaults, as shown above, are all lines configured as digital input.

Figure 1-3 shows the U3 test window. This window continuously (once per second) writes to and reads from the selected LabJack.

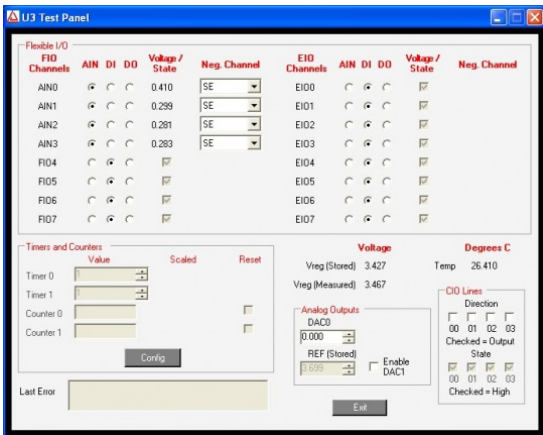


Figure 1-3. LJControlPanel U3 Test Window

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 U3.
- Search for Ethernet devices using specified IP addresses: Does not apply to the U3.

1.2 - Self-Upgrade Application (LJSelfUpgrade)

The processor in the U3 has field upgradeable flash memory. The self-upgrade application shown in Figure 1-5 programs the latest firmware onto the processor.

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

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

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

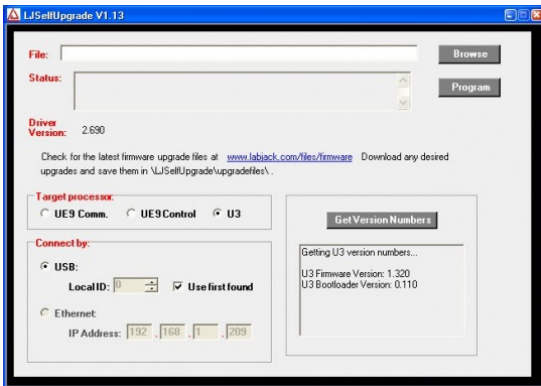


Figure 1-5. Self-Upgrade Application

If problems are encountered during programming, try the following:

1. Unplug the U3, wait 5 seconds then reconnect the U3. Click OK then press program again.
2. If step 1 does not fix the problem unplug the U3 and watch the LED while plugging the U3 back in. Follow the following steps based on the LED's activity.
 1. **If the LED is blinking continuously (flash mode)**, connect a jumper between FIO4 and SPC (FIO0 to SCL on U3 1.20/1.21), then unplug the U3, wait 5 seconds and plug the U3 back in. Try programming again (disconnect the jumper before programming).
 2. **If the LED blinks several times and stays on**, connect a jumper between FIO5 and SPC (FIO1 to SCL on U3 1.20/1.21), then unplug the U3, wait 5 seconds and plug the U3 back in. Try programming again (disconnect the jumper before programming).
 3. **If the LED blinks several times and stays off**, the U3 is not enumerating. Please restart your computer and try to program again.
 4. **If there is no LED activity**, connect a jumper between FIO5 and SPC (FIO1 to SCL on U3 1.20/1.21), then unplug the U3, wait 5 seconds and plug the U3 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 FIO4 and SPC (FIO0 to SCL on U3 1.20/1.21), then unplug the U3, wait 5 seconds and plug the U3 back in.
 5. **If the LED does a repeating pattern of 3 blinks then pause**, the U3 has detected internal memory corruption and you will have to contact LabJack Support.
3. If there is no activity from the U3's LED after following the above steps, please contact support.

2 - Hardware Description

The U3 has 3 different I/O areas:

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

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

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

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



Figure 2-1. LabJack U3

2.1 - USB

For information about USB installation, see [Section 1](#).

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

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

The LabJack vendor ID is 0x0CD5. The product ID for the U3 is 0x0003.

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 U3 to the host.

2.2 - Status LED

There is a green status LED on the LabJack U3. This LED blinks on reset, and then remains steadily lit. Other LED behavior is generally related to flash upgrade modes ([Section 1.2](#)).

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 U3.

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

2.4 - VS

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

2.5 - Flexible I/O (FIO/EIO)

The FIO and EIO ports on the LabJack U3 can be individually configured as digital input, digital output, or analog input. This is FIO0-EIO7 on the U3-LV (16 lines), **or FIO4-EIO7 on the U3-HV (12 lines)**. In addition, up to 2 of these lines can be configured as timers, and up to 2 of these lines can be configured as counters. If a line is configured as analog, it is called AINx according to the following table:

AIN0	FIO0		AIN8	EIO0
AIN1	FIO1		AIN9	EIO1
AIN2	FIO2		AIN10	EIO2
AIN3	FIO3		AIN11	EIO3
AIN4	FIO4		AIN12	EIO4
AIN5	FIO5		AIN13	EIO5
AIN6	FIO6		AIN14	EIO6
AIN7	FIO7		AIN15	EIO7
Table 2.5-1. Analog Input Pin Locations				

On the U3-HV, compared to the -LV, **the first four flexible I/O are fixed as analog inputs (AIN0-AIN3) with a nominal ± 10 volt input range**. All digital operations, including analog/digital configuration, are ignored on these 4 fixed analog inputs.

Timers and counters can appear on various pins, but other I/O lines never move. For example, Timer1 can appear anywhere from FIO4 to EIO1, depending on TimerCounterPinOffset and whether Timer0 is enabled. On the other hand, FIO5 (for example), is always on the screw terminal labeled FIO5, and AIN5 (if enabled) is always on that same screw terminal.

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

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

Byte #			
6	WriteMask	15	Write all parameters
8	TimerCounterConfig	0	No Timers/Counters. Offset = 4
9	DAC1 Enable	0	DAC1 Disabled. (Ignored on HW 1.3)
10	FIOAnalog	0	FIO all digital.
11	EIOAnalog	0	EIO all digital.
Table 2.5-2. ConfigIO Factory Default Values			
Byte #			
8	TimerClockConfig	130	Set clock to 48MHz.
9	TimerClockDivisor	0	Divisor = 0.
Table 2.5-3. ConfigTimerClock Factory Default Values			

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

```
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 0, 0);
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 4, 0);
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tc48MHZ, 0);
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 0, 0);
ePut (lngHandle, LJ_ioPUT_COUNTER_ENABLE, 0, 0, 0);
ePut (lngHandle, LJ_ioPUT_COUNTER_ENABLE, 1, 0, 0);
ePut (lngHandle, LJ_ioPUT_DAC_ENABLE, 1, 0, 0); //Ignored on hardware rev 1.30+.
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_PORT, 0, 0, 16);
```

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

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

2.6 - AIN

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

Analog input resolution is 12-bits. The range of single-ended analog inputs is normally about 0-2.44, and there is a "special" 0-3.6 volt range available. The range of differential analog inputs is typically ± 2.4 volts, but is pseudobipolar, not true bipolar. The difference (positive channel minus negative channel) can be -2.4 volts, but neither input can have a voltage less than -0.3 volts to ground. For valid measurements, the voltage on every low-voltage analog input pin, with respect to ground, must be within -0.3 to +3.6 volts. See [Appendix A](#) for voltage limits to avoid damage.

On the U3-HV, compared to the -LV, the first four flexible I/O are fixed as analog inputs (AIN0-AIN3), and have scaling such that the input range is a true bipolar ± 10 volts normally, and -10 to +20 volts when using the "special" range. The input impedance of these four lines is roughly 1 M Ω , which is good, but less than the normal low voltage analog inputs. Analog/digital configuration and all other digital operations on these pins are ignored. FIO4-EIO7 are still available as flexible I/O, same as the U3-LV.

Because the scaling on the high-voltage inputs on the U3-HV (AIN0-AIN3) is inherently single-ended, a factory calibration is not possible for differential readings. If a differential reading is requested where either channel is a high-voltage channel, the driver will return the raw binary reading and the user must handle calibration/conversion.

The analog inputs have a QuickSample option where each conversion is done faster at the expense of increased noise. This is enabled by passing a nonzero value for put_config special channel *LJ_chAIN_RESOLUTION*. There is also a LongSettling option where additional settling time is added between the internal multiplexer configuration and the analog to digital conversion. This allows signals with more source impedance, and is enabled by passing a nonzero value for put_config special channel *LJ_chAIN_SETTLING_TIME*. Both of these options are disabled by default. This applies to command/response mode only, and the resulting typical data rates are discussed in [Section 3.1](#). For stream mode, see [Section 3.2](#).

Note that sinking excessive current into digital outputs can cause substantial errors in analog input readings. See [Section 2.8.1.4](#) for more info.

2.6.1 - Channel Numbers

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

Positive Channel #	
0-7	AIN0-AIN7 (FIO0-FIO7)
8-15	AIN8-AIN15 (EIO0-EIO7)
30	Temp Sensor
31	Vreg
Table 2.6.1-1. Positive Channel Numbers	
Negative Channel #	
0-7	AIN0-AIN7 (FIO0-FIO7)
8-15	AIN8-AIN15 (EIO0-EIO7)
30	Vref
31 or 199	Single-Ended
32	Special 0-3.6 or -10/+20 (UD Only)
Table 2.6.1-2 Negative Channel Numbers	

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

If the negative channel is set to anything besides 31/199, the U3 does a differential conversion and returns a pseudobipolar value. If the negative channel is set to 31/199, the U3 does a single-ended conversion and returns a unipolar value. Channel 30 puts the internal voltage reference Vref (~2.44 volts) on the negative input of the ADC.

Channel 32 is a special negative channel supported by the LabJack UD driver. When used, the driver will actually pass 30 as the negative channel to the U3, and when the result is returned the driver adds Vref to the value. For a low-voltage analog input this results in a full span on the positive channel of about 0 to 4.88 volts (versus ground), but since the voltage on any analog input cannot exceed 3.6 volts, only 75% of the converter's range is used and the span is about 0 to 3.6 volts. For a high-voltage analog input, channel 32 (special range) results in a span of about -10 to +20 volts.

In the U3 examples that accompany the [Exodriver](#), u3.c also supports channel 32 in calls to *eAIN()*.

Channel 32 is also supported in [LabJackPython](#):

```
# On the U3, wire a jumper from DAC0 to FIO0, then run:
>>> import u3
>>> d = u3.U3()
>>> d.configIO(FIOAnalog = 1) # Set FIO0 to analog
>>> d.writeRegister(5000, 3) # Set DAC0 to 3 V
```

```
>>> d.getAIN(0, 32)
3.0141140941996127
```

For the four high-voltage channels on the U3-HV, the special channel negative channel also puts Vref on the negative. This results in an overall range of about -10 to +20 volts on the positive input.

2.6.2 - Converting Binary Readings to Voltages

Following are the nominal input voltage ranges for the low-voltage analog inputs. This is all analog inputs on the U3-LV, and AIN4-AIN15 on the U3-HV.

	Max V	Min V		
Single-Ended	2.44	0		
Differential	2.44	-2.44		
Special 0-3.6	3.6	0		
Table 2.6.2-1. Nominal Analog Input Voltage Ranges for Low-Voltage Channels				
	Max V	Min V		
Single-Ended	10.3	-10.3		
Differential	N/A	N/A		
Special -10/+20	20.1	-10.3		
Table 2.6.2-2. Nominal Analog Input Voltage Ranges for High-Voltage Channels				
	Max V	Min V		

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

The “special” range (0-3.6 on low-voltage channels and -10/+20 volts on high-voltage channels) is obtained by doing a differential measurement where the negative channel is set to the internal Vref (2.44 volts). For low-voltage channels, simply do the low-voltage differential conversion as described below, then add the stored Vref value. For high-voltage channels, do the same thing, then multiply by the proper high-voltage slope, divide by the single-ended low-voltage slope, and add the proper high-voltage offset. The UD driver handles these conversions automatically.

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

```
Volts(uncalibrated) = (Bits/65536)*Span (Single-Ended)
Volts(uncalibrated) = (Bits/65536)*Span - Span/2 (Differential)
```

Binary readings are always unsigned integers.

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

```
Volts = (Slope * Bits) + Offset
```

Since the U3 uses multiplexed channels connected to a single analog-to-digital converter (ADC), all low-voltage channels have the same calibration for a given configuration. High-voltage channels have individual scaling circuitry out front, and thus the calibration is unique for each channel.

See [Section 5.4](#) for detail about the location of the U3 calibration constants.

2.6.2.1 - Analog Inputs With DAC1 Enabled (Hardware Revisions 1.20 & 1.21 only)

This Section only applies to the older hardware revisions 1.20 and 1.21. Starting with hardware revision 1.30, DAC1 is always enabled and does not affect the analog inputs.

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

	Max V	Min V		
Single-Ended	3.3	0		
Differential	3.3	-3.3		
Special -10/+20	N/A	N/A		
Table 2.6.2.1-1. Nominal Analog Input Voltage Ranges (DAC1 Enabled)				
	Max V	Min V		

Note that the minimum differential input voltage of -3.3 volts means that the positive channel can be as much as 3.3 volts less than the negative channel, not that a channel can measure 3.3 volts less than ground. The voltage of any analog input pin, compared to ground, must be in the range -0.3 to +3.6 volts, for specified performance. See [Appendix A](#) for voltage limits to avoid damage.

Negative channel numbers 30 and 32 are not valid with DAC1 enabled.

When DAC1 is enabled, the slope/offset calibration constants are not used to convert raw readings to voltages. Rather, the Vreg value is retrieved from the Mem area, and used with the approximate single-ended or differential conversion equations above, where Span is Vreg (single-ended) or 2Vreg (differential).

2.6.3 - Typical Analog Input Connections

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

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

Impedance: When connecting the U3, or any measuring device, to a signal source, it must be considered what impact the measuring device will have on the signal. The main consideration is whether the currents going into or out of the U3 analog input will cause noticeable voltage errors due to the impedance of the source. To maintain consistent 12-bit results, it is recommended to keep the source impedance within the limits specified in [Appendix A](#).

Resolution (and Accuracy): Based on the measurement type and resolution of the U3, the resolution can be determined in terms of

voltage or engineering units. For example, assume some temperature sensor provides a 0-10 mV signal, corresponding to 0-100 degrees C. Samples are then acquired with the U3 using the 0-2.44 volt single-ended input range, resulting in a voltage resolution of about $2.44/4096 = 596 \mu\text{V}$. That means there will be about 17 discrete steps across the 10 mV span of the signal, and the temperature resolution is about 6 degrees C. If this experiment required a resolution of 1 degrees C, this configuration would not be sufficient. Accuracy will also need to be considered. Appendix A places some boundaries on expected accuracy, but an in-system calibration can generally be done to provide absolute accuracy down to the linearity (INL) limits of the U3.

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

2.6.3.1 - Signal from the LabJack

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

2.6.3.2 - Unpowered Isolated Signal

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

2.6.3.3 - Signal Powered By the LabJack

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

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

2.6.3.4 - Signal Powered Externally

An example is a box with a wire coming out that is defined as a 0-2 volt analog signal and a second wire labeled as ground. The signal is known to have 0-2 volts compared to the ground wire, but the complication is what is the voltage of the box ground compared to the LabJack ground.

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

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

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

The SGND terminals (on the same terminal block as SPC) 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 U3. If no large voltages are noted, connect the ground to U3 SGND with a 100 Ω series resistor. Then again use the DMM to measure the voltage of each signal wire before connecting to the U3.

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 U3. 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 U3 ground.

2.6.3.5 - Amplifying Small Signal Voltages

The best results are generally obtained when a signal voltage spans the full analog input range of the LabJack. If the signal is too small it can be amplified before connecting to the LabJack. One good way to handle low-level signals such as thermocouples is the [LJTick-InAmp](#), which is a 2-channel instrumentation amplifier module that plugs into the U3 screw-terminals.

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

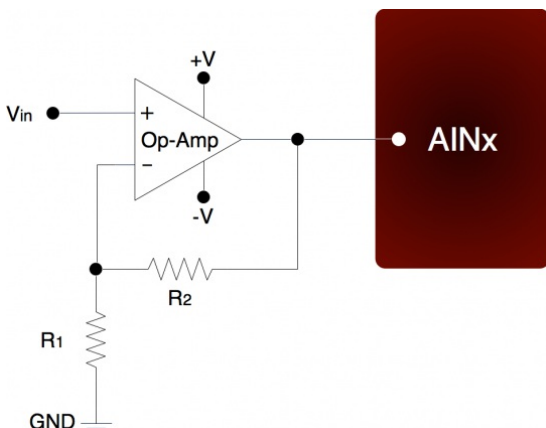


Figure 2-3. Non-Inverting Op-Amp Configuration

The gain of this configuration is:

$$V_{out} = V_{in} * (1 + (R2/R1))$$

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

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

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

2.6.3.6 - Signal Voltages Beyond 0-2.44 Volts (and Resistance Measurement)

The normal input range for a low voltage channel on the U3 is about 0-2.44 volts. The easiest way to handle larger voltages is often by using the [LTick-Divider](#), which is a two channel buffered divider module that plugs into the U3 screw-terminals.

The basic way to handle higher unipolar 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 U3 (GND).

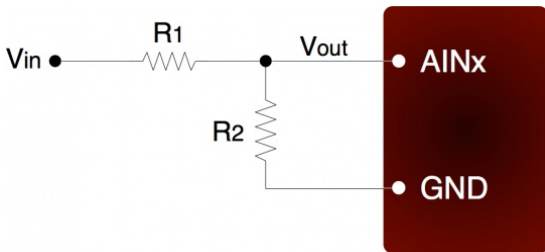


Figure 2-4. Voltage Divider Circuit

The attenuation of this circuit is determined by the equation:

$$V_{out} = V_{in} * (R2 / (R1+R2))$$

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 the values specified in Appendix A, so R1 can generally be fixed at the max recommended value and R2 can be adjusted for the desired attenuation.

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

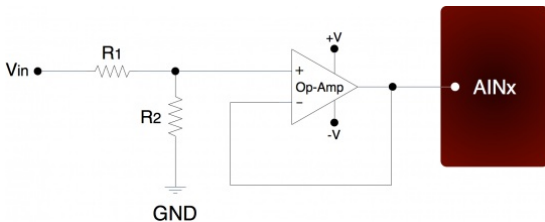


Figure 2-5. Buffered Voltage Divider Circuit

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

The information above also applies to resistance measurement. A common way to measure resistance is to build a voltage divider as shown in Figure 2-4, 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.

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.

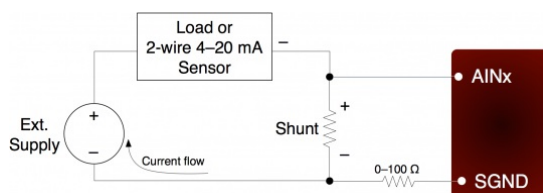


Figure 2-5. Current Measurement With Arbitrary Load or 2-Wire 4-20 mA Sensor

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

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

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

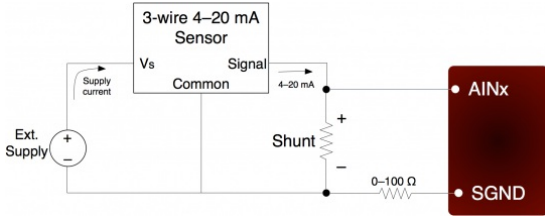


Figure 2-6. Current Measurement With 3-Wire 4-20 mA (Sourcing) Sensor

The sensor shown in Figure 2-6 is a sourcing type, where the signal sources the 4-20 mA current which is then sent through the shunt resistor and sunk into ground. Another type of 3-wire sensor is the sinking type, where the 4-20 mA current is sourced from the positive supply, sent through the shunt resistor, and then sunk into the signal wire. If sensor ground is connected to U3 ground, the sinking type of sensor presents a problem, as at least one side of the resistor has a high common mode voltage (equal to the positive sensor supply). If the sensor is isolated, a possible solution is to connect the sensor signal or positive sensor supply to U3 ground (instead of sensor ground). This requires a good understanding of grounding and isolation in the system. The LJTick-CurrentShunt is often a simple solution.

Both Figure 2-5 and 2-6 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 U3 screw-terminals.

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 (pull-down or pull-up) can be placed from the AINx screw terminal to the desired voltage (GND, VS, DACx, ...). A 100 k Ω resistor should pull the analog input readings to within 50 mV of any desired voltage, but obviously degrades the input impedance to 100 k Ω . For the specific case of pulling a floating channel to 0 volts, a 1 M Ω resistor to GND can typically be used to provide analog input readings of less than 50 mV. This information is for a low-voltage analog input channel on a U3.

Note that the four high-voltage channels on the U3-HV do sit at a predictable 1.4 volts. You can use a pull-down or pull-up resistor with the high-voltage inputs, but because their input impedance is lower the resistor must be lower (~1k might be typical) and thus the signal is going to have to drive substantial current.

2.6.3.9 - Signal Voltages Near Ground

The nominal input range of a low-voltage single-ended analog input is 0-2.44 volts. So the nominal minimum voltage is 0.0 volts, but the variation in that minimum can be about +/-40 mV, and thus the actual minimum voltage could be 0.04 volts.

This is not an offset error, but just a minimum limit. Assume the minimum limit of your U3 happens to be 10 mV. If you apply a voltage of 0.02 volts it will read 0.02 volts. If you apply a voltage of 0.01 volts it will read 0.01 volts. If you apply a voltage less than 0.01 volts, however, it will still read the minimum limit of 0.01 volts in this case.

One impact of this, is that a short to GND is usually not a good test for noise and accuracy. We often use a 1.5 volt battery for simple tests.

If performance all the way to 0.0 is needed, use a differential reading (which is pseudobipolar). Connect some other channel to GND with a small jumper, and then take a differential reading of your channel compared to that grounded channel.

The nominal input range of a high-voltage single-ended analog input is +/-10 volts, so readings around 0.0 are right in the middle of the range and not an issue.

2.6.4 - Internal Temperature Sensor

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

With the UD driver, the internal temperature sensor is read by acquiring single-ended analog input channel 30, and returns degrees K.

2.7 - DAC

The LabJack U3 has 2 analog outputs (DAC0 and DAC1) that are available on the screw terminals. Each analog output can be set to a voltage between about 0.04 and 4.95 volts with 10 bits of resolution (8 bits on older hardware revision 1.20/1.21). The maximum output voltage is limited by the supply voltage to the U3.

Starting with hardware revision 1.30, DAC1 is always enabled and does not affect the analog inputs, but with older hardware the second analog output is only available in certain configurations. With hardware revisions <1.30, if the analog inputs are using the internal 2.4 volt reference (the most accurate option), then DAC1 outputs a fixed voltage of 1.5*Vref. Also with hardware revisions <1.30, if DAC1 is enabled the analog inputs use Vreg (3.3 volts) as the ADC reference, which is not as stable as the internal 2.4 volt reference.

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

The DACs are derived from PWM signals that are affected by the timer clock frequency (Section 2.9). The default timer clock frequency of the U3 is set to 48 MHz, and this results in the minimum DAC output noise. If the frequency is lowered, the DACs will have more noise, where the frequency of the noise is the timer clock frequency divided by 216. This effect is more exaggerated with the 10-bit DACs on hardware revision 1.30+, compared to the 8-bit DACs on previous hardware revisions. The noise with a timer clock of 48/12/4/1 MHz is roughly 5/20/100/600 mV. If lower noise performance is needed at lower timer clock frequencies, use the power-up default setting in LJControlPanel to force the device to use 8-bit DAC mode (uses the low-level CompatibilityOptions byte documented in Section 5.2.2). A large capacitor (at least 220 uF) from DACn to GND can also be used to reduce noise.

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

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

$$\text{Bits}(\text{uncalibrated}) = (\text{Volts}/4.95) * 256$$

For a proper calculation, though, use the calibration values (Slope and Offset) stored in the internal flash on the processor (Table 2-7):

$$\text{Bits} = (\text{Slope} * \text{Volts}) + \text{Offset}$$

The previous apply when using the original 8-bit DAC commands supported on all hardware versions. To take advantage of the 10-bit resolution on hardware revision 1.30, new commands have been added (Section 5.2.5) where the binary values are aligned to 16-bits. The cal constants are still aligned to 8-bits, however, so the slope and offset should each be multiplied by 256 before using in the above formula.

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 U3 to add 20 analog outputs. The LJTick-DAC improves on the various shortcomings of the built-in DACs on the U3:

- Range of +10.0 to -10.0 volts.
- Resolution of 14-bits.
- Slew rate of 0.1 V/ μ s.
- Based on a reference, rather than regulator, so more accurate and stable.
- Does not affect analog inputs in any configuration.

2.7.1 - Typical Analog Output Connections

2.7.1.1 - High Current Output

The DACs on the U3 can output quite a bit of current, but they 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 2-3. A simple RC filter can be added between the DAC output and the amp input for further noise reduction. Note that the ability of the amp to source/sink current near the power rails must still be considered. A possible op-amp choice would be the TLV246x family (ti.com).

2.7.1.2 - Different Output Ranges

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

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

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

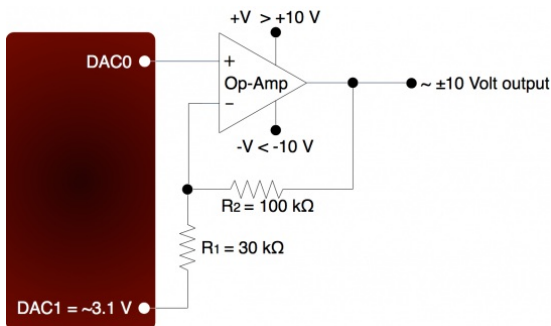


Figure 2-8. ± 10 Volt DAC Output Circuit

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

2.8 - Digital I/O

The LabJack U3 has up to 20 digital I/O channels. 16 are available from the flexible I/O lines, and 4 dedicated digital I/O (CIO0-CIO3) are available on the DB15 connector. The first 4 lines, FIO0-FIO3, are unavailable on the U3-HV. Each digital line can be individually configured as input, output-high, or output-low. The digital I/O use 3.3 volt logic and are 5 volt tolerant.

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

```
0-7    FIO0-FIO7    (0-3 unavailable on U3-HV)
8-15   EIO0-EIO7
16-19  CIO0-CIO3
```

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

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

All digital I/O on the U3 have 3 possible states: input, output-high, or output-low. Each bit of I/O can be configured individually. When configured as an input, a bit has a ~100 k Ω 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 5 ms at power-up where all digital I/O are in the factory default condition.

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

Many function parameters contain specific bits within a single integer parameter to write/read specific information. In particular, most digital I/O parameters contain the information for each bit of I/O in one integer, where each bit of I/O corresponds to the same bit in the parameter (e.g. the direction of FIO0 is set in bit 0 of parameter FIODir). For instance, in the low-level function ConfigU3, the parameter FIODirection is a single byte (8 bits) that writes/reads the power-up direction of each of the 8 FIO lines:

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

2.8.1 - Typical Digital I/O Connections

2.8.1.1 - Input: Driven Signals

The most basic connection to a U3 digital input is a driven signal, often called push-pull. With a push-pull signal the source is typically providing a high voltage for logic high and zero volts for logic low. This signal is generally connected directly to the U3 digital input, considering the voltage specifications in [Appendix A](#). If the signal is over 5 volts, it can still be connected with a series resistor. The digital inputs have protective devices that clamp the voltage at GND and VS, so the series resistor is used to limit the current through these protective devices. For instance, if a 24 volt signal is connected through a 22 k Ω 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 U3. The series resistor should be 22 k Ω 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 U3, then no additional ground connection is used. If the signal is known to not have a common ground with the U3, then the signal ground can simply be connected to U3 GND. If there is uncertainty about the relationship between signal ground and U3 ground (e.g. possible common ground through AC mains), then a ground connection with a ~10 Ω series resistor is generally recommended (see [Section 2.6.3.4](#)).

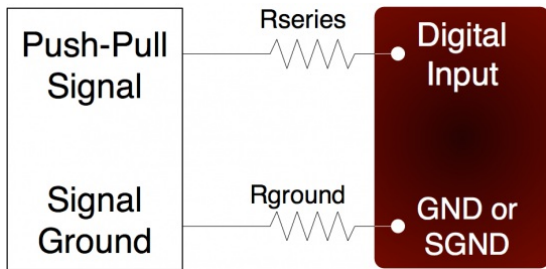


Figure 2-8. Driven Signal Connection To Digital Input

Figure 2-8 shows typical connections. Rground is typically 0-100 Ω . Rseries is typically 0 Ω (short-circuit) for 3.3/5 volt logic, or 22 k Ω (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 U3 if possible.

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

2.8.1.2 - Input: Open-Collector Signals

Open-collector (also called open-drain 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 U3 digital inputs have a 100 k Ω 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 k Ω from Vs to digital input) will be installed to increase the strength and speed of the logic high condition.

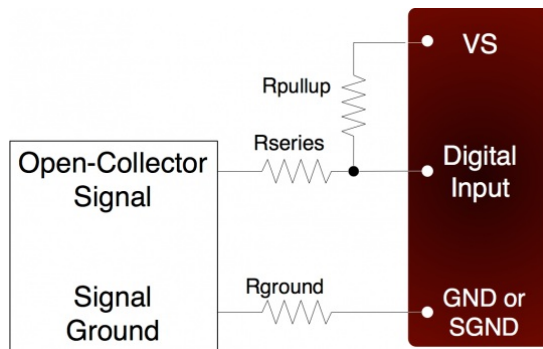


Figure 2-9. Open-Collector (NPN) Connection To Digital Input

Figure 2-9 shows typical connections. Rground is typically 0-100 Ω , Rseries is typically 0 Ω , and Rpullup, 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 k Ω 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.

Without the optional resistors, the figure simplifies to:

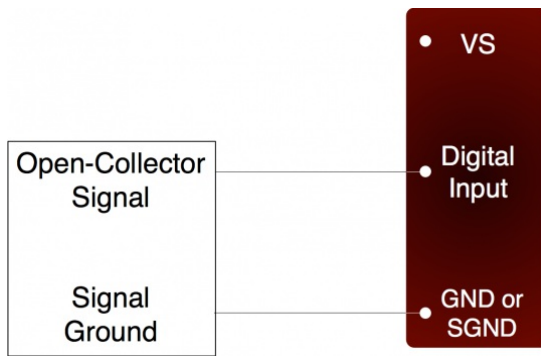


Figure 2-9b. Simplified Open-Collector (NPN) Connection To Digital Input Without Optional Resistors

Note that an individual ground connection is often not needed for every signal. Any signals powered by the same external supply, or otherwise referred to the same external ground, should share a single ground connection to the U3 if possible.

2.8.1.3 - Input: Mechanical Switch Closure

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

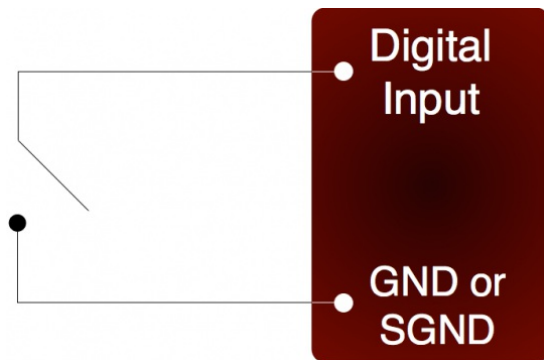


Figure 2-10. Basic Mechanical Switch Connection To Digital Input

When the switch is open, the internal 100 k Ω 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.

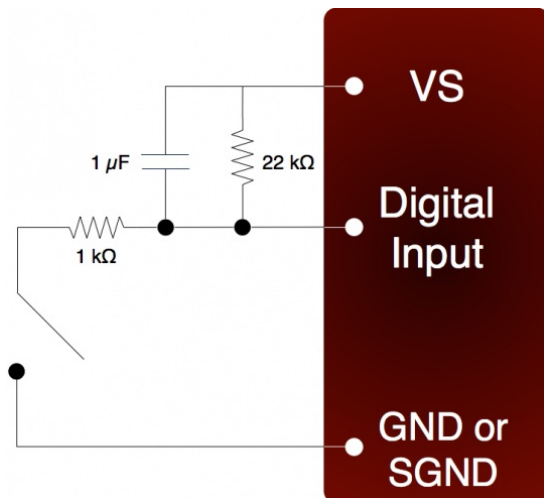


Figure 2-11. Passive Hardware Debounce

Figure 2-11 shows one possible configuration for passive hardware debounce. First, consider the case where the 1 k Ω 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 closed. 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).

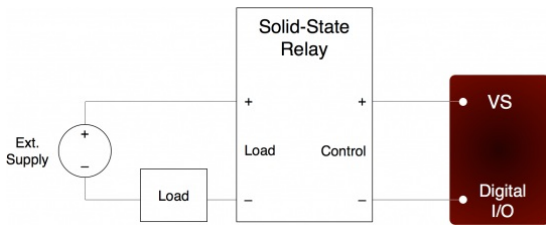


Figure 2-13. Relay Connections (Sinking Control, High-Side Load Switching)

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) or 550 Ω (FIO) in series. When using an EIO/CIO line, the resulting voltage across the control inputs of the relay will be about $5 \times 1500 / (1500 + 180) = 4.5$ volts (the other 0.5 volts is dropped across the internal resistance of the EIO/CIO line). With an FIO line the voltage across the inputs of the relay will be about $5 \times 1500 / (1500 + 550) = 3.7$ 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) or 550 Ω (FIO) in series. When using an EIO/CIO line, the resulting voltage across the control inputs of the relay will be about $1.7 \times 1500 / (1500 + 180) = 1.5$ volts. With an FIO line the voltage across the inputs of the relay will be about $1.7 \times 1500 / (1500 + 550) = 1.2$ 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.

Note that sinking excessive current into digital outputs can cause noticeable shifts in analog input readings. For example, the FIO sinking configuration above sinks about 2.4 mA into the digital output to turn the SSR on, which could cause a shift of roughly 1 mV to analog input readings.

Mechanical relays require more control current than SSRs, and cannot be controlled directly by the digital I/O on the U3. 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 U3 DACs can source enough current to control almost any SSR and even some mechanical relays, and thus can be a convenient way to control 1 or 2 relays. With the DACs you would typically use a sourcing configuration (DAC/GND) rather than sinking (VS/DAC).

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

Another accessory available from LabJack is the LJTick-RelayDriver. This is a two channel module that plugs into the U3 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 U3 has 2 timers (Timer0-Timer1) and 2 counters (Counter0-Counter1). When any of these timers or counters are enabled, they take over an FIO/EIO line in sequence (Timer0, Timer1, Counter0, then Counter1), starting with FIO0+TimerCounterPinOffset. Some examples:

1 Timer enabled, Counter0 disabled, Counter1 disabled, and TimerCounterPinOffset=4:
FIO4=Timer0

1 Timer enabled, Counter0 disabled, Counter1 enabled, and TimerCounterPinOffset=6:
FIO6=Timer0
FIO7=Counter1

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

Starting with hardware revision 1.30, timers/counters cannot appear on FIO0-3, and thus **TimerCounterPinOffset must be 4-8**. A value of 0-3 will result in an error. This error can be suppressed by a power-up default setting in LJControlPanel. If suppressed, a 0-3 will result in an offset of 4.

Timers and counters can appear on various pins, but other I/O lines never move. For example, Timer1 can appear anywhere from FIO4 to EIO1, depending on TimerCounterPinOffset and whether Timer0 is enabled. On the other hand, FIO5 (for example), is always on the screw terminal labeled FIO5, and AIN5 (if enabled) is always on that same screw terminal.

Note that Counter0 is not available with certain timer clock base frequencies. In such a case, it does not use an external FIO/EIO pin. 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-Timer1) 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)
Table 2.9-1. U3 Timer Modes	
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
Table 2.9-2. U3 Timer Base Clock Options	

Note that these clocks apply to the U3 hardware revision 1.21+. With hardware revision 1.20 all clocks are half of the values above.

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 FIOEIO pin. The divisor has a range of 0-255, where 0 corresponds to a division of 256.

Note that the DACs (Section 2.7) are derived from PWM signals that are affected by the timer clock frequency. The default timer clock frequency of the U3 is set to 48 MHz, as this results in the minimum DAC output noise. If the frequency is lowered, the DACs will have more noise, where the frequency of the noise is the timer clock frequency divided by 2^{16} .

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	0.06
4	4 MHz /Divisor	61.04	0.238
5	12 MHz /Divisor	183.11	0.715
6	48 MHz /Divisor	732.42	2.861
Table 2.9.1.1-1. 16-bit PWM Frequencies			

Note that the clocks above apply to the U3 hardware revision 1.21. With hardware revision 1.20 all clocks are half of those values.

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	
TimerBase		Divisor=1	Divisor=256
0	4 MHz	15625	N/A
1	12 MHz	46875	N/A
2	48 MHz (default)	187500	N/A
3	1 MHz /Divisor	3906.25	15.259
4	4 MHz /Divisor	15625	61.035
5	12 MHz /Divisor	46875	183.105
6	48 MHz /Divisor	187500	732.422
Table 2.9.1.2-1. 8-bit PWM Frequencies			

Note that the clocks above apply to the U3 hardware revision 1.21. With hardware revision 1.20 all clocks are half of those values.

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 $\text{TimerClockBase}/\text{TimerClockDivisor}$) 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 U3 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 $\text{TimerClockBase}/\text{TimerClockDivisor}$.

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):

$\text{Debounce Period} = (0-16 \text{ ms}) + (\text{TimerValue} * 16 \text{ ms})$

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 ([Figure 2-10](#)). 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 $\text{TimerClockBase}/\text{TimerClockDivisor}$ divided by $2 * \text{Timer\#Value}$. 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			
TimerBaseClock		Divisor=1 Value=1	Divisor=256 Value=256
0	4 MHz	2000000	7812.5
1	12 MHz	6000000	23437.5
2	48 MHz (default)	24000000	93750
		Divisor=1 Value=1	Divisor=256 Value=256
3	1 MHz /Divisor	500000	7.629
4	4 MHz /Divisor	2000000	30.518
5	12 MHz /Divisor	6000000	91.553
6	48 MHz /Divisor	24000000	366.211
Table 2.9.1.7-1. Mode 7 Frequency Ranges			

Note that the clocks above apply to the U3 hardware revision 1.21. With hardware revision 1.20 all clocks are half of those values.

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

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

Z-phase support

Quadrature mode supports Z-Phase. When enabled this feature will set the count to zero when the specified IO line sees a logic high.

Z-phase is controlled by the value written to the timer during initialization. To enable z-phase support set bit 15 to 1 and set bits 0 through 4 to the DIO number that Z is connected to. EG: for a Z-line on EIO3 set the timer value to 0x800B or 32779. This value should be sent to both the A and B timers. When enabled this feature will set the count to zero when the specified IO line sees a logic high.

Note that the LabJack will only check Z when it sees an edge on A or B.

Z-phase support requires Firmware 1.30 or later.

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 in some output timer mode. 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 was on that terminal, it goes to the state "digital-input" (which has a 100 kΩ pull-up to 3.3 volts). That means the best results are generally obtained if the terminal used by Timer0 was initially configured as digital 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 U3 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 U3, 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](#).

2.9.1.12 - Line-to-Line Measurement (Mode 14)

This timer mode requires firmware 1.30 or later.

Introduction:

The Line-to-Line timer mode uses two timers to measure the time between specified edges on two different lines. For instance, you can measure the time between a rising edge on Timer0 and a falling edge on Timer1. When the LabJack sees the specified edge on Timer0 it starts counting until it sees the specified edge on Timer1. High resolution up to 20.8ns can be achieved with this mode.

Configuring:

To configure a LabJack for Line-to-Line mode set an even timer and the next (odd) timer to mode 14. The timer values determine the edge that the timer will respond to, 1 being rising, 0 being falling. So, if Timer0's value is 0 and Timer1's is 1 then the LabJack will measure the time between a falling edge on Timer0 to a rising edge on Timer1.

Readings:

Once configured the timer will return zero until both specified edges have been detected. The time difference in TimerClock periods is then returned by both timers until they are reset. Both timers will return the same reading, so it is only necessary to read one or the other. To convert to time, divide the value returned by the timer clock. This mode returns 16-bit values, so care should be taken to be sure that the specified condition does not exceed the maximum time. The maximum time can be calculated by $(2^{16}-1)/\text{TimerClock}$.

Resetting:

Once a measurement has been acquired the even timer needs to be reset before the LabJack will measure again. Values specified when resetting have no effect. Once reset the even timer will return zero until a new measurement has been completed. Resetting the odd timer is optional, if not reset it will continue to return the last measurement until a new one has been completed.

2.9.1.12 - Line-to-Line (Mode 14)

2.9.2 - Timer Operation/Performance Notes

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

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

The edge-detecting timer input modes do require U3 processing resources, as an interrupt is required to handle each edge. Timer modes 2, 3, 5, 9, 12, and 13 must process every applicable edge (rising **or** falling). Timer modes 4 and 8 must process every edge (rising **and** falling). To avoid missing counts, keep the total number of processed edges (all timers) less than 30,000 per second (hardware V1.21). 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 (... and SCL/SDA/SCA)

The SPC terminal is used for manually resetting default values or jumping in/out of flash programming mode.

Hardware revision 1.20 and 1.21, had terminals labeled SCL, SDA, and/or SCA. On revision 1.20, these terminals did nothing except that SCL is used for the SPC functionality described above. On revision 1.21, these terminals were used for asynchronous functionality, and SCL is used for the SPC functionality described above. Note that these terminals never have anything to do with PC.

2.11 - DB15

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

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

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

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

All digital I/O on the U3 have 3 possible states: input, output-high, or output-low. Each bit of I/O can be configured individually. When configured as an input, a bit has a $\sim 100\text{ k}\Omega$ 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			
Table 2.11-1: DB15 Connector Pinouts				

2.11.1 - CB15 Terminal Board

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

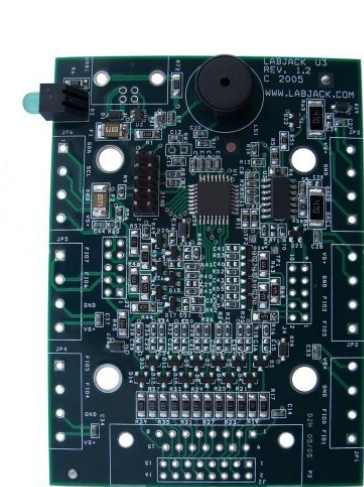
2.11.2 - RB12 Relay Board

The [RB12 relay board](#) provides a convenient interface for the U3 to industry standard digital I/O modules, allowing electricians,

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

2.12 - U3-OEM

There is an OEM version of the U3 available (LV and HV). It is a board only (no enclosure, no screwdriver, no cable), and does not have most of the through-hole components installed. The picture below shows how the U3-OEM ships by default. Leaving the through-hole parts off makes the OEM board very flexible. Many applications do not need the through-hole parts, but if needed they are much easier to install than uninstall.



The U3 PCB has alternate holes available for standard 0.1" pin-header installation. Example connectors are the Digikey WM268xx (where, for example, xx is 16 for the 2x8 header).

The 2x5 header JTAG is normally installed on the U3 and U3-OEM. This header is for factory use. You can use a mating header for physical support purposes, but it should not connect to anything.

Connectors J3 & J4 provide pin-header access to the connections that would normally appear on the left and right screw-terminals. Connector J2 provides a pin-header alternative to the DB15 connector. All these connector holes are always present, but J2 is obstructed when the DB15 is installed. The idea is that an OEM can connect ribbon cables to the pin-headers, or even plug the U3 directly to the customers main board designed with mating pin-header receptacles. See Appendix B for connector coordinates on the PCB.

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	
Table 2.12-1. J2 Connector Pin-Headers					
J3					
1	FIO4		2	FIO5	
3	FIO6		4	FIO7	
5	VS		6	GND	
7	GND *		8	SPC **	
9	VS		10	GND	
* SDA on <1.30					
** SCL on <1.30					
Table 2.12-2. J3 Connector Pin-Headers					
J4					
1	FIO0		2	FIO1	
3	FIO2		4	FIO3	
5	VS		6	GND	
7	DAC0		8	DAC1	
9	VS		10	GND	
Table 2.12-3. J4 Connector Pin-Headers					

USB (USB1)

There are 4 holes for a standard Type-B USB connection (plus a couple large holes for mounting tabs). Looking at the bottom (solder-side) of the PCB with the USB/LED end of the PCB up, GND (pin 4, black wire) is in the upper-right corner. Then clockwise it goes Vbus (5 volts, lower-right, pin 1, red wire), D- (lower-left, pin 2, white wire), and D+ (upper-left, pin 3, green wire). If using a normal Type-B USB connector (such as the Digikey 609-1039), it must be installed on the component side of the PCB.

Alternative Power Supply

Generally 5 volt power is provided via the USB connector holes, and usually it is provided from the USB host. There are few reasons, if any, to power the U3 from anything besides the USB host. The only valid reason we hear has to do with keeping the U3 powered even when the USB host loses power, which is an unusual requirement in itself since the U3 does not really do anything without a USB host connected. If you are considering an external supply for reasons related to noise or stability, you are probably "barking up the wrong tree" and should contact support@labjack.com.

The power supply provided by USB is typically 5 volts +/-5% @500 mA. The basic way to use an alternate supply is connecting it to hole 1 of the USB connector holes, instead of the supply from the USB host. Or if using a USB cable, cut the red wire inside the cable and connect your positive supply lead to that (also might need a connection of the negative supply lead to the black wire but

don't cut it). You can also connect an external supply to VS/GND screw-terminals (after cutting the red wire in the USB cable), but it is preferable to bring the supply in through the USB connector.

Note that USB ground and the external supply common/negative/ground must both connect to GND on the U3 (which could mean just the ground in the USB cable is needed if the power supply is already connected to that same ground). Also note that you never want 2 supplies connected directly to each other without any mechanism to prevent one supply from backfeeding the other.

2.13 - Hardware Revision Notes

U3A = Revision 1.20
U3B = Revision 1.21
U3C = Revision 1.30

Starting September of 2006, all U3 shipments changed from hardware revision 1.20 to 1.21. Following are the some of the main changes in revision 1.21:

- The default timer clock frequency is 48 MHz.
- All TimerBaseClock frequencies are twice the previous frequencies.
- The input timer edge limit is now 30,000 edges/second, compared to the old limit of 10,000 edges/second.
- Stream mode is now supported. See Section 3.2.
- Other new functions are supported, including Watchdog, SPI, Asynch, I2C, and SHT1X.
- Typical supply current is 50 mA.

Revision 1.20 can be upgraded to 1.21 by LabJack for a small fee. For information about upgrading a rev 1.20 unit, contact LabJack Corporation.

Hardware revision 1.30 was released in mid-March 2008 with 2 variations: U3-LV and U3-HV. The U3-LV is the most compatible with the previous U3, and the only changes possibly affecting backwards compatibility are:

- Timers/Counters cannot appear on FIO0-3. TimerCounterPinOffset must be 4-8. A value of 0-3 will result in an error. This error can be suppressed by a power-up default setting in LJControlPanel. If suppressed, a 0-3 will result in an offset of 4.
- The 3.66 reference voltage is no longer available on the REF/DAC1 terminal.
- There is no longer a buzzer.
- SDA terminal is gone. SCL terminal changed to SPC.
- UART (Asynch functionality) no longer uses SDA and SPC terminals, but rather uses terminals dynamically assigned after timers and counters. Also, the BaudFactor is different.

Other changes:

- Analog outputs are now specified for 10-bit resolution and DAC1 is always enabled. The higher resolution is available with a new IOType in the low-level Feedback function, which the high-level UD driver uses automatically. This causes the DACs to have more noise when the timer clock is decreased from the default of 48 MHz, so there is a compatibility option available in LJControlPanel to use 8-bit DACs.
- On the U3-HV only, the first four flexible I/O are fixed as analog inputs (AIN0-AIN3), and have scaling such that the input range is ± 10 volts normally, and +20 to -10 volts when using the "Special" range. The input impedance of these four lines is roughly 1 M Ω , which is good, but less than the normal low voltage analog inputs. Analog/digital configuration and all other digital operations on these pins are ignored. FIO4-EIO7 are still available as flexible I/O, same as the U3-LV.

Revision 1.20/21 U3s cannot be upgraded to 1.30.

3 - Operation

The following sections discuss command/response mode and stream mode.

Command/response mode is where communication is initiated by a command from the host which is followed by a response from the LabJack. Command/response is generally used at 1000 scans/second or slower and is generally simpler than stream mode.

Stream mode 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 automatically by the LabJack, they are placed in a buffer on the LabJack, until retrieved by the host. Stream mode is generally used at 10 scans/second or faster.

Command/response mode is generally best for minimum-latency applications such as feedback control. By latency here we mean the time from when a reading is acquired to when it is available in the host software. A reading or group of readings can be acquired in times on the order of a millisecond.

Stream mode is generally best for maximum-throughput applications where latency is not so important. Data is acquired very fast, but to sustain the fast rates it must be buffered and moved from the LabJack to the host in large chunks. For example, a typical stream application might set up the LabJack to acquire a single analog input at 50,000 samples/second. The LabJack moves this data to the host in chunks of 25 samples each. The Windows UD driver moves data from the USB host memory to the UD driver memory in chunks of 2000 samples. The user application might read data from the UD driver memory once a second in a chunk of 50,000 samples. The computer has no problem retrieving, processing, and storing, 50k samples once per second, but it could not do that with a single sample 50k times per second.

3.1 - Command/Response

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

For everything besides pin configuration, the low-level Feedback function is the primary function used, as it writes and reads virtually all I/O on the U3. The Windows 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.

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 include everything (Windows latency, UD driver overhead, communication time, U3 processing time, etc.).

	USB high-high	USB other	
# AIN	[milliseconds]	[milliseconds]	
0	0.6	4	<- Write/Read all DIO, DACs, Timers and Counters
1	1	4	
4	2.4	4	
8	4.7	9.2	
16	8.3	12.2	
Table 3.1-1. Typical Feedback Function Execution Times (QuickSample=0, LongSettling=0)			
	USB high-high	USB other	
# AIN	[milliseconds]	[milliseconds]	
0	0.6	4	<- Write/Read all DIO, DACs, Timers and Counters
1	0.7	4	
4	1	4	
8	2.1	8	
16	3	8	
Table 3.1-2. Typical Feedback Function Execution Times (QuickSample=1, LongSettling=0)			
	USB high-high	USB other	
# AIN	[milliseconds]	[milliseconds]	
0	0.6	4	<- Write/Read all DIO, DACs, Timers and Counters
1	4.2	5.2	
4	16	17	
8	31	36	
16	60	62	
Table 3.1-3. Typical Feedback Function Execution Times (QuickSample=0, LongSettling=1)			

A “USB high-high” configuration means the U3 is connected to a high-speed USB2 hub which is then connected to a high-speed USB2 host. Even though the U3 is not a high-speed USB device, such a configuration does provide improved performance.

The analog inputs have a QuickSample option where each conversion is done faster at the expense of increased noise. This is enabled by passing a nonzero value for put_config special channel *LJ_chAIN_RESOLUTION*. There is also a LongSettling option where additional settling time is added between the internal multiplexer configuration and the analog to digital conversion. This allows signals with more source impedance, and is enabled by passing a nonzero value for put_config special channel *LJ_chAIN_SETTLING_TIME*. Both of these options are disabled by default, so the first table above shows the default conditions.

The first row in each of the above tables (# AIN = 0) includes a write and read to all I/O on the U3 besides analog inputs (digital I/O, DACs, timers, and counters). The times in other rows basically consist of that fixed overhead plus the time per analog input channel, so times can be interpolated for other numbers of channels.

How about for 2 analog input channels with QuickSample=0 and LongSettling=0? You know that 1 channel takes about 1.0 ms and 4 channels takes about 2.4 ms. That means it takes about $(2.4-1.0)/(4-1) = 0.46$ ms/channel plus overhead of about 0.6 ms, so 2 channels would take about $(2*0.46)+0.6 = 1.5$ ms.

How about for 20 channels? This is a little different because the commands and/or responses for 20 channels can't fit in one low-level packet. From Section 5.2.5, the Feedback command has room for 57 bytes of command data and 55 bytes of response data. From Section 5.2.5.1, the AIN low-level IOType has 3 command bytes and 2 response bytes. That means the low-level Feedback command can hold 19 commands and 27 responses. Thus the commands are limiting and 19 channels is the most we can get into 1 low-level Feedback packet. The timing for 20-channels can then be calculated as a 19-channel read plus a subsequent 1-channel read. If you do an Add/Go/Get block with 20 channels the UD driver will split it like that.

The tables above were measured with U3 hardware version 1.21 which started shipping in late August of 2006. The times could be up to twice as long with hardware version 1.20 or less.

3.2 - Stream Mode

The highest input data rates are obtained in stream mode, which is supported with U3 hardware version 1.21 or higher. Hardware version 1.21 started shipping in late August of 2006. Contact LabJack for information about upgrading older U3s. **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 U3, 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.

Since the data buffer on the U3 is very small it uses a feature called **auto-recovery**. If the buffer overflows, the U3 will continue streaming but discard data until the buffer is emptied, and then data will be stored in the buffer again. The U3 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 in the last row of the table. A “USB high-high” configuration means the U3 is connected to a high-speed USB2 hub which is then connected to a high-speed USB2 host. Even though the U3 is not a high-speed USB device, such a configuration does 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.

Low-Level	UD	Max Stream	ENOB	ENOB	Noise	Interchannel
Res Index	Res Index	(Samples/s)	(RMB)	(Noise-Free)	(Counts)	Delay (us)
0	100	2500	12.8	10	±2	320
1	101	10000	11.9	9	±4	82
2	102	20000	11.3	8.4	±6	42
3	103	50000	10.5	7.5	±11	12.5
Table 3.2-1. Streaming at Various Resolutions						

Full resolution streaming is limited to 2500 samples/s, but higher speeds are possible at the expense of reduced effective resolution (increased noise). The first column above is the index passed in the Resolution parameter to the low-level StreamConfig function, while the second column is the corresponding index for the Resolution parameter in the UD driver. In the UD driver, the default Resolution index is 0, which corresponds to automatic selection. In this case, the driver will use the highest resolution for the specified sample rate.

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 128 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 128 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 12-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
194	CIO
200	Timer0
201	Timer1
210	Counter0
211	Counter1
224	TC_Capture
230	Timer0 with reset
231	Timer1 with reset
240	Counter0 with reset
241	Counter1 with reset

Special Channels:

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.

194: Returns the input states of 16 bits of digital I/O. CIO is the lower 8 bits.

200-201 and 210-211: Retrieves 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).

230-231 and 240-241: These channels perform the same operation as their 200-211 counterpart above, then reset the timer or counter.

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, counter, or analog inputs, 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 U3, 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 U3 functions are described in [Section 5](#), but most Windows applications will use the LabJackUD driver instead.

The latest version of the driver requires a PC running Windows XP, Vista, or 7. [The 3.07 version](#), supports Windows 98, ME, 2000. It is recommended to install the software before making a USB connection to a LabJack.

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

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

4.1 - Overview

The general operation of the LabJackUD functions is as follows:

- Open a LabJack.
- Build a list of requests to perform (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 U3
//over USB and get a handle to the device.
//The general form of the open function is:
//OpenLabJack (DeviceType, ConnectionType, Address, FirstFound, *Handle)

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

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 read from AIN3 (FIO3), assuming it has been enabled as
//an analog line.
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 U3 is already open):

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

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

In the case of the U3, the first example using add/go/get handles both the DAC command and AIN read in a single low-level call, while in the second example using ePut/eGet two low-level commands are used. Examples in the following documentation will use both the add/go/get method and the 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_chHARDWARE_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_chHARDWARE_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_chHARDWARE_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

The UD 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 `LJ_NO_DATA_AVAILABLE` or a similar error, chances are the requests and results are in different threads.

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

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

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

4.2 - Function Reference

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

Some parameters are common to many functions:

- **LJ_ERROR** – A LabJack specific numeric 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_dtU3" 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_dtU3" 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 only for the U3.
- **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.

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 LJ_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 single-ended channels on the U3, this parameter should be 31 or 199 (see Section 2.6.1).
- **Range** – Ignored on the U3.
- **Resolution** – Pass a nonzero value to enable QuickSample.
- **Settling** – Pass a nonzero value to enable LongSettling.
- **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.

When needed (on hardware revisions <1.30 perhaps), this function automatically enables the specified analog output.

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. 0-19 corresponds to FIO0-CIO3.

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. 0-19 corresponds to FIO0-CIO3.
- **State** – The state to write to the digital output. 0=False=Low and 1=True=High.

4.2.21 - eTCCConfig()

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 eTCCConfig (  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 U3, this array must always have at least 2 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 U3, this array must always have at least 2 elements.
- **TCPinOffset** – Value from 4-8 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 U3, this array must always have at least 2 elements.
- **aTimerValues** – An array where each element specifies the initial value for that timer. For the U3, this array must always have at least 2 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 U3, this array must always have at least 2 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 U3, this array must always have at least 2 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 U3, 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 U3, 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 U3, this array must always have at least 2 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 U3 is:

```
LJ_dtU3
```

There is only one valid ConnectionType for the U3:

```
LJ_ctUSB
```

Following is example pseudocode to open a U3 over USB:

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

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

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

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

Talking to multiple devices from a single application is no problem. Make multiple open calls to get a handle to each device and be sure to set FirstFound=FALSE:

```
//Open U3s with Local ID #2 and #3.  
OpenLabJack (LJ_dtU3, LJ_ctUSB, "2", FALSE, &lngHandleA);  
OpenLabJack (LJ_dtU3, LJ_ctUSB, "3", FALSE, &lngHandleB);
```

... then when making further calls use the handle for the desired device.

4.3.2 - Configuration

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

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

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

```
//Configure FIO3 as an analog input.  
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_BIT, 3, 1, 0);  
  
//Configure FIO3 as digital I/O.  
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_BIT, 3, 0, 0);  
  
//Configure FIO0-FIO2 and EIO0-EIO7 as analog, all others as digital. That  
//means a starting channel of 0, a value of b1111111100000111 (=d65287), and  
//all 16 bits will be updated.  
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_PORT, 0, 65287, 16);  
  
//Configure FIO2-FIO4 as analog, and FIO5-FIO6 as digital, without  
//configuring any other bits. That means a starting channel of 2,  
//a value of b00111 (=d7), and 5 bits will be updated.  
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_PORT, 2, 7, 5);
```

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

```
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 0, 0);  
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 4, 0);  
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tc48MHZ, 0);  
ePut (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 0, 0);  
ePut (lngHandle, LJ_ioPUT_COUNTER_ENABLE, 0, 0, 0);  
ePut (lngHandle, LJ_ioPUT_COUNTER_ENABLE, 1, 0, 0);  
ePut (lngHandle, LJ_ioPUT_DAC_ENABLE, 1, 0, 0); //Ignored on hardware rev 1.30+.  
ePut (lngHandle, LJ_ioPUT_ANALOG_ENABLE_PORT, 0, 0, 16);
```

This disables all timers and counters, sets the timer/counter pin offset to 4, sets the timer clock base to 48 MHz (no divisor), sets the timer clock divisor to 0, and sets all flexible I/O to digital. A simpler option is using the following IOType created exactly for this purpose, which does the same thing as the 8 function calls above:

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

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

```
LJ_ioPUT_CONFIG  
LJ_ioGET_CONFIG
```

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

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

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

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

```
eGet (lngHandle, LJ_ioGET_CONFIG, LJ_chLOCALID, &dblValue, 0);
```

4.3.3 - Analog Inputs

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

```
LJ_ioGET_AIN           //Single-ended. Negative channel is fixed as 31/199.
LJ_ioGET_AIN_DIFF     //Specify negative channel in x1.
```

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

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

Following is example pseudocode to read analog inputs:

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

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

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

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

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

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

//Request a single-ended read of AIN1.
AddRequest (lngHandle, LJ_ioGET_AIN_DIFF, 1, 0, 199, 0);

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

//Execute the requests.
GoOne (lngHandle);

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

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

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

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

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

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

4.3.4 - Analog Outputs

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

```
LJ_ioPUT_DAC
```

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

```
LJ_ioPUT_DAC_ENABLE //Ignored on hardware rev 1.30+, as DAC1 always enabled.
LJ_ioGET_DAC_ENABLE
```

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

```
LJ_chDAC_BINARY
```

Following is example pseudocode to set DAC0 to 2.5 volts:

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

4.3.5 - Digital I/O

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

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

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

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

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

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

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

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

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

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

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

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

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

//Execute the requests.
GoOne (lngHandle);

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

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

4.3.6 - Timers & Counters

There are eight IOTypes used to write or read 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_tmSYSTEMERLOW	//System timer low read (no FIO)
LJ_tmSYSTEMERHIGH	//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	//0-2
LJ_chTIMER_CLOCK_BASE	//Value constants below
LJ_chTIMER_CLOCK_DIVISOR	//0-255, where 0=256
LJ_chTIMER_COUNTER_PIN_OFFSET	//4-8 only starting with hardware rev 1.30.

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:

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

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

//Set the pin offset to 4, which causes the timers to start on FIO4.
AddRequest (lngHandle, LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 4, 0, 0);

//Enable both timers. They will use FIO4-FIO5
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, FIO6.
```

```

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. Note that this timer
//clock base is not valid with U3 hardware version 1.20.
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 e-functions are used in the following pseudocode, but some applications might combine the following calls into a single add/go/get block so that a single low-level call is used.

```

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

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

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

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

```

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

```

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

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

```

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

4.3.7 - Stream Mode

The highest input data rates are obtained in stream mode, which is supported with U3 hardware version 1.21 or higher. See Section 3.2 for more information about stream mode.

There are five IOTypes 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
```

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 U3 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 U3 itself. The U3 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 U3 buffer (*COMM* or *CONTROL* are the same thing on the U3), where 0 means 0% full and 256 would mean 100% full. The UD driver retrieves stream data from the U3 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 U3 is acquiring it, and thus there will be data left over in the U3 buffer.

To obtain the maximum stream rates documented in Section 3.2, the data must be transferred between host and U3 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 U3 is very small a feature called auto-recovery is used. If the buffer overflows, the U3 will continue streaming but discard data until the buffer is emptied, and then data will be stored in the buffer again. The U3 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 U3 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 AIN3-AIN9.
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, 3, 0, 9, 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 U3 backlog. The UD driver retrieves
    //stream data from the U3 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 U3 is acquiring it, and thus there will
    //be data left over in the U3 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.

```

LJ_ioRAW_OUT
LJ_ioRAW_IN

```

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

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

```

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

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

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

```

4.3.9 - 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, 31, &dblVoltage, 0, 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, 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}; //Enable Timer0-Timer1
alngTimerModes = {LJ_tmPWMS,LJ_tmRISINGEDGES32}; //Set timer modes
adblTimerValues = {16384,0}; //Set PWMS 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}; //Read Timer1
alngUpdateResetTimers = {1,1}; //Update Timer0 and reset Timer1
alngReadCounters = {1,0}; //Read Counter0
alngResetCounters = {1,0}; //Reset Counter0
adblTimerValues = {32768,0}; //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 U3 (hardware version 1.21+ only) 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 U3 over USB, and the U3 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 5.2.15](#) 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 U3 (hardware version 1.21+ only) supports Inter-Integrated Circuit (PC or I2C) communication as the master only. PC is a synchronous serial protocol typically used to communicate with chips that support PC as slave devices. Any 2 digital I/O lines are used for SDA and SCL. Note that the PC 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 PC.

This serial link is not an alternative to the USB connection. Rather, the host application will write/read data to/from the U3 over USB, and the U3 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 PC data:

```
LJ_ioI2C_COMMUNICATION
```

The following are special channels used with the PC 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 PC bus. See the low-level function description in [Section 5.2.19](#) 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 PC 24C01C EEPROM chip. Following is example pseudocode to configure PC 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 U3 (hardware version 1.21+ only) has a UART available that supports asynchronous serial communication. On hardware version 1.30 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 4, TX=FIO4 and RX=FIO5. On hardware version 1.21, the UART uses SDA for TX and SCL for RX.

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 U3 over USB, and the U3 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 to the following formula: $\text{BaudFactor16} = 2^{16} - 48000000 / (2 * \text{Desired Baud})$. For example, a $\text{BaudFactor16} = 63036$ provides a baud rate of 9600 bps. With hardware revision 1.21, the value is only 8-bit and the formula is $\text{BaudFactor8} = 2^8 - \text{TimerClockBase} / (\text{Desired Baud})$.

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 U3 (hardware version 1.21+ only) has firmware based watchdog capability. Unattended systems requiring maximum up-time might use this capability to reset the U3 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 U3 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 U3 with a short from FIO6 to SPC (FIO2 to SCL on U3 1.20/1.21), 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 // x1 points to an array with 64 doubles.
LJ_chUSER_MEM // x1 points to an array with 256 bytes.
```

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.
```

Any value (in microseconds) from 0-8388480 can be passed, but the actual resolution is 128 microseconds. On hardware V1.20, the delay and resolution are 2x.

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 recieved 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	
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_PWMSTAOP_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 stream.
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 4.4-1. Request Level Errorcodes		
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() call 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 4.4-2. Group Level Errorcodes		

Table 4-1 lists the errors which are specific to a request. For example, LJE_INVALID_CHANNEL_NUMBER. If this error occurs, other requests are not affected. Table 4-2 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 U3. These are commands sent over USB directly to the processor on the U3.

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 U3 low-level communication protocol. There are two types of commands:

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

Extended: 3 command words plus 0-125 data words.

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

Normal command format:

	Byte				
	0	Checksum8: Includes bytes 1-15.			
	1	Command Byte: DCCCWWW			
		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:

Extended Command Format:					
	Byte				
	0	Checksum8: Includes bytes 1-5.			
	1	Command Byte: D111WWWW			
		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:

1. Get the subarray consisting of bytes 1 and up.
2. Convert bytes to U16 and sum into a U16 accumulator.
3. Divide by 2^8 and sum the quotient and remainder.
4. 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:

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

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

1. Get the subarray consisting of bytes 1 through 5.
2. Convert bytes to U16 and sum into a U16 accumulator.
3. Divide by 2^8 and sum the quotient and remainder.
4. 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 U3.

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 ($2^0 + \dots + 2^7$), FIO0-FIO7 are output.

5.2 - Low-Level Functions

5.2.1 - Bad Checksum

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

Response:	
Byte	
0	0xB8
1	0xB8

5.2.2 - ConfigU3

Writes and reads various configuration settings. Although this function has many of the same parameters as other functions, most parameters in this case are affecting the power-up values, not the current values. There is a hardware method to restore bytes 9-20 to the factory default value of 0x00: Power up the U3 with a short from FIO6 to SPC (FIO2 to SCL on U3 1.20/1.21), then remove the jumper and power cycle the device again.

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

Note: If the stream is running, you cannot update any of the values (WriteMask must equal 0).

Command:	
Byte	
0	Checksum8
1	0xF8
2	0x0A
3	0x08
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	WriteMask0
	Bit 3: LocalID
	Bit 2: DAC Defaults
	Bit 1: Digital I/O Defaults
	Bit 0: Reserved
7	WriteMask1 (Reserved)
8	LocalID
9	TimerCounterConfig
	Bits 4-7: TimerCounterPinOffset
	Bit 3: Enable Counter1
	Bit 2: Enable Counter0
	Bit 0-1: Number of timers enabled
10	FIOAnalog
11	FIODirection
12	FIOState
13	EIOAnalog
14	EIODirection
15	EIOState
16	CIODirection
17	CIOState
18	DAC1Enable
19	DAC0
20	DAC1
21	TimerClockConfig
22	TimerClockDivisor (0 = +256)
23	CompatibilityOptions
24	0x00
25	0x00
Response:	
Byte	
0	Checksum8
1	0xF8
2	0x10
3	0x08
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	Reserved
8	Reserved
9-10	FirmwareVersion
11-12	BootloaderVersion
13-14	HardwareVersion
15-18	SerialNumber
19-20	ProductID
21	LocalID
22	TimerCounterMask
23	FIOAnalog
24	FIODirection
25	FIOState
26	EIOAnalog
27	EIODirection
28	EIOState
29	CIODirection
30	CIOState
31	DAC1Enable
32	DAC0
33	DAC1
34	TimerClockConfig
35	TimerClockDivisor (0 = +256)
36	CompatibilityOptions
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. Note that reads return reset defaults, not necessarily current values (except for LocalID). For instance, the value returned by FIODirection is the directions at reset, not necessarily the current directions.
- **LocalID:** If the WriteMask bit 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.
- **TimerCounterConfig:** If the WriteMask bit 1 is set, the value passed becomes the default value, meaning it is written to flash and used at reset. The return value of this parameter is a read of the power-up default. See [Section 5.2.3](#).
- **FIOEIO/CIO:** If the WriteMask bit 1 is set, the values passed become the default values, meaning they are written to flash and used at reset. Regardless of the mask bit, this function has no effect on the current settings. The return value of these parameters are a read of the power-up defaults.

- **DAC:** If the WriteMask bit 2 is set, the values passed become the default values, meaning they are written to flash and used at reset. Regardless of the mask bit, this function has no effect on the current settings. The return values of these parameters are a read of the power-up defaults.
- **TimerClockConfig & TimerClockDivisor:** If the WriteMask bit 4 is set, the values passed become the default values, meaning they are written to flash and used at reset. The return values of these parameters are a read of the power-up defaults. See [Section 5.2.4](#).
- **CompatibilityOptions:** If the WriteMask bit 5 is set, the value passed becomes the default value, meaning it is written to flash and used at reset. The return value of this parameter is a read of the power-up default. If bit 0 is set, Timer Counter Pin Offset errors are ignored. If bit 1 is set, all DAC operations will use 8-bit mode rather than 10-bit mode. Once this value has been changed the U3 will need to be restarted before the new setting will take affect.
- **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:** (3) Fixed parameter identifies this LabJack as a U3.
- **VersionInfo:** Bit 0 specifies U3B. Bit 1 specifies U3C and if set then bit 4 specifies -HV version.

5.2.3 - ConfigIO

Writes and reads the current IO configuration.

Command:			
Byte			
0	Checksum8		
1	0xF8		
2	0x03		
3	0x0B		
4	Checksum16 (LSB)		
5	Checksum16 (MSB)		
6	WriteMask		
		Bit 5: Write UART Related settings	
		Bit 4: Reserved, Pass 0	
		Bit 3: EIOAnalog	
		Bit 2: FIOAnalog	
		Bit 1: DAC1Enable	
		Bit 0 : TimerCounterConfig	
7	Reserved		
8	TimerCounterConfig		
		Bits 4-7: TimerCounterPinOffset	
		Bit 3: Enable Counter1	
		Bit 2: Enable Counter0	
		Bits 0-1: Number of timers enabled	
9	DAC1Enable (ignored on hardware rev 1.30+)		
		Bit 2: Assign UART Pins (HW 1.30 only)	
		Bit 1: Reserved, Pass 0	
		Bit 0: Enable DAC1	
10	FIOAnalog		
11	EIOAnalog		
Response:			
Byte			
0	Checksum8		
1	0xF8		
2	0x03		
3	0x0B		
4	Checksum16 (LSB)		
5	Checksum16 (MSB)		
6	Errorcode		
7	Reserved		
8	TimerCounterConfig		
9	DAC1Enable		
10	FIOAnalog		
11	EIOAnalog		

- **WriteMask:** Has bits that determine which, if any, of the parameters will be written.
- **TimerCounterConfig:** Used to enable/disable timers and counters. Timers/counters will be assigned to IO pins starting with FIO0 plus TimerCounterPinOffset (4-8 only starting with hardware revision 1.30). Timer0 takes the first IO pin, then Timer1, Counter0, and Counter1. Whenever this function is called and timers are enabled, the timers are initialized to mode 10, so the desired timer mode must always be specified after every call to this function. Note that Counter0 is not available when using a timer clock base that supports a timer clock divisor (TimerClockBase = 3-6).
- **Assign UART Pins:** On hardware 1.30 setting this bit will assign IO lines to the UART module. This setting will be ignored unless the UART write bit is set in the WriteMask byte.
- **DAC1Enable:** On hardware revisions 1.20/1.21 only, bit 0 enables DAC1. When DAC1 is disabled, it outputs a constant voltage of 1.5 times the internal Vref (~2.44 volts). When DAC1 is enabled, the internal Vref is not available for the analog inputs and Vreg (~3.3 volts) is used as the AIN reference. Starting with hardware revision 1.30, DAC1 is always enabled.
- **FIOAnalog:** Each bit determines whether that bit of FIO is analog input (=1) or digital I/O (=0).
- **EIOAnalog:** Each bit determines whether that bit of EIO is analog input (=1) or digital I/O (=0).

LabJackPython Example

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
'''Assign timer0 to FIO6, set FIO4, FIO5 as analog input, and set EIO1, EIO2 as analog input'''
>>> d.configIO(TimerCounterPinOffset = 6, NumberOfTimersEnabled = 1, FIOAnalog = 48, EIOAnalog = 3)
Sent: [0xa8, 0xf8, 0x3, 0xb, 0xa1, 0x0, 0xd, 0x0, 0x61, 0x0, 0x30, 0x3]
Result: [0x9b, 0xf8, 0x3, 0xb, 0x94, 0x0, 0x0, 0x0, 0x61, 0x0, 0x30, 0x3]
{'NumberOfTimersEnabled': 1, 'TimerCounterPinOffset': 6, 'DAC1Enable': 0, 'FIOAnalog': 48, 'EIOAnalog': 3, 'TimerCounterConfig': 97, 'EnableCounter1': False, 'EnableCounter0': False}
```

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.

Name	IOType (dec)	WriteBytes	ReadBytes
AIN	1	3	2
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
Counter0	54	2	4
Counter1	55	2	4
Buzzer	63	6	0

5.2.5.1 - AIN: IOType = 1

AIN_3 Command Bytes:	
0	IOType = 1
1	Bits 4-0: Positive Channel
	Bit 6: LongSettling
	Bit 7: QuickSample
2	Negative Channel
2 Response Bytes:	
0	AIN LSB
1	AIN MSB

This IOType returns a single analog input reading.

Note: Do **not** use this IO type while streaming.

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

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

```

Analog Input Feedback command

specify the positive and negative channels to use
(0-16, 30 and 31 are possible)
also specify whether to turn on longSettle or quick Sample

returns 16-bit signed int sample

>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.AIN(PositiveChannel = 0, NegativeChannel=31, LongSettling=False)
Sent:  [0x1b, 0xf8, 0x2, 0x0, 0x20, 0x0, 0x0, 0x1, 0x0, 0x1f]
Response:  [0xab, 0xf8, 0x3, 0x0, 0xaf, 0x0, 0x0, 0x0, 0x0, 0x20, 0x8f, 0x0]
[36640]

```

[This Gist](#) brought to you by [GitHub](#). [u3-feedback-AIN.txt](#) [view raw](#)

5.2.5.2 - 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 (256 on hardware V1.20) to determine the delay.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

```

WaitShort Feedback command

specify the number of 128us time increments to wait

>>> import u3

```



```
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.WaitShort(Time = 9))
Sent: [0x9, 0xf8, 0x2, 0x0, 0xe, 0x0, 0x0, 0x5, 0x9, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-WaitShort.txt](#) [view raw](#)

5.2.5.3 - 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 (64 on hardware V1.20) to determine the delay.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

WaitLong Feedback command

specify the number of 32ms time increments to wait

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.WaitLong(Time = 70))
Sent: [0x47, 0xf8, 0x2, 0x0, 0x4c, 0x0, 0x0, 0x6, 0x46, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-WaitLong.txt](#) [view raw](#)

5.2.5.4 - 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.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

LED Toggle

specify whether the LED should be on or off by truth value

1 or True = On, 0 or False = Off

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.LED(State = False))
Sent: [0x4, 0xf8, 0x2, 0x0, 0x9, 0x0, 0x0, 0x9, 0x0, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
>>> d.getFeedback(u3.LED(State = True))
Sent: [0x5, 0xf8, 0x2, 0x0, 0xa, 0x0, 0x0, 0x9, 0x1, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-LED.txt](#) [view raw](#)

5.2.5.5 - BitStateRead: IOType=10

BitStateRead, 2 Command Bytes:		
0	IOType = 10	
1	Bits 0-4: IONumber	
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.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

BitStateRead Feedback command

read the state of a single bit of digital I/O. Only digital lines return valid readings.

IONumber: 0-7=FIO, 8-15=EIO, 16-19=CIO
return 0 or 1

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.BitStateRead(IONumber = 5))
Sent: [0xa, 0xf8, 0x2, 0x0, 0xf, 0x0, 0x0, 0xa, 0x5, 0x0]
Response: [0xfb, 0xf8, 0x2, 0x0, 0x1, 0x0, 0x0, 0x0, 0x0, 0x1]
[1]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-BitStateRead.txt](#) [view raw](#)

5.2.5.6 - BitStateWrite: IOType=11

BitStateWrite, 2 Command Bytes:		
0	IOType = 11	
1	Bits 0-4: IO Number	
	Bit 7: State	
0 Response Byte:		

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.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

BitStateWrite Feedback command

write a single bit of digital I/O. The direction of the specified line is forced to output.

IONumber: 0-7=FIO, 8-15=EIO, 16-19=CIO
State: 0 or 1

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.BitStateWrite(IONumber = 5, State = 0))
Sent: [0xb, 0xf8, 0x2, 0x0, 0x10, 0x0, 0x0, 0xb, 0x5, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-BitStateWrite.txt](#) [view raw](#)

5.2.5.7 - 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.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

Read the digital direction of one I/O

IONumber: 0-7=FIO, 8-15=EIO, 16-19=CIO
returns 1 = Output, 0 = Input

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.BitDirRead(IONumber = 5))
Sent: [0xc, 0xf8, 0x2, 0x0, 0x11, 0x0, 0x0, 0xc, 0x5, 0x0]
Response: [0xfb, 0xf8, 0x2, 0x0, 0x1, 0x0, 0x0, 0x0, 0x0, 0x1]
[1]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-BitDirRead.txt](#) [view raw](#)

5.2.5.8 - 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.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

```
BitDirWrite Feedback command

Set the digital direction of one I/O

IONumber: 0-7=FIO, 8-15=EIO, 16-19=CIO
Direction: 1 = Output, 0 = Input

>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.BitDirWrite(IONumber = 5, Direction = 0))
Sent: [0xd, 0xf8, 0x2, 0x0, 0x12, 0x0, 0x0, 0xd, 0x5, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#). [u3-feedback-BitDirWrite.txt](#) [view raw](#)

5.2.5.9 - PortStateRead: IOType=26

PortStateRead, 1 Command Byte:		
0	IOType = 26	
3 Response Bytes:		
0-2	State	

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

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

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

```
PortStateRead Feedback command
Reads the state of all digital I/O.

>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.PortStateRead())
Sent: [0x14, 0xf8, 0x1, 0x0, 0x1a, 0x0, 0x0, 0x1a]
Response: [0xeb, 0xf8, 0x3, 0x0, 0xee, 0x1, 0x0, 0x0, 0xe0, 0xff, 0xf]
[{'CIO': 15, 'FIO': 224, 'EIO': 255}]
```

[This Gist](#) brought to you by [GitHub](#). [u3-feedback-PortStateRead.txt](#) [view raw](#)

5.2.5.10 - 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 (b000000010000011100000111), State=d67335.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

```
PortStateWrite Feedback command

State: A list of 3 bytes representing FIO, EIO, CIO
WriteMask: A list of 3 bytes, representing which to update.
The Default is all ones.

>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.PortStateWrite(State = [0xab, 0xcd, 0xef], WriteMask = [0xff, 0:
```

```
Sent: [0x81, 0xf8, 0x4, 0x0, 0x7f, 0x5, 0x0, 0x1b, 0xff, 0xff, 0xff, 0xab, 0xcd, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-PortStateWrite.txt](#) [view raw](#)

5.2.5.11 - PortDirRead: IOType=28

PortDirRead, 1 Command Byte:		
0	IOType = 28	
3 Response Bytes:		
0-2	Direction	

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

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

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

```
PortDirRead Feedback command
Reads the direction of all digital I/O.

>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.PortDirRead())
Sent: [0x16, 0xf8, 0x1, 0x0, 0x1c, 0x0, 0x0, 0x1c]
Response: [0xfb, 0xf8, 0x3, 0x0, 0xfe, 0x1, 0x0, 0x0, 0x0, 0xf0, 0xff, 0xf]
[{'CIO': 15, 'FIO': 240, 'EIO': 255}]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-PortDirRead.txt](#) [view raw](#)

5.2.5.12 - 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 (b000000010000011100000111), Direction=d67335.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

```
PortDirWrite Feedback command

Direction: A list of 3 bytes representing FIO, EIO, CIO
WriteMask: A list of 3 bytes, representing which to update. Default is all ones.

>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.PortDirWrite(Direction = [0xaa, 0xcc, 0xff], WriteMask = [0xff,
Sent: [0x91, 0xf8, 0x4, 0x0, 0x8f, 0x5, 0x0, 0x1d, 0xff, 0xff, 0xff, 0xaa, 0xcc, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-PortDirWrite.txt](#) [view raw](#)

5.2.5.13 - 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.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

```
8-bit DAC Feedback command
```

Controls a single analog output

Dac: 0 or 1
Value: 0-255

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.DAC8(Dac = 0, Value = 0x55))
Sent: [0x72, 0xf8, 0x2, 0x0, 0x77, 0x0, 0x0, 0x22, 0x55, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-DAC8.txt](#) [view raw](#)

8-bit DAC Feedback command for DAC0

Controls DAC0 in 8-bit mode.

Value: 0-255

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.DAC0_8(Value = 0x33))
Sent: [0x50, 0xf8, 0x2, 0x0, 0x55, 0x0, 0x0, 0x22, 0x33, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-DAC0_8.txt](#) [view raw](#)

8-bit DAC Feedback command for DAC1

Controls DAC1 in 8-bit mode.

Value: 0-255

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.DAC1_8(Value = 0x22))
Sent: [0x40, 0xf8, 0x2, 0x0, 0x45, 0x0, 0x0, 0x23, 0x22, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-DAC1_8.txt](#) [view raw](#)

5.2.5.14 - 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.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

16-bit DAC Feedback command

Controls a single analog output

Dac: 0 or 1
Value: 0-65535

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.DAC16(Dac = 0, Value = 0x5566))
Sent: [0xdc, 0xf8, 0x2, 0x0, 0xe1, 0x0, 0x0, 0x26, 0x66, 0x55]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-DAC16.txt](#) [view raw](#)

16-bit DAC Feedback command for DAC0

Controls DAC0 in 16-bit mode.

Value: 0-65535

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.DAC0_16(Value = 0x1122))
Sent: [0x54, 0xf8, 0x2, 0x0, 0x59, 0x0, 0x0, 0x26, 0x22, 0x11]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-DAC0_16.txt](#) [view raw](#)

16-bit DAC Feedback command for DAC1

Controls DAC1 in 16-bit mode.

Value: 0-65535

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.getFeedback(u3.DAC1_16(Value = 0x2233))
Sent: [0x77, 0xf8, 0x2, 0x0, 0x7c, 0x0, 0x0, 0x27, 0x33, 0x22]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-DAC1_16.txt](#) [view raw](#)

5.2.5.15 - Timer#: IOType=42,44

Timer#. 4 Command Bytes:	
0	IOType = 42, 44
1	Bit 0: UpdateReset
2	Value LSB
3	Value MSB
4 Response Bytes:	
0	Timer LSB
1	Timer
2	Timer
3	Timer MSB

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

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

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

For reading the value of the Timer. It provides the ability to update/reset a given timer, and read the timer value.
(Section 5.2.5.14 of the User's Guide)

timer: Either 0 or 1 for timer 0 or timer 1

UpdateReset: Set True if you want to update the value

Value: Only updated if the UpdateReset bit is 1. The meaning of this parameter varies with the timer mode.

Mode: Set to the timer mode to handle any special processing. See classes QuadratureInputTimer and TimerStopInput1.

Returns an unsigned integer of the timer value, unless Mode has been specified and there are special return values. See Section 2.9.1 for expected return values.

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.configIO(NumberOfTimersEnabled = 1)
Sent: [0x49, 0xf8, 0x3, 0xb, 0x42, 0x0, 0x1, 0x0, 0x41, 0x0, 0x0, 0x0]
Response: [0x57, 0xf8, 0x3, 0xb, 0x50, 0x0, 0x0, 0x0, 0x41, 0x0, 0xf, 0x0]
{'NumberOfTimersEnabled': 1, 'TimerCounterPinOffset': 4, 'DAC1Enable': 0, 'FIOAnalog'
>>> d.getFeedback(u3.Timer(timer = 0, UpdateReset = False, Value = 0, Mode = None))
Sent: [0x26, 0xf8, 0x3, 0x0, 0x2a, 0x0, 0x0, 0x2a, 0x0, 0x0, 0x0, 0x0]
Response: [0xfc, 0xf8, 0x4, 0x0, 0xfe, 0x1, 0x0, 0x0, 0x0, 0x63, 0xdd, 0x4c, 0x72, 0x1917640035]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-Timer.txt](#) [view raw](#)

For reading the value of the Timer0. It provides the ability to update/reset Timer0, and read the timer value.
(Section 5.2.5.14 of the User's Guide)

UpdateReset: Set True if you want to update the value

Value: Only updated if the UpdateReset bit is 1. The meaning of this parameter varies with the timer mode.

Mode: Set to the timer mode to handle any special processing. See classes QuadratureInputTimer and TimerStopInput1.

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.configIO(NumberOfTimersEnabled = 1)
Sent: [0x49, 0xf8, 0x3, 0xb, 0x42, 0x0, 0x1, 0x0, 0x41, 0x0, 0x0, 0x0]
Response: [0x57, 0xf8, 0x3, 0xb, 0x50, 0x0, 0x0, 0x0, 0x41, 0x0, 0xf, 0x0]
{'NumberOfTimersEnabled': 1, 'TimerCounterPinOffset': 4, 'DAC1Enable': 0, 'FIOAnalog'
>>> d.getFeedback(u3.Timer0(UpdateReset = False, Value = 0, Mode = None))
Sent: [0x26, 0xf8, 0x3, 0x0, 0x2a, 0x0, 0x0, 0x2a, 0x0, 0x0, 0x0, 0x0]
Response: [0x51, 0xf8, 0x4, 0x0, 0x52, 0x2, 0x0, 0x0, 0x0, 0xf6, 0x90, 0x46, 0x86, 0x2252771574]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-Timer0.txt](#) [view raw](#)

For reading the value of the Timer1. It provides the ability to update/reset Timer1, and read the timer value.
(Section 5.2.5.14 of the User's Guide)

UpdateReset: Set True if you want to update the value

Value: Only updated if the UpdateReset bit is 1. The meaning of this parameter varies with the timer mode.

Mode: Set to the timer mode to handle any special processing. See classes QuadratureInputTimer and TimerStopInput1.

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.configIO(NumberOfTimersEnabled = 2)
Sent: [0x4a, 0xf8, 0x3, 0xb, 0x43, 0x0, 0x1, 0x0, 0x42, 0x0, 0x0, 0x0]
Response: [0x58, 0xf8, 0x3, 0xb, 0x51, 0x0, 0x0, 0x0, 0x42, 0x0, 0xf, 0x0]
{'NumberOfTimersEnabled': 2, 'TimerCounterPinOffset': 4, 'DAC1Enable': 0, 'FIOAnalog': 0}
>>> d.getFeedback(u3.Timer1(UpdateReset = False, Value = 0, Mode = None))
Sent: [0x28, 0xf8, 0x3, 0x0, 0x2c, 0x0, 0x0, 0x2c, 0x0, 0x0, 0x0, 0x0]
Response: [0x8d, 0xf8, 0x4, 0x0, 0x8e, 0x2, 0x0, 0x0, 0x0, 0xf3, 0x31, 0xd0, 0x9a, 0x0, 0x0, 0x0, 0x0]
[259733539]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-Timer1.txt](#) [view raw](#)

For reading Quadrature input timers. They are special because their values are signed.

(Section 2.9.1.8 of the User's Guide)

Args:

UpdateReset: Set True if you want to reset the counter.

Value: Set to 0, and UpdateReset to True to reset the counter.

Returns a signed integer.

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.configIO(NumberOfTimersEnabled = 2)
Sent: [0x4a, 0xf8, 0x3, 0xb, 0x43, 0x0, 0x1, 0x0, 0x42, 0x0, 0x0, 0x0]
Response: [0x58, 0xf8, 0x3, 0xb, 0x51, 0x0, 0x0, 0x0, 0x42, 0x0, 0xf, 0x0]
{'NumberOfTimersEnabled': 2, 'TimerCounterPinOffset': 4, 'DAC1Enable': 0, 'FIOAnalog': 0}
>>> # Setup the two timers to be quadrature
>>> d.getFeedback(u3.Timer0Config(8), u3.Timer1Config(8))
Sent: [0x66, 0xf8, 0x3, 0x0, 0x68, 0x0, 0x0, 0x2b, 0x8, 0x0, 0x0, 0x2d, 0x8, 0x0, 0x0, 0x0, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None, None]
>>> # Read the value
[0]
>>> d.getFeedback(u3.QuadratureInputTimer())
Sent: [0x26, 0xf8, 0x3, 0x0, 0x2a, 0x0, 0x0, 0x2a, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
Response: [0xf5, 0xf8, 0x4, 0x0, 0xf5, 0x3, 0x0, 0x0, 0x0, 0xf8, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff]
[-8]
>>> d.getFeedback(u3.QuadratureInputTimer())
Sent: [0x26, 0xf8, 0x3, 0x0, 0x2a, 0x0, 0x0, 0x2a, 0x0, 0x0, 0x0, 0x2a, 0x0, 0x0, 0x0, 0x0, 0x0]
Response: [0x9, 0xf8, 0x4, 0x0, 0xc, 0x0, 0x0, 0x0, 0x0, 0xc, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[12]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-QuadratureInputTimer.txt](#) [view raw](#)

For reading a stop input timer. They are special because the value returns the current edge count and the stop value.

(Section 2.9.1.9 of the User's Guide)

Args:

UpdateReset: Set True if you want to update the value.

Value: The stop value. Only updated if the UpdateReset bit is 1.

Returns a tuple where the first value is current edge count, and the second value is the stop value.

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.configIO(NumberOfTimersEnabled = 2)
Sent: [0x4a, 0xf8, 0x3, 0xb, 0x43, 0x0, 0x1, 0x0, 0x42, 0x0, 0x0, 0x0]
Response: [0x58, 0xf8, 0x3, 0xb, 0x51, 0x0, 0x0, 0x0, 0x42, 0x0, 0xf, 0x0]
{'NumberOfTimersEnabled': 2, 'TimerCounterPinOffset': 4, 'DAC1Enable': 0, 'FIOAnalog': 0}
>>> # Setup the timer to be Stop Input
>>> d.getFeedback(u3.Timer1Config(9, Value = 30))
Sent: [0x50, 0xf8, 0x3, 0x0, 0x54, 0x0, 0x0, 0x2d, 0x9, 0x1e, 0x0, 0x0]
Response: [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
>>> d.getFeedback(u3.TimerStopInput1())
Sent: [0x28, 0xf8, 0x3, 0x0, 0x2c, 0x0, 0x0, 0x2c, 0x0, 0x0, 0x0, 0x0]
Response: [0x1b, 0xf8, 0x4, 0x0, 0x1e, 0x0, 0x0, 0x0, 0x0, 0x1e, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[(0, 0)]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-TimerStopInput1.txt](#) [view raw](#)

5.2.5.16 - Timer#Config: IOType=43,45

Timer#Config. 4 Command Bytes:		
0	IOType = 43, 45	
1	TimerMode	
2	Value LSB	
3	Value MSB	
0 Response Bytes:		

This IOType configures a particular timer.

- **TimerMode:** See [Section 2.9](#) for more information about the available modes.
- **Value:** The meaning of this parameter varies with the timer mode.

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

This IOType configures a particular timer.

```

timer = # of the timer to configure

TimerMode = See Section 2.9 for more information about the available modes.

Value = The meaning of this parameter varies with the timer mode.

>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.configIO(NumberOfTimersEnabled = 1)
Sent:  [0x49, 0xf8, 0x3, 0xb, 0x42, 0x0, 0x1, 0x0, 0x41, 0x0, 0x0, 0x0]
Response:  [0x57, 0xf8, 0x3, 0xb, 0x50, 0x0, 0x0, 0x0, 0x41, 0x0, 0xf, 0x0]
{'NumberOfTimersEnabled': 1, 'TimerCounterPinOffset': 4, 'DAC1Enable': 0, 'FIOAnalog'
>>> d.getFeedback(u3.TimerConfig(timer = 0, TimerMode = 0, Value = 0))
Sent:  [0x27, 0xf8, 0x3, 0x0, 0x2b, 0x0, 0x0, 0x2b, 0x0, 0x0, 0x0, 0x0]
Response:  [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
>>> d.getFeedback(u3.TimerConfig(timer = 0, TimerMode = 0, Value = 65535))
Sent:  [0x27, 0xf8, 0x3, 0x0, 0x29, 0x2, 0x0, 0x2b, 0x0, 0xff, 0xff, 0x0]
Response:  [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]

```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-TimerConfig.txt](#) [view raw](#)

This IOType configures Timer0.

```

TimerMode = See Section 2.9 for more information about the available modes.

Value = The meaning of this parameter varies with the timer mode.

>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.configIO(NumberOfTimersEnabled = 1)
Sent:  [0x49, 0xf8, 0x3, 0xb, 0x42, 0x0, 0x1, 0x0, 0x41, 0x0, 0x0, 0x0]
Response:  [0x57, 0xf8, 0x3, 0xb, 0x50, 0x0, 0x0, 0x0, 0x41, 0x0, 0xf, 0x0]
{'NumberOfTimersEnabled': 1, 'TimerCounterPinOffset': 4, 'DAC1Enable': 0, 'FIOAnalog'
>>> d.getFeedback(u3.Timer0Config(TimerMode = 1, Value = 0))
Sent:  [0x28, 0xf8, 0x3, 0x0, 0x2c, 0x0, 0x0, 0x2b, 0x1, 0x0, 0x0, 0x0]
Response:  [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]
>>> d.getFeedback(u3.Timer0Config(TimerMode = 1, Value = 65535))
Sent:  [0x28, 0xf8, 0x3, 0x0, 0x2a, 0x2, 0x0, 0x2b, 0x1, 0xff, 0xff, 0x0]
Response:  [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]

```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-Timer0Config.txt](#) [view raw](#)

This IOType configures Timer1.

```

TimerMode = See Section 2.9 for more information about the available modes.

Value = The meaning of this parameter varies with the timer mode.

>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.configIO(NumberOfTimersEnabled = 2)
Sent:  [0x4a, 0xf8, 0x3, 0xb, 0x43, 0x0, 0x1, 0x0, 0x42, 0x0, 0x0, 0x0]
Response:  [0x58, 0xf8, 0x3, 0xb, 0x51, 0x0, 0x0, 0x0, 0x42, 0x0, 0xf, 0x0]
{'NumberOfTimersEnabled': 2, 'TimerCounterPinOffset': 4, 'DAC1Enable': 0, 'FIOAnalog'
>>> d.getFeedback(u3.Timer1Config(TimerMode = 6, Value = 1))
Sent:  [0x30, 0xf8, 0x3, 0x0, 0x34, 0x0, 0x0, 0x2d, 0x6, 0x1, 0x0, 0x0]
Response:  [0xfa, 0xf8, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[None]

```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-Timer1Config.txt](#) [view raw](#)

5.2.5.17 - 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).

LabJackPython example session

Automatically extracted from [u3.py](#). Debugging turned on to show the bytes sent and received.

Counter Feedback command

Reads a hardware counter, optionally resetting it

counter: 0 or 1

Reset: True (or 1) = Reset, False (or 0) = Don't Reset

Returns the current count from the counter if enabled. If reset, this is the value before the reset.

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.configIO(EnableCounter0 = True, FIOAnalog = 15)
Sent: [0x5f, 0xf8, 0x3, 0xb, 0x58, 0x0, 0x5, 0x0, 0x44, 0x0, 0xf, 0x0]
Response: [0x5a, 0xf8, 0x3, 0xb, 0x53, 0x0, 0x0, 0x0, 0x44, 0x0, 0xf, 0x0]
{'NumberOfTimersEnabled': 0, 'TimerCounterPinOffset': 4, 'DAC1Enable': 0, 'FIOAnalog': 15}
>>> d.getFeedback(u3.Counter(counter = 0, Reset = False))
Sent: [0x31, 0xf8, 0x2, 0x0, 0x36, 0x0, 0x0, 0x36, 0x0, 0x0]
Response: [0xfc, 0xf8, 0x4, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[0]
>>> # Tap a ground wire to counter 0
>>> d.getFeedback(u3.Counter(counter = 0, Reset = False))
Sent: [0x31, 0xf8, 0x2, 0x0, 0x36, 0x0, 0x0, 0x36, 0x0, 0x0]
Response: [0xe9, 0xf8, 0x4, 0x0, 0xec, 0x0, 0x0, 0x0, 0xe8, 0x4, 0x0, 0x0]
[1256]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-Counter.txt](#) [view raw](#)

Counter0 Feedback command

Reads hardware counter0, optionally resetting it

Reset: True (or 1) = Reset, False (or 0) = Don't Reset

Returns the current count from the counter if enabled. If reset, this is the value before the reset.

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.configIO(EnableCounter0 = True, FIOAnalog = 15)
Sent: [0x5f, 0xf8, 0x3, 0xb, 0x58, 0x0, 0x5, 0x0, 0x44, 0x0, 0xf, 0x0]
Response: [0x5a, 0xf8, 0x3, 0xb, 0x53, 0x0, 0x0, 0x0, 0x44, 0x0, 0xf, 0x0]
{'NumberOfTimersEnabled': 0, 'TimerCounterPinOffset': 4, 'DAC1Enable': 0, 'FIOAnalog': 15}
>>> d.getFeedback(u3.Counter0(Reset = False))
Sent: [0x31, 0xf8, 0x2, 0x0, 0x36, 0x0, 0x0, 0x36, 0x0, 0x0]
Response: [0xfc, 0xf8, 0x4, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[0]
>>> # Tap a ground wire to counter 0
>>> d.getFeedback(u3.Counter0(Reset = False))
Sent: [0x31, 0xf8, 0x2, 0x0, 0x36, 0x0, 0x0, 0x36, 0x0, 0x0]
Response: [0xe, 0xf8, 0x4, 0x0, 0x11, 0x0, 0x0, 0x0, 0x11, 0x0, 0x0, 0x0]
[17]
>>> # Tap a ground wire to counter 0
>>> d.getFeedback(u3.Counter0(Reset = False))
Sent: [0x31, 0xf8, 0x2, 0x0, 0x36, 0x0, 0x0, 0x36, 0x0, 0x0]
Response: [0x19, 0xf8, 0x4, 0x0, 0x1c, 0x0, 0x0, 0x0, 0xb, 0x11, 0x0, 0x0]
[4363]
```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-Counter0.txt](#) [view raw](#)

Counter1 Feedback command

Reads hardware counter1, optionally resetting it

Reset: True (or 1) = Reset, False (or 0) = Don't Reset

Returns the current count from the counter if enabled. If reset, this is the value before the reset.

```
>>> import u3
>>> d = u3.U3()
>>> d.debug = True
>>> d.configIO(EnableCounter1 = True, FIOAnalog = 15)
Sent: [0x63, 0xf8, 0x3, 0xb, 0x5c, 0x0, 0x5, 0x0, 0x48, 0x0, 0xf, 0x0]
Response: [0x5e, 0xf8, 0x3, 0xb, 0x57, 0x0, 0x0, 0x0, 0x48, 0x0, 0xf, 0x0]
{'NumberOfTimersEnabled': 0, 'TimerCounterPinOffset': 4, 'DAC1Enable': 0, 'FIOAnalog': 15}
>>> d.getFeedback(u3.Counter1(Reset = False))
```

```

Sent: [0x32, 0xf8, 0x2, 0x0, 0x37, 0x0, 0x0, 0x37, 0x0, 0x0]
Response: [0xfc, 0xf8, 0x4, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
[0]
>>> # Tap a ground wire to counter 1
>>> d.getFeedback(u3.Counter1(Reset = False))
Sent: [0x32, 0xf8, 0x2, 0x0, 0x37, 0x0, 0x0, 0x37, 0x0, 0x0]
Response: [0xfd, 0xf8, 0x4, 0x0, 0x1, 0x0, 0x0, 0x0, 0x0, 0x1, 0x0, 0x0, 0x0, 0x0]
[1]
>>> # Tap a ground wire to counter 1
>>> d.getFeedback(u3.Counter1(Reset = False))
Sent: [0x32, 0xf8, 0x2, 0x0, 0x37, 0x0, 0x0, 0x37, 0x0, 0x0]
Response: [0xb4, 0xf8, 0x4, 0x0, 0xb7, 0x0, 0x0, 0x0, 0x0, 0x6b, 0x2b, 0x21, 0x0, 0:
[2173803]

```

[This Gist](#) brought to you by [GitHub](#).

[u3-feedback-Counter1.txt](#) [view raw](#)

5.2.5.18 - Buzzer: IOType=63

Buzzer 6 Command Bytes:	
0	IOType = 63
1	Bit 0: Continuous
2	Period LSB
3	Period MSB
4	Toggles LSB
5	Toggles MSB
0 Response Bytes:	

This IOType is used to make the buzzer buzz. The buzzer is only available on hardware revisions 1.20 and 1.21, not on 1.30.

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

5.2.6 - ReadMem (ReadCal)

Reads 1 block (32 bytes) from the non-volatile user or calibration memory. Command number 0x2A accesses the user memory area which consists of 256 bytes (block numbers 0-7). Command number 0x2D accesses the calibration memory area consisting of 512 bytes (block numbers 0-15), of which the last 8 blocks are not used. 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 consisting of 512 bytes (block numbers 0-15), of which the last 8 blocks are not used. Memory must be erased before writing. Do not call this function while streaming.

Command:	
Byte	
0	Checksum8
1	0xF8
2	0x11
3	0x28 (0x2b)
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	0x00
7	BlockNum
8-39	32 Bytes of Data
Response:	
Byte	
0	Checksum8
1	0xF8
2	0x01
3	0x28 (0x2B)
4	Checksum16 (LSB)
5	Checksum16 (MSB)
6	Errorcode
7	0x00

5.2.8 - EraseMem (EraseCal)

The U3 uses flash memory that must be erased before writing. Command number 0x29 erases the entire user memory area. Command number 0x2C erases the entire calibration memory area. The EraseCal command has two extra constant bytes, to

make it more difficult to call the function accidentally. Do not call this function while streaming.

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

5.2.9 - Reset

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

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

5.2.10 - StreamConfig

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. Requires U3 hardware version 1.21.

Command:				
Byte				
0	Checksum8			
1	0xF8			
2	NumChannels + 3			
3	0x11			
4	Checksum16 (LSB)			
5	Checksum16 (MSB)			
6	NumChannels			
7	SamplesPerPacket (1-25)			
8	Reserved			
9	ScanConfig			
		Bit 7: Reserved		
		Bit 6: Reserved		
		Bit 3: Internal stream clock frequency.		
			b0: 4 MHz	
			b1: 48 MHz	
		Bit 2: Divide Clock by 256		
		Bits 0-1: Resolution		
			b00: 12.8-bit effective	
			b01: 11.9-bit effective	
			b10: 11.3-bit effective	
			b11: 10.5-bit effective	
10-11	Scan Interval (1-65535)			
12	PChannel			
13	NChannel			
Repeat 12-13 for each channel				
Response:				
Byte				
0	Checksum8			
1	0xF8			
2	0x01			
3	0x11			
4	Checksum16 (LSB)			
5	Checksum16 (MSB)			
6	Errorcode			
7	0x00			

- **NumChannels:** This is the number of channels you will sample per scan (1-25).
- **SamplesPerPacket:** Specifies how many samples will be pulled out of the U3 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 bas clock and effective resolution.
- **ScanInterval:** (1-65535) This value divided by the clock frequency defined in the ScanConfig parameter, gives the interval (in seconds) between scans.
- **PChannel/NChannel:** For each channel, these two parameters specify the positive and negative voltage measurement point. PChannel is 0-7 for FIO0-FIO7, 8-15 for EIO0-EIO15, 30 for temp sensor, 31 for Vreg, or 193..224 for digital/timer/counter channels. NChannel is 0-7 for FIO0-FIO7, 8-15 for EIO0-EIO15, 30 for Vref, or 31 for single-ended.

5.2.11 - StreamStart

Once the stream settings are configured, this function is called to start the stream. Requires U3 hardware version 1.21.

Command:		
Byte		
0	0xA8	
1	0xA8	
Response:		
Byte		
0	Checksum8	
1	0xA9	
2	Errorcode	
3	0x00	

5.2.12 - StreamData

After starting the stream, the data will be sent as available in the following format. Reads oldest data from buffer. Requires U3 hardware version 1.21.

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 bytes 6-7 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 U3 uses a feature called auto-recovery. If the stream buffer gets too full, the U3 will go into auto-recovery mode. In this mode, the U3 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 U3 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.

5.2.13 - StreamStop

Requires U3 hardware version 1.21.

Command:		
Byte		
0	0xB0	
1	0xB0	
Response:		
Byte		
0	Checksum8	
1	0xB1	
2	Errorcode	
3	0x00	

5.2.14 - Watchdog

Requires U3 hardware version 1.21. Controls a firmware based watchdog timer. Unattended systems requiring maximum up-time might use this capability to reset the U3 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 U3 with a short from FIO6 to SPC (FIO2 to SCL on U3 1.20/1.21), then remove the jumper and power cycle the device again. This also affects the parameters in the ConfigU3 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.

Note: Do **not** call this function while streaming.

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

5.2.15 - SPI

Requires U3 hardware version 1.21. 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. A zero corresponds to the maximum speed of about 80kHz and 255 the minimum speed of about 5.5kHz.
- **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=0, CPOL=0

Mode B: CPHA=0, CPOL=1

Mode C: CPHA=1, CPOL=0

Mode D: CPHA=1, CPOL=1

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.

5.2.16 - AsynchConfig

Requires U3 hardware version 1.21+. Configures the U3 UART for asynchronous communication. On hardware version 1.30 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 4, TX=FIO4 and RX=FIO5. On hardware version 1.21, the UART uses SDA for TX and SCL for RX. 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.

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	BaudFactor LSB (1.30 only)		
9	BaudFactor MSB		
Response:			
Byte			
0	Checksum8		
1	0xF8		
2	0x02		
3	0x14		
4	Checksum16 (LSB)		
5	Checksum16 (MSB)		
6	Errorcode		
7	AsynchOptions		
8	BaudFactor LSB (1.30 only)		
9	BaudFactor MSB		

- **AsynchOptions:**

Bit 7: Update If true, the new parameters are written (otherwise just a read is done).

Bit 6: UARTEnable If true, the UART module is enabled. Note that no data can be transferred until pins have been assigned to the UART module using the ConfigIO function.

- **BaudFactor16 (BaudFactor8):** This 16-bit value sets the baud rate according the following formula: BaudFactor16 = 2¹⁶ – 48000000/(2 x 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 = 2⁸ – TimerClockBase/(Desired Baud)).

5.2.17 - AsynchTX

Requires U3 hardware version 1.21. Sends bytes to the U3 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.

5.2.18 - AsynchRX

Requires U3 hardware version 1.21. Reads the oldest 32 bytes from the U3 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	1 + NumAsynchWords	
3	0x15	
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.

5.2.19 - I²C

Requires U3 hardware version 1.21+. Sends and receives serial data using I²C (I2C) synchronous communication.

Command:		
Byte		
0	Checksum8	
1	0xF8	
2	4 + NumI2CWordsSend	
3	0x3B	
4	Checksum16 (LSB)	
5	Checksum16 (MSB)	
6	I2COptions	Bits 7-4: Reserved Bit 3: 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 (2ⁿ-(n+1))-1.

5.2.20 - SHT1X

Requires U3 hardware version 1.21. Reads temperature and humidity from a Sensirion SHT1X sensor (which is used by the EI-

1050). For more information, see the [EI-1050 datasheet](#), 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.

5.2.21 - 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 U3 flash has a rated endurance of at least 20000 writes, which is plenty for reasonable operation, but if this function is called in a high-speed loop the flash could eventually be damaged.

Note: Do **not** call this function while streaming.

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		

5.2.22 - ReadDefaults (ReadCurrent)

Reads the power-up defaults from flash (Read the current configuration).

Command:		Defaults Map			
Byte		Block Number	Byte Offset	Description	Nominal Values
0	Checksum8	0	0-3	Not Used	0x00
1	0xF8	0	4	FIO Directions	0x00
2	0x01	0	5	FIO States	0xFF
3	0x0E	0	6	FIO Analog	0x00
4	Checksum16 (LSB)	0	7	Not Used	0x00
5	Checksum16 (MSB)	0	8	EIO Directions	0x00
6	0x00	0	9	EIO States	0xFF
7	bits[0:6] BlockNum 0-7	0	10	EIO Analog	0x00
	bit 7: 1 = ReadCurrent	0	11	Not Used	0x00
		0	12	CIO Directions	0x00
		0	13	CIO States	0xFF
Response:		0	14-15	Not Used	0x00
Byte		0	16	Config Write Mask	0x00 (NM)
0	Checksum8	0	17	NumOfTimersEnabled	0x00
1	0xF8	0	18	Counter Mask	0x00
2	0x01	0	19	Pin Offset	0x04
3	0x0E	0	20	Options	0x00
4	Checksum16 (LSB)	0	21-31	Not Used	0x00
5	Checksum16 (MSB)	0	0 (32)	Clock_Source	0x02
6	Errorcode	1	1 (33)	Divisor	0x00
7	0x00	1	2-15 (33-47)	Not Used	0x00
8-39	Data	1	16 (48)	TMR0 Mode	0x0A
		1	17 (49)	TMR0 Value L	0x00
		1	18 (50)	TMR0 Value H	0x00
		1	19 (51)	Not Used	0x00
		1	20 (52)	TMR1 Mode	0x0A
		1	21 (53)	TMR1 Value L	0x00
		1	22 (54)	TMR1 Value H	0x00
		1	23-31 (55-63)	Not Used	0x00
		2	0-15 (64-79)	Not Used	0x00
		2	16-17 (80-81)	DAC0 (2 Bytes)	0x0000
		2	18-19 (82-83)	Not Used	0x00
		2	20-21 (84-85)	DAC1 (2 Bytes)	0x0000
		2	22-31 (86-95)	Not Used	0x00
		3	0-15 (96-111)	AIN Neg Channel	0x1F
		3	16-31 (112-127)	Not Used	0x00

5.3 - Errorcodes

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

Code	Description
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
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
112	UART_TIMEOUT
113	UART_NOT_CONNECTED
114	UART_NOT_ENABLED

5.4 - Calibration Constants

Calibration Constant

The majority of the U3's analog interface functions return or require binary values. Converting between binary and voltages requires the use of calibration constants and formulas.

When using ModBus the U3 will apply calibration automatically, so voltages are sent to and read from the U3, formatted as a float.

Which Constants Should I Use?

The calibration constants stored on the U3 can be categorized as follows:

- Analog Input
- Analog Output
- Internal Temperature

Analog Input: Since the U3 uses multiplexed channels connected to a single analog-to-digital converter (ADC), all low-voltage channels have the same calibration for a given configuration. High-voltage channels have individual scaling circuitry out front, and thus the calibration is unique for each channel. The table below shows where the various calibration values are stored in the Mem area. Generally when communication is initiated with the U3, four calls will be made to the ReadMem function to retrieve the first 4 blocks of memory. This information can then be used to convert all analog input readings to voltages. Again, the high level Windows DLL (LabJackUD) does this automatically.

Analog Output: Only two calibrations are provided, one for DAC0 and one for DAC1.

Internal Temperature: This calibration is applied to the bits of a reading from channel 30 (internal temp).

U3 Input Ranges

The U3 input ranges can be found in section 2.6.2 of the User's Guide. For your convenience, that table has been provided again below.

	Max V	Min V		
Single-Ended	2.44	0		
Differential	2.44	-2.44		
Special 0-3.6	3.6	0		
Table 2.6.2-1. Nominal Analog Input Voltage Ranges for Low-Voltage Channels				
	Max V	Min V		
Single-Ended	10.3	-10.3		
Differential	N/A	N/A		
Special -10/+20	20.1	-10.3		
Table 2.6.2-2. Nominal Analog Input Voltage Ranges for High-Voltage Channels				

U3 Calibration Formulas (Analog In)

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 (Single-Ended)

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

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

U3 Calibration Formulas (Analog Out)

Writing to the U3's DAC require that the desired voltage be converted into a binary value. To convert the desired voltage to binary select the Slope and Offset calibration constants for the DAC being used and plug into the following formula.

Bits = (DesiredVolts * Slope) + Offset

U3 Calibration Formulas (Internal Temp)

Internal Temperature can be obtained by reading channel 30 and using the following formula.

Temp (K) = Bits * TemperatureSlope

U3 Calibration Constants

Below are the various calibration values are stored in the Mem area. Generally when communication is initiated with the U3, eight calls will be made to the ReadMem function to retrieve the first 8 blocks of memory. This information can then be used to convert all analog input readings to voltages. Again, the high level Windows DLL (LabJackUD) does this automatically.

Block #	Starting Byte		Nominal Value	
0	0	LV AIN SE Slope	3.7231E-05	volts/bit
0	8	LV AIN SE Offset	0.0000E+00	volts
0	16	LV AIN Diff Slope	7.4463E-05	volts/bit
0	24	LV AIN Diff Offset	-2.4400E+00	volts
1	0	DAC0 Slope	5.1717E-01	bits/volt
1	8	DAC0 Offset	0.0000E+00	bits
1	16	DAC1 Slope	5.1717E+1	bits/volt
1	24	DAC1 Offset	0.0000E+00	bits
2	0	Temp Slope	1.3021E-02	degK/bit
2	8	Vref @Cal	2.4400E+00	volts
2	16	Vref*1.5 @Cal	3.6600E+00	volts
2	24	Vreg @Cal	3.3000E+00	volts
Table 5.4-1. Normal Calibration Constant Memory Locations				
Block #	Starting Byte		Nominal Value	
3	0	HV AIN0 Slope	3.1400E-4	volts/bit
3	8	HV AIN1 Slope	3.1400E-4	volts/bit
3	16	HV AIN2 Slope	3.1400E-4	volts/bit
3	24	HV AIN3 Slope	3.1400E-4	volts/bit
4	0	HV AIN0 Offset	-10.3	volts
4	8	HV AIN1 Offset	-10.3	volts
4	16	HV AIN2 Offset	-10.3	volts
4	24	HV AIN3 Offset	-10.3	volts
Table 5.4-2. Additional High-Voltage Calibration Constant Memory Locations				

Format of the Calibration Constants

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

Fixed Point Byte Array (LSB, ..., MSB)	Floating Point Double
{0,0,0,0,0,0,0,0}	0.0000000000
{0,0,0,0,1,0,0,0}	1.0000000000
{0,0,0,0,255,255,255,255}	-1.0000000000
{51,51,51,51,0,0,0,0}	0.2000000000
{205,204,204,204,255,255,255,255}	-0.2000000000
{73,20,5,0,0,0,0,0}	0.000775030
{255,122,20,110,2,0,0,0}	2.4300000000
{102,102,102,38,42,1,0,0}	298.1500000000
Table 5.4-3. Fixed Point Conversion Examples	

Appendix A - Specifications

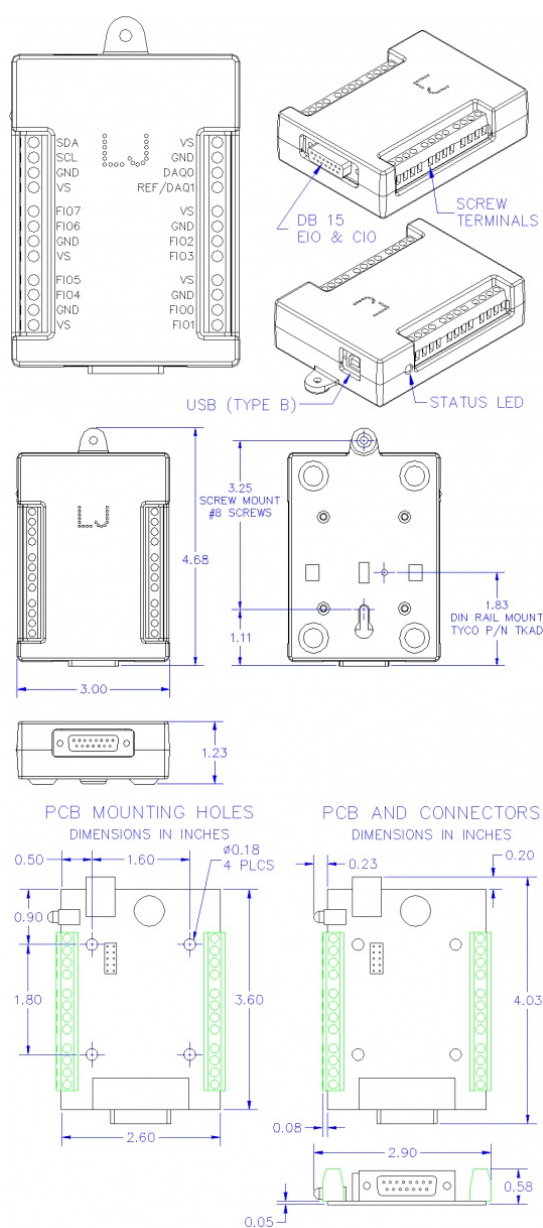
Specifications at 25 degrees C and Vusb/Vext = 5.0V, except where noted.

Parameter	Conditions	Min	Typical	Max	Units
General					
USB Cable Length				5	meters
Supply Voltage		4	5	5.25	volts
Supply Current (1)	Hardware V1.21+		50		mA
Operating Temperature		-40		85	°C
Clock Error	-40 to 85 °C			1.5	%
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	5.25	volts
	Bus-Powered	4	5	5.25	volts
Maximum Current (3)	Self-Powered		450		mA
	Bus-Powered		50		mA
(1) Typical current drawn by the U3 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 U3. Self-powered would apply to USB hubs with a power supply, all known desktop computer USB hosts, and some notebook computer USB hosts. An example of bus-powered would be a hub with no power supply, or many PDA ports. The current rating is the maximum current that should be sources through the U3 and out of the Vs terminals.					
Parameter	Conditions	Min	Typical	Max	Units
Analog Inputs					
Typical input Range (1)	Single-Ended, LV	0		2.44	volts
	Differential, LV	-2.44		2.44	volts
	Special, LV	0		3.6	volts
	Single-Ended, HV	-10.3		10.3	volts
	Special, HV	-10.3		20.1	volts
Max AIN Voltage to GND (2)	Valid Readings, LV	-0.3		3.6	volts
Max AIN Voltage to GND (3)	No Damage, FIO	-10		10	volts
	No Damage, EIO	-6		6	volts
	No Damage, HV	-40		40	volts
Input Impedance (4)	LV		40		MΩ
	HV		1.3		MΩ
Source Impedance (4)	LongSettling Off, LV			10	kΩ
	LongSettling On, LV			200	kΩ
	LongSettling Off, HV			1	kΩ
	LongSettling On, HV			1	kΩ
Resolution			12		bits
Integral Linearity Error			±0.05		% FS
Differential Linearity Error			±1		counts
Absolute Accuracy	Single-Ended		±0.13		% FS
	Differential		±0.25		% FS
	Special 0-3.6		±0.25		% FS
Temperature Drift			15		ppm/°C
Noise (Peak-To-Peak) (5)	QuickSample Off		±1		counts
	QuickSample On		±2		counts
Effective Resolution (RMS) (6)	QuickSample Off		>12		bits
Noise-Free Resolution (5)	QuickSample Off		11		bits
	Single-Ended, LV		1.2		mV
	Diff., Special, LV		2.4		mV
	Single-Ended, HV		9.8		mV
	Special, HV		19.5		mV
Command/Response Speed	See Section 3.1				
Stream Performance	See Section 3.2				
(1) With DAC1 disabled on hardware version < 1.30					
(2) This is the maximum voltage on any AIN pin compared to ground for valid measurements. Not that differential channel has a minimum voltage of -2.44 volts, meaning that the positive channel can be 2.44 volts less than the negative channel, but no AIN pin can go more than 0.3 volts below ground.					
(3) Maximum voltage, compared to ground, to avoid damage to the device. Protection level is the same whether the device is powered or not.					
(4) 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			10		bits
Absolute Accuracy	5% to 95% FS		±5		% FS
Integral Linearity Error			±1		counts
Differential Linearity Error			±1		counts
Error Due To Loading	@ 100 µA		0.1		%
	@ 1 mA		1		%
Source Impedance			50		Ω
Short Circuit Current (2)	Max to GND		45		mA
Slew Rate			0.4		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)	Hardware V1.21+			8	MHz
Input Timer Total Edge Rate (6)	No Stream, V1.21+			30000	edges/s

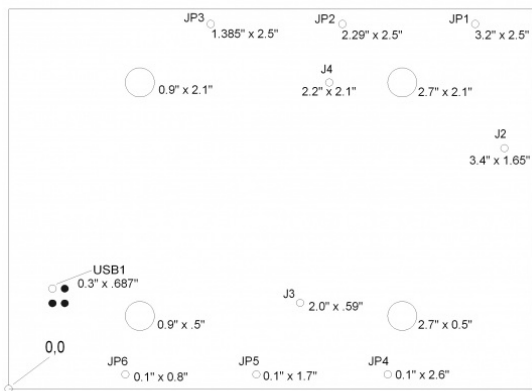
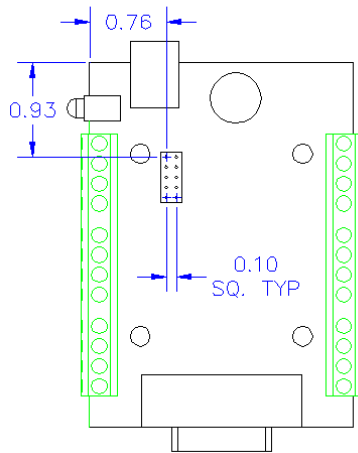
	write streaming		/000	edges/s	
(1)	Maximum and minimum analog output voltage is limited by the supply voltages (Vs and GND). The specifications assume Vs is 5.0 volts. Also, the ability of the DAC output buffer to driver voltages close to the power rails, decreases with increasing output current, but 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 recommended when the U3 is unpowered, as the voltage will attempt to supply operating power to the U3 possible 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 drop 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.				

Appendix B - Enclosure and PCB Drawings

Various drawings follow. CAD drawings of the U3 attached to the bottom of this page (DWG, DXF, IGES, STEP).



PIN HEADER DIMENSIONS IN INCHES



U3 PCB drawing showing the coordinates (in inches) for pin 1 of each connector.

File attachment:

[U3 Enclosure CAD Drawings.zip](#)