LabJack UD Driver for Windows Quick Reference – U3 – Mar 14, 2008

Other resources: "LabJack U3 User's Guide", LabJack UD header file ("labjackud.h"), and "DAQFactory-LabJack Application Guide".

1.0 Function Reference Introduction:

The "@" symbol used below means pass the address of the data, rather than the data itself. This is the equivalent of the "&" symbol in C or C++. This allows the parameter to pass data into and out of the function, whereas normal parameters only pass data in.

"[]" is used to signify an array. Arrays always must be initialized with some minimum number of elements, so always consider the maximum number of elements the array could pass out of the function (which is always known). The method of passing arrays varies from language to language, but what is actually passed to the UD driver is the address of the first element of the array. In DAQFactory, only eGet supports passing array data in and out, and always assumes bytes/chars.

At the core, the UD driver only has three basic functions: AddRequest, Go, and GetResult. First, use AddRequest to build a list of requests (read analog input, set analog output, etc.). Then a Go function is called which causes the driver to actually write to and read from the hardware. Finally, the GetResult functions are used to get the results (errorcodes and data) from the driver. This is the basic add/go/get method of using the UD driver. Alternatively, the E ("easy") functions can be used for some basic applications.

When adding requests, there are always at least five parameters:

Handle or DeviceNum (lng): Handle is from the open function and specifies which device the request is destined for. In DAQFactory, opening is handled by DAQFactory and this first parameter is the DeviceNum (0 for "first found over USB").

IOType (lng): Constants starting with "LJ_io…" are IOTypes and always passed in the IOType parameter. They are the only thing ever passed in the IOType parameter. The IOType specifies what type of action is being done.

Channel (lng): Constants starting with "LJ_ch…" are special channels and always passed in the Channel parameter. The only thing ever passed in the channel parameter is such a special channel constant or an actual channel number.

Value (dbl): This is usually the actual value passed in and/or out of the function. Sometimes special value constants are used.

x1 (lng): Extra parameter used with only a few IOTypes or special channels. Either passes an integer or an address to an array.

In general, if the IOType relates to a specific channel or port (e.g. LJ_ioGET_AIN), then the Channel parameter is the desired channel (or starting channel) and the Value parameter is the write or read data. On the other hand, if the desired action relates to a device-wide setting, the put/get config IOTypes are used with a special channel constant:

eGet (DeviceNum, LJ_ioGET_AIN, 3, @Value, 0); // IOType deals with specific channel (AIN3).
eGet (DeviceNum, LJ_ioGET_CONFIG, LJ_chLOCALID, @Value, 0); // IOType deals with device-wide setting.

When programming in any language, it is recommended to have the header file handy (or an editable copy of this document), so that constants can be copied and pasted into the code.

1.1 Open Function:

Errorcode = OpenLabJack (lndDeviceType, lngConnectionType, strAddress, lngFirstFound, @Handle)
With languages other than DAQFactory, use this open function to get a handle that is passed in the Handle/DeviceNum parameter.

1.2 Add/Go/Get Functions:

Errorcode = eGet (Handle, IOType, Channel, @Value, x1)
Errorcode = ePut (Handle, IOType, Channel, Value, x1)
Errorcode = AddRequest (Handle, IOType, Channel, Value, x1, dblUserData)
Errorcode = Go (void)
Errorcode = GoOne (Handle)
Errorcode = GetResult (Handle, IOType, Channel, @Value)
Errorcode = GetFirstResult (Handle, @IOType, @Channel, @Value, @x1, @dblUserData)
Errorcode = GetNextResult (Handle, @IOType, @Channel, @Value, @x1, @dblUserData)

The Go function executes all open requests on all devices, while GoOne executes open requests only on the one specified device.

There are two basic ways to read results. With GetResult, the IOType and Channel are passed in to specify which result should be read (thus this method does not work if there are multiple requests with the same IOType and Channel). The alternative method is to call GetFirstResult followed by multiple calls to GetNextResult to simply read the results in sequential order (same order in which the requests were added). With this method the Handle parameter is the only input to the function and all other parameters are outputs.

The eGet function combines one AddRequest, a GoOne, and one GetResult, into a single call. The ePut function is the same, except Value is not passed by reference and thus cannot pass data out of the function (into the function only).

1.3 E Functions:

The E ("easy") functions are a simplified alternative to the more flexible and powerful add/go/get method discussed above.

Errorcode = eAIN (Handle, lngChannelP, lngChannelN, @dblVoltage, lngRange, lngResolution, lngSettling, lngBinary, lngReserved1, lngReserved2)
Reads a single analog input (and configures as analog in if needed). ChannelP/ChannelN= see Section 2.3 below. Voltage parameter returns reading. Range ignored. Resolution=TRUE=QuickSample. Settling=TRUE=LongSettling. Binary=TRUE will return raw value.
eAIN (Handle, 3, 31, @dblVoltage, 0, 0, 0, 0, 0, 0); // Take single-ended reading of AIN3.

Errorcode = eDAC (Handle, lngChannel, dblVoltage, lngBinary, lngReserved1, lngReserved2)
Updates a single analog output (and enables if needed). Channel=0-1. Voltage=0-5 (unless Binary=TRUE).
eDAC (Handle, 0, 3.1, 0, 0, 0); // Set DAC0 to 3.1 volts.

Errorcode = eDI (Handle, lngChannel, @lngState)
Reads the state of a single digital input (and configures if needed). Channel=0-19. State returns TRUE/FALSE.
eDI (Handle, 2, @lngState); // Read state of FIO2.

Errorcode = eDO (Handle, lngChannel, lngState)
Updates a single digital output (and configures if needed). Channel=0-19. State=TRUE/FALSE.
eDO (Handle, 3, 1); // Set FIO3 to output-high.

Errorcode = eAddGoGet (Handle, lngNumRequests, @lngIOTypes[], @lngChannels[], @dblValues[], @lngX1s[], @lngRequestErrors[], @lngGoError, @lngResultErrors[])
Passes multiple requests/results via arrays. All arrays must be initialized with at least NumRequests elements.

Errorcode = eTCConfig (Handle, @lngEnableTimers[], @lngEnableCounters[], lngTCPinOffset, lngTimerClockBaseIndex, lngTimerClockDivisor, @lngTimerModes[], @dblTimerValues[], lngReserved1, lngReserved2)
Configures and initializes all timers and counters. See Section 2.6 below. Initialize all arrays with 2 elements.

Errorcode = eTCValues (Handle, @lngReadTimers[], @lngUpdateResetTimers[], @lngReadCounters[], @lngResetCounters[], @dblTimerValues[], @dblCounterValues[], lngReserved1, lngReserved2)
Updates and reads all the timers and counters. Initialize all arrays with 2 elements.

1.4 Other Functions:

Errorcode = ListAll (lngDeviceType, lngConnectionType, @lngNumFound, @lngSerialNumbers[], @lngIDs[], @dblAddresses[])
Returns all the devices found of a given device type and connection type. Arrays must be initialized to 128 elements.

lngConstant = StringToConstant (strString);
Converts the given string to the appropriate constant number. Useful with languages that cannot use the header file.

ErrorToString (ErrorCode, strString);
Returns a string describing the given error code or an empty string if not found. String must be initialized to at least 256 chars.

dblDriverVersion = GetDriverVersion (void);
Returns the version # of the UD driver.

TCVoltsToTemp (lngTCType, dblTCVolts, dblCJTempK, @dblTCTempK);
Converts voltage readings from thermocouples to temperatures. TCType is a constant that specifies the thermocouple type, such as LJ_ttK (also supported are B, E, J, N, R, S, and T). TCVolts is the thermocouple voltage. CJTempK is the cold junction temperature in Kelvin. The thermocouple temperature in Kelvin is returned via TCTempK. Note that DAQFactory has it's own conversion functions.

1.5 DAQFactory:

DAQFactory Express is included software that provides the ability to make custom applications, in a higher level environment than a typical programming language such as C or VB. DAQFactory supports scripting which provides the ability to write code doing almost anything the UD driver can do. For complete documentation **see the DAQFactory-LabJack Application Guide**.

In order to use the standard UD constants and functions as shown, always put *using("device.labjack.")* and *include("c:\program files\labjack\drivers\labjackud.h")* at the beginning of an auto-start sequence in each DAQFactory application.

Parameters throughout this quick reference are sometimes prefixed with lng, dbl, and str, to signify that the driver is expecting a long integer, double-precision float, or string. Mention is also made to certain operations that pass arrays of bytes. The numeric data types (lng, dbl, and bytes) are handled automatically by DAQFactory, so just use a normal variable or array. String variables must be declared as such. Only eGet (not AddRequest) can pass arrays (bytes only) in the x1 parameter by reference ("@" symbol).

The add/go/get (and eGet/ePut) method is most common in DAQFactory scripting. The four basic E functions (eAIN, eDAC, eDI, and eDO) would generally not be used in DAQFactory, as this sort of capability is built into DAQFactory itself without using scripting. The fifth E function (eAddGoGet) would also generally not be used, but the last two E functions (eTCConfig and eTCValues) are a good alternative for timer/counter operations.

2.0 Constant Listings and Pseudocode Examples:

The pseudocode throughout this document is formatted for DAQFactory scripting, but easily adapted to C, VB, LabVIEW, and others. Since DAQFactory handles device opening, the first parameter in most UD calls below is a device number rather than a handle.

2.1 Open:

LJ_dtU3: DeviceType constant for the OpenLabJack or ListAll function.
LJ_ctUSB: ConnectionType constant for the OpenLabJack or ListAll function.

OpenLabJack (LJ_dtU3, LJ_ctUSB, "1", TRUE, @Handle) //Generally not needed in DAQFactory.

2.2 Configuration:

LJ_ioPIN_CONFIGURATION_RESET
LJ_ioPUT_ANALOG_ENABLE_BIT
LJ_ioGET_ANALOG_ENABLE_BIT
LJ_ioPUT_ANALOG_ENABLE_PORT // Channel is starting bit. x1 is number of bits.
LJ_ioGET_ANALOG_ENABLE_PORT // Channel is starting bit. x1 is number of bits.
LJ_ioPUT_CONFIG
LJ_ioGET_CONFIG

Special channels used with the get/put config IOTypes:
LJ_chLOCALID
LJ_chHARDWARE_VERSION // Read-only.
LJ_chSERIAL_NUMBER // Read-only.
LJ_chFIRMWARE_VERSION // Read-only.
LJ_chBOOTLOADER_VERSION // Read-only.
LJ_chPRODUCTID // Read-only.
LJ_chLED_STATE

ePut (DeviceNum, LJ_ioPIN_CONFIGURATION_RESET, 0, 0, 0)
ePut (DeviceNum, LJ_ioPUT_ANALOG_ENABLE_PORT, 0, 255, 16) // Set FIO0-FIO7 to analog and all others to digital.
eGet (DeviceNum, LJ_ioGET_CONFIG, LJ_chLOCALID, @Value, 0) // Read local ID.

2.3 Analog Inputs:

LJ_ioGET_AIN // Single-ended.
LJ_ioGET_AIN_DIFF // Differential. Specify negative channel in x1.

Special channels used with the get/put config IOTypes:
LJ_chAIN_RESOLUTION // QuickSample enabled if TRUE.
LJ_chAIN_SETTLING_TIME // LongSettling enabled if TRUE.
LJ_chAIN_BINARY // Raw binary readings returned if TRUE.

Positive Channel Numbers:	Negative Channel Numbers:
0-7 = AIN0-AIN7 (FIO0-FIO7)	0-7 = AIO0-AIN7 (FIO0-FIO7)
8-15 = AIN8-AIN15 (EIO0-EIO7)	8-15 = AIN8-AIN15 (EIO0-EIO7)
30 = Temp Sensor	30 = Vref
31 = Vreg	31 = GND (Single-Ended)
	32 = Special Range (0 to 3.6 on LV channels, -10 to +20 on HV channels)

AddRequest (DeviceNum, LJ_ioPUT_ANALOG_ENABLE_PORT, 0, 60, 16) // Set FIO2-FIO5 to analog and all others to digital.
AddRequest (DeviceNum, LJ_ioGET_AIN, 2, 0, 0, 0) // Request a single-ended read of AIN2.
AddRequest (DeviceNum, LJ_ioGET_AIN_DIFF, 3, 0, 31, 0) // Request a single-ended read of AIN3.
AddRequest (DeviceNum, LJ_ioGET_AIN_DIFF, 3, 0, 32, 0) // Request a read of AIN3 using special range.
AddRequest (DeviceNum, LJ_ioGET_AIN_DIFF, 4, 0, 5, 0) // Request a differential read of AIN4-AIN5.
GoOne (DeviceNum);
GetFirstResult (DeviceNum, 0, 0, @Value, 0, 0) // Retrieve AIN2 reading.
GetNextResult (DeviceNum, 0, 0, @Value, 0, 0) // Retrieve first AIN3 reading.
GetNextResult (DeviceNum, 0, 0, @Value, 0, 0) // Retrieve second AIN3 reading.
GetNextResult (DeviceNum, 0, 0, @Value, 0, 0) // Retrieve AIN4-AIN5 reading.

2.4 Analog Outputs:

LJ_ioPUT_DAC

ePut (DeviceNum, LJ_ioPUT_DAC, 0, 2.50, 0) // Set DAC0 to 2.5 volts.

2.5 Digital I/O:

LJ_ioGET_DIGITAL_BIT // Also sets direction to input.
LJ_ioGET_DIGITAL_BIT_DIR // Read direction without changing direction.
LJ_ioGET_DIGITAL_BIT_STATE // Read state without changing direction.
LJ_ioGET_DIGITAL_PORT // Also sets directions to input. Channel is starting bit. x1 is number of bits.
LJ_ioGET_DIGITAL_PORT_DIR // Does not change directions. Channel is starting bit. x1 is number of bits.
LJ_ioGET_DIGITAL_PORT_STATE // Does not change directions. Channel is starting bit. x1 is number of bits.

LJ_ioPUT_DIGITAL_BIT // Also sets direction to output.
LJ_ioPUT_DIGITAL_PORT // Also sets directions to output. Channel is starting bit. x1 is number of bits.

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

AddRequest (DeviceNum, LJ_ioGET_DIGITAL_BIT, 2, 0, 0, 0) // Request a read from FIO2.
AddRequest (DeviceNum, LJ_ioGET_DIGITAL_PORT, 4, 0, 10, 0) // Request a read from FIO4-EIO5.
AddRequest (DeviceNum, LJ_ioPUT_DIGITAL_BIT, 3, 1, 0, 0) // Set FIO3 to output-high.
AddRequest (DeviceNum, LJ_ioPUT_DIGITAL_PORT, 14, 7, 5, 0) // Set EIO6-CIO2 to b00111 (=d7).
GoOne (DeviceNum)
GetResult (DeviceNum, LJ_ioGET_DIGITAL_BIT, 2, @Value) // Get the FIO2 read.
GetResult (DeviceNum, LJ_ioGET_DIGITAL_PORT, 4, @Value) // Get the FIO4-EIO5 read.
// A typical application might also get the other 2 results just to check for errors.

2.6 Timers and Counters:

LJ_ioGET_COUNTER // Channel is counter #.
LJ_ioPUT_COUNTER_ENABLE // Channel is counter #.
LJ_ioGET_COUNTER_ENABLE // Channel is counter #.
LJ_ioPUT_COUNTER_RESET // Channel is counter #.
LJ_ioGET_TIMER // Channel is timer #.
LJ_ioPUT_TIMER_VALUE // Channel is timer #.
LJ_ioPUT_TIMER_MODE // Channel is timer #.
LJ_ioGET_TIMER_MODE // Channel is timer #.

Special value constants to specify timer modes with IOType LJ_ioPUT_TIMER_MODE:
LJ_tmPWM16 // 16-bit PWM output.
LJ_tmPWM8 // 8-bit PWM output.
LJ_tmRISINGEDGES32 // Period input (32-bit, rising edges).
LJ_tmFALLINGEDGES32 // Period input (32-bit, falling edges).
LJ_tmDUTYCYCLE // Duty cycle input.
LJ_tmFIRMCOUNTER // Firmware counter input.
LJ_tmFIRMCOUNTERDEBOUNCE // Firmware counter input (with debounce).
LJ_tmFREQOUT // Frequency output.
LJ_tmQUAD // Quadrature input.
LJ_tmTIMERSTOP // Timer stop input (odd timers only).
LJ_tmSYSTIMERLOW // System timer low read (no pin used by this mode).
LJ_tmSYSTIMERHIGH // System timer high read (no pin used by this mode).
LJ_tmRISINGEDGES16 // Period input (16-bit, rising edges).
LJ_tmFALLINGEDGES16 // Period input (16-bit, falling edges).

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 = divide by 256.
LJ_chTIMER_COUNTER_PIN_OFFSET // 0-8 allowed on < hardware rev 1.30. Only 4-8 allowed on 1.30+.

Special value constants used with the clock base special channel above (hardware rev 1.21+):
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 (Counter0 not available).
LJ_tc4MHZ_DIV // 4 MHz clock base w/ divisor (Counter0 not available).
LJ_tc12MHZ_DIV // 12 MHz clock base w/ divisor (Counter0 not available).
LJ_tc48MHZ_DIV // 48 MHz clock base w/ divisor (Counter0 not available).
LJ_tcSYS // Equivalent to LJ_tc48MHZ.

// First, an add/go/get block to configure the timers and counters.
AddRequest (DeviceNum, LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 4, 0, 0) // Start on FIO4.
AddRequest (DeviceNum, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 1, 0, 0) // Enable 1 timer on FIO4.

AddRequest (DeviceNum, LJ_ioPUT_COUNTER_ENABLE, 0, 0, 0, 0) // Disable Counter0.
AddRequest (DeviceNum, LJ_ioPUT_COUNTER_ENABLE, 1, 1, 0, 0) // Enable Counter1 on FIO5.
AddRequest (DeviceNum, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tc48MHZ_DIV, 0, 0)
AddRequest (DeviceNum, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 48, 0, 0) // Timer clock = 48M/48 = 1 MHz.
AddRequest (DeviceNum, LJ_ioPUT_TIMER_MODE, 0, LJ_tmPWM8, 0, 0) // Set Timer0 to 8-bit PWM output.
AddRequest (DeviceNum, LJ_ioPUT_TIMER_VALUE, 0, 32768, 0, 0) // Initialize the 8-bit PWM with a 50% duty-cycle.
GoOne (DeviceNum)
// A typical application might get the configuration results just to check for errors.

ePut (DeviceNum, LJ_ioPUT_TIMER_VALUE, 0, 49152, 0) // Change Timer0 PWM duty-cycle to 25%.
eGet (DeviceNum, LJ_ioGET_COUNTER, 1, @Value, 0) // Read the count from Counter1.
ePut (DeviceNum, LJ_ioPUT_COUNTER_RESET, 1, 1, 0) // Reset Counter1 to zero.

Note that the above pseudocode shows add/go/get methods, but the E functions (eTCConfig and eTCValues) are often recommended for timer/counter applications.

2.7 Streaming:

LJ_ioCLEAR_STREAM_CHANNELS
LJ_ioADD_STREAM_CHANNEL // Analog channels must be configured as analog.
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

Special channel used with LJ_ioGET_STREAM_DATA, as an alternative to a specific channel:
LJ_chALL_CHANNELS

Special channels used with the get/put config IOTypes:
LJ_chSTREAM_SCAN_FREQUENCY // Up to 50000 samples/sec divided by number of channels.
LJ_chSTREAM_BUFFER_SIZE // UD driver (PC RAM) buffer size in samples.
LJ_chSTREAM_WAIT_MODE // See below.
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 // 1 = lowest latency. 25 (default) = highest sample rate.

Special value constants used with the wait mode special channel above:
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.

In addition to the channels described in the analog input section, the following digital channels are available in stream mode:
193 = EIO_FIO (digital states), 200 = Timer0, 201 = Timer1, 210 = Counter0, 211 = Counter1, and 224 = TC_Capture.

AddRequest (DeviceNum, LJ_ioPUT_ANALOG_ENABLE_PORT, 0, 524, 16, 0) // Set channels 2/3/9 to analog, all others to digital.
AddRequest (DeviceNum, LJ_ioPUT_CONFIG, LJ_chSTREAM_SCAN_FREQUENCY, 10000, 0, 0) // Set scan rate.
AddRequest (DeviceNum, LJ_ioPUT_CONFIG, LJ_chSTREAM_BUFFER_SIZE, 200000, 0, 0) // 10 second UD buffer (2 channels).
AddRequest (DeviceNum, LJ_ioPUT_CONFIG, LJ_chSTREAM_WAIT_MODE, LJ_swSLEEP, 0, 0)
AddRequest (DeviceNum, LJ_ioCLEAR_STREAM_CHANNELS, 0, 0, 0, 0)
AddRequest (DeviceNum, LJ_ioADD_STREAM_CHANNEL, 2, 0, 0, 0) // 1st channel in scan list. AIN2 single-ended.
AddRequest (DeviceNum, LJ_ioADD_STREAM_CHANNEL_DIFF, 3, 0, 9, 0) // 2nd channel in scan list. AIN3-AIN9.
GoOne (DeviceNum)
// A typical application might get the configuration results just to check for errors.

eGet(DeviceNum, LJ_ioSTART_STREAM, 0, @Value, 0) // Start the stream. Actual scan rate returned in Value.

// Read a 0.5 second block of stream data. This would normally be done repeatedly in a loop. ***DAQFactory Note:*** *The following*
// call with IOType LJ_ioGET_STREAM_DATA would never be used in DAQFactory, as stream reading is handled automatically.
numScans = 5000
eGet(DeviceNum, LJ_ioGET_STREAM_DATA, LJ_chALL_CHANNELS, @numScans, @array) // numScans returns number read.
eGet(DeviceNum, LJ_ioGET_CONFIG, LJ_chSTREAM_BACKLOG_COMM, @CommBacklog, 0) // Check the U3 backlog.
eGet(DeviceNum, LJ_ioGET_CONFIG, LJ_chSTREAM_BACKLOG_UD, @UDBacklog, 0) // Check the UD backlog.

ePut (DeviceNum, LJ_ioSTOP_STREAM, 0, 0, 0) // Stop the stream.

2.8 Low-Level Communication:

LJ_ioRAW_OUT
LJ_ioRAW_IN
Channel = 0 for normal pipe or 1 for stream pipe. Value is number of bytes (1-16384). x1 is the address of a byte array.

// Following is example pseudocode to write and read the low-level command ConfigTimerClock:
private array = {0x05,0xF8,0x02,0x0A,0x00,0x00,0x00,0x00,0x00,0x00}
private numBytesToWrite = 10
private numBytesToRead = 10
eGet(DeviceNum, LJ_ioRAW_OUT, 0, @numBytesToWrite, @array) // Write command to the device.
eGet(DeviceNum, LJ_ioRAW_IN, 0, @numBytesToRead, @array) // Read response from the device.

2.9 SPI:

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

Special channels used with the get/put config IOTypes:
LJ_chSPI_AUTO_CS // If TRUE, the CS line is automatically driven low during communication.
LJ_chSPI_DISABLE_DIR_CONFIG // If TRUE, the direction of the lines is not set automatically.
LJ_chSPI_MODE // See Section 5.3.15 of LabJack U3 User's Guide.
LJ_chSPI_CLOCK_FACTOR // Clock frequency = 1000000/(10+10*(256-SPIClockFactor), where 0=256.
LJ_chSPI_MOSI_PIN_NUM // 0-19.
LJ_chSPI_MISO_PIN_NUM // 0-19.
LJ_chSPI_CLK_PIN_NUM // 0-19.
LJ_chSPI_CS_PIN_NUM // 0-19.

// First, configure the SPI communication.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSPI_AUTO_CS,1,0,0) // Enable automatic chip-select control.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSPI_DISABLE_DIR_CONFIG,0,0,0) // Allow automatic direction control.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSPI_MODE,0,0,0) // Mode A: CPHA=1, CPOL=1.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSPI_CLOCK_FACTOR,0,0,0) // Maximum clock rate (~100kHz).
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSPI_MOSI_PIN_NUM,2,0,0) // Set MOSI to FIO2.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSPI_MISO_PIN_NUM,3,0,0) // Set MISO to FIO3.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSPI_CLK_PIN_NUM,0,0,0) // Set CLK to FIO0.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSPI_CS_PIN_NUM,1,0,0) // Set CS to FIO1.
GoOne(DeviceNum)
// A typical application might get the configuration results just to check for errors.

eGet(DeviceNum, LJ_ioSPI_COMMUNICATION, 0, @numBytesToTransfer, @array) // Transfer the data.

2.10 I^2C:

LJ_ioI2C_COMMUNICATION

Special channels used with the I^2C 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_WRITE_READ // Value: bits 0-7= # to write, bits 8-15 = # to read. Value returns ACKs. x1= array.
LJ_chI2C_GET_ACKS // See I2C examples.

Special channels, used with the get/put config IOTypes:
LJ_chI2C_ADDRESS_BYTE // Generally 2x the chip's address.
LJ_chI2C_SCL_PIN_NUM // 0-19.
LJ_chI2C_SDA_PIN_NUM // 0-19.
LJ_chI2C_OPTIONS // Bit 1 is ResetAtStart.
LJ_chI2C_SPEED_ADJUST // 0 is max speed of about 150 kHz. 255 is the minimum speed of about 10 kHz.

// Example configuration to talk to 24C01C EEPROM chip on the LJTick-DAC:
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chI2C_ADDRESS_BYTE,160,0,0) // Address byte is 0xA0 or d160.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chI2C_SCL_PIN_NUM,0,0,0) // SCL is FIO0
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chI2C_SDA_PIN_NUM,1,0,0) // SDA is FIO1
GoOne(DeviceNum)
// A typical application would get the configuration results just to check for errors.

// Read of EEPROM bytes 0-3 in the user memory area. Requires a write (memory address) and read.
*private numWriteRead = 1 + 4*256 // Write 1 byte (1st byte of Value) and read 4 bytes (2nd byte of Value).*
private array = {0,0,0,0}
eGet(DeviceNum, LJ_ioI2C_COMMUNICATION, LJ_chI2C_WRITE_READ, numWriteRead, @array)
GoOne(DeviceNum);

2.11 Asynchronous Serial (similar to RS232):

LJ_ioASYNCH_COMMUNICATION

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 (0-256). x1= returns array of 32 oldest bytes.
LJ_chASYNCH_TX // Value = Send number to write (0-56), returns number in rx buffer. x1= array.
LJ_chASYNCH_FLUSH // Flushes the rx buffer. All data discarded. Value ignored.

Special channel, used with the get/put config IOTypes:
LJ_chASYNCH_BAUDFACTOR // Value= 2^8 – TimerClockBase/(2*bps).

ePut(DeviceNum, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tc1MHZ_DIV, 0) // 1 MHz timer clock base.
ePut(DeviceNum, LJ_ioPUT_CONFIG, LJ_chASYNCH_BAUDFACTOR, 204, 0) // Set data rate for 9600 bps communication.
ePut(DeviceNum, LJ_ioASYNCH_COMMUNICATION, LJ_chASYNCH_ENABLE, 1, 0) // Enable UART.
eGet(DeviceNum, LJ_ioASYNCH_COMMUNICATION, LJ_chASYNCH_TX, @numBytes, @array) // Write data.
eGet(DeviceNum, LJ_ioASYNCH_COMMUNICATION, LJ_chASYNCH_RX, @numBytes, @array) // Read data.

2.12 Watchdog Timer:

LJ_ioSWDT_CONFIG //Channel is enable or disable constant.

Special channels used with the watchdog config IOType above:
LJ_chSWDT_ENABLE // Value is timeout in seconds (1-65535).
LJ_chSWDT_DISABLE

Special channels, used with the put config IOType.
LJ_chSWDT_RESET_DEVICE
LJ_chSWDT_UDPATE_DIOA
LJ_chSWDT_DIOA_CHANNEL
LJ_chSWDT_DIOA_STATE

// Configure and enable the watchdog. DIOA must be set to output previously.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSWDT_RESET_DEVICE,1,0,0) // Reset on timeout.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSWDT_UDPATE_DIOA,1,0,0) // Update DIOA on timeout.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSWDT_DIOA_CHANNEL,10,0,0) // Specify EIO2 as DIOA.
AddRequest(DeviceNum, LJ_ioPUT_CONFIG, LJ_chSWDT_DIOA_STATE,1,0,0) // Set DIOA high on timeout.
AddRequest(DeviceNum, LJ_ioSWDT_CONFIG, LJ_chSWDT_ENABLE,60,0,0) // Enable watchdog with 60 s timeout.
GoOne(DeviceNum)

// Disable the watchdog.
ePut(DeviceNum, LJ_ioSWDT_CONFIG, LJ_chSWDT_DISABLE,0,0)

2.13 Miscellaneous:

Special channels, used with the get/put config IOTypes, to read/write the calibration memory and user memory:
LJ_chCAL_CONSTANTS // x1 is 20-element array of doubles (note that DF passes bytes). Value must be 0x4C6C to write.
LJ_chUSER_MEM // x1 is 256-element array of bytes.

Miscellaneous special channel constants used with put/get config:
LJ_chCOMMUNICATION_TIMEOUT // Milliseconds driver will allow per call. Default=1500.
LJ_chSTREAM_COMMUNICATION_TIMEOUT // Milliseconds driver will allow per stream read. Default=1500.
LJ_chUSB_STRINGS // Used to write or read USB descriptor strings. See header file.

2.14 Error Handling:

Error handling (which could just mean notification) is very important and should always be considered. The obvious technique is simply to look at the return value of every function call to check for a nonzero errorcode. Another option is the GetNextError() function, which would generally be called anytime after a Go, within a loop, to iterate through all errors. For information about error handling in DAQFactory, see Sections 7.3-7.5 of the DF-LJ Application Guide.

3.1 Errorcodes:

```
LJE_NOERROR = 0;
LJE_INVALID_CHANNEL_NUMBER = 2;   // Attempted to get a result from a channel that was not requested.
LJE_INVALID_RAW_INOUT_PARAMETER = 3;
LJE_UNABLE_TO_START_STREAM = 4;
LJE_UNABLE_TO_STOP_STREAM = 5;
LJE_NOTHING_TO_STREAM = 6;
LJE_UNABLE_TO_CONFIG_STREAM = 7;
LJE_BUFFER_OVERRUN = 8;   // Occurs on UD buffer overrun, not hardware buffer.  Stream is stopped.
LJE_STREAM_NOT_RUNNING = 9;
LJE_INVALID_PARAMETER = 10;
LJE_INVALID_STREAM_FREQUENCY = 11;
LJE_INVALID_AIN_RANGE = 12;
LJE_STREAM_CHECKSUM_ERROR = 13;
LJE_STREAM_COMMAND_ERROR = 14;
LJE_STREAM_ORDER_ERROR = 15;
LJE_AD_PIN_CONFIGURATION_ERROR = 16;
LJE_REQUEST_NOT_PROCESSED = 17;   // … because previous request caused an error.
LJE_SCRATCH_ERROR = 19;
LJE_FLASH_ERROR = 24;
LJE_STREAM_IS_ACTIVE = 25;
LJE_STREAM_TABLE_INVALID = 26;
LJE_STREAM_CONFIG_INVALID = 27;
LJE_STREAM_BAD_TRIGGER_SOURCE = 28;
LJE_STREAM_INVALID_TRIGGER = 30;
LJE_STREAM_ADC0_BUFFER_OVERFLOW = 31;
LJE_STREAM_SCAN_RATE_INVALID = 35;
LJE_TIMER_INVALID_MODE = 36;
LJE_TIMER_STREAM_ACTIVE = 40;
LJE_TIMER_PWMSTOP_MODULE_ERROR = 41;
LJE_TIMER_SHARING_ERROR = 43;
LJE_TIMER_LINE_SEQUENCE_ERROR = 44;
LJE_INVALID_PIN = 48;
LJE_IOTYPE_SYNCH_ERROR = 49;
LJE_INVALID_OFFSET = 50;
LJE_SHT_MEASREADY = 53;
LJE_SHT_ACK = 54;
LJE_SHT_SERIAL_RESET = 55;
LJE_SHT_COMMUNICATION = 56;
LJE_AIN_WHILE_STREAMING = 57;
LJE_STREAM_TIMEOUT = 58;
LJE_STREAM_CONTROL_BUFFER_OVERFLOW = 59;
LJE_STREAM_SCAN_OVERLAP = 60;
LJE_FIRMWARE_VERSION_IOTYPE = 61;
LJE_FIRMWARE_VERSION_CHANNEL = 62;
LJE_FIRMWARE_VERSION_VALUE = 63;
LJE_HARDWARE_VERSION_IOTYPE = 64;
LJE_HARDWARE_VERSION_CHANNEL = 65;
LJE_HARDWARE_VERSION_VALUE = 66;
LJE_CANT_CONFIGURE_PIN_FOR_ANALOG = 67;
LJE_CANT_CONFIGURE_PIN_FOR_DIGITAL = 68;
LJE_LJTDAC_ACK_ERROR = 69;
LJE_ TC_PIN_OFFSET_MUST_BE_4_8 = 70;
LJE_MIN_GROUP_ERROR = 1000;   // Errors above this number will stop all requests.  Below this number are request level errors.
LJE_UNKNOWN_ERROR = 1001;   // Hardware threw an error not recognized by the driver.
LJE_INVALID_DEVICE_TYPE = 1002;
LJE_INVALID_HANDLE = 1003;
LJE_DEVICE_NOT_OPEN = 1004;
LJE_NO_DATA_AVAILABLE = 1005;   // Result attempted for data that was not requested.
LJE_NO_MORE_DATA_AVAILABLE = 1006;
LJE_LABJACK_NOT_FOUND = 1007;
LJE_COMM_FAILURE = 1008;
LJE_CHECKSUM_ERROR = 1009;
LJE_DEVICE_ALREADY_OPEN = 1010;
LJE_COMM_TIMEOUT = 1011;
LJE_USB_DRIVER_NOT_FOUND = 1012;
LJE_INVALID_CONNECTION_TYPE = 1013;
LJE_INVALID_MODE = 1014;   // Device is likely in flash programming mode.
LJE_DISCONNECT = 2000;
LJE_RECONNECT = 2001;
LJE_MIN_USER_ERROR = 3000;
LJE_MAX_USER_ERROR = 3999;
LJE_DEVICE_NOT_CALIBRATED = -1;   // Warning.  Default cal constants will be used instead of actual.
LJE_UNABLE_TO_READ_CALDATA = -2;  // Warning.  Default cal constants will be used instead of actual.
```