

LabJackUD Driver for Windows

Revision 1.01
5/16/2005

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

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

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

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

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

Reproduction of products or written or electronic information from LabJack Corporation is prohibited without permission. Reproduction of any of these with alteration is an unfair and deceptive business practice.

Copyright © 2005, LabJack Corporation

Document Revision History

V1.00 released 3/31/2005

V1.01 released 5/16/2005

Section 3.9 – Corrected error in DoubleToStringAddress() declaration.

Table Of Contents

1. Installation	1
2. Overview	2
2.1 General Operation	2
2.2 Function Flexibility	3
2.3 Recommended Programming Practices:	3
2.4 Basic Analog Input Example	6
2.5 Stream Example	7
2.6 Multi-Threaded Operation	10
2.7 LabVIEW	11
2.8 Visual Basic (VB/VBA)	11
3. Function Reference	12
3.1 ListAll()	12
3.2 OpenLabJack()	13
3.3 eGet() and ePut()	14
3.4 AddRequest()	14
3.5 Go()	16
3.6 GoOne()	16
3.7 GetResult()	16
3.8 GetFirstResult() and GetNextResult()	17
3.9 DoubleToStringAddress()	18
3.10 StringToDoubleAddress()	18
3.11 StringToConstant()	19
3.12 ErrorToString()	19
3.13 GetDriverVersion()	19
3.14 ResetLabJack()	20
4. Constants	21

Tables

Table 4-1. DeviceType Constants	21
Table 4-2. ConnectionType Constants	21
Table 4-3. IOType Constants	22
Table 4-4. Special Channels	23
Table 4-5. Other Constants	24
Table 4-6. Error Codes	25

1. Installation

This document describes the LabJackUD driver for Windows. Currently this driver supports the LabJack UE9. Support is planned for all future LabJacks.

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

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

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

2. Overview

This document describes the LabJack driver for Windows. Currently this driver supports the LabJack UE9.

2.1 General Operation

The general operation of the functions is as follows:

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

For example, to set the resolution on the analog inputs:

```
//Open the first found LabJack UE9.
errorcode = OpenLabJack (LJ_dtUE9, LJ_ctUSB, 0, 1, &Handle);

//Configure for 16-bit analog input measurements.
errorcode = AddRequest (Handle, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 16, 0, 0);

//Execute the request.
errorcode = GoOne (Handle);

//Get the configuration result just to check for an errorcode.
errorcode = GetResult (Handle, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 0);
```

Multiple requests can be executed with a single Go() or GoOne() call. By combining requests, the driver might be able to optimize them into fewer low level calls. If speed optimization is not a concern in a simple application, eGet and ePut can be used. These functions combine the AddRequest, Go, and GetResult functions into one call. For example, to configure the resolution of the analog inputs as was done above:

```
//Open the first found LabJack UE9.
errorcode = OpenLabJack (LJ_dtUE9, LJ_ctUSB, 0, 1, &Handle);

//Configure for 16-bit analog input measurements in one call
errorcode = ePut (Handle, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 16, 0);
```

This obviously looks simpler, but there are a few reasons why these functions might not be the best approach in a particular application:

- 1) This method is slightly slower just in the amount of code that has to be executed.
- 2) This method does not allow the driver to perform optimizations. For example, if 4 analog inputs are read with 4 eGet calls, the driver will make four low level calls to the LabJack device instead of the one required if a list of requests is built followed by a single Go call. This is especially important when bandwidth is a factor.
- 3) For more complicated applications, the special GetFirstResult/GetNextResult functions can be used to create a generic result parser, resulting in cleaner and more efficient code.

Most of the examples provided here will use the Add/Go/Get method rather than the e functions.

2.2 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 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_ioGETCONFIG, LJ_ioHARDWARE_VERSION,0,0,0);
```

The bad way (hard to read) when LabJackUD.h cannot be included:

```
AddRequest(Handle, 1001, 10, 0, 0, 0);
```

The better way when LabJackUD.h cannot be included:

```
AddRequestSS(Handle, "LJ_ioGETCONFIG", "LJ_ioHARDWARE_VERSION",0,0,0);
```

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

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

Once again, this is much clearer than:

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

2.3 Recommended Programming Practices:

Because there are many different languages, and many different ways this driver will be used, only generic error handling and data parsing is provided. The driver is designed, however, to allow the creation of custom functions to handle this very easily. If creating anything more than a basic application reading or writing a few values, it is recommend to create functions for adding requests and getting results similar to those shown here. Of course, the structure of the particular application and data flow will determine whether these exact techniques will work well.

For adding requests, create a function that looks something like this:

```

err AddMyRequest(LJ_HANDLE Handle, long IOType, long Channel, double Value, double x1,
double UserData)
{
    LJ_ERROR err = AddRequest(Handle, IOType, Channel, Value, x1, UserData);
    if (err != LJE_NOERROR)
        ... put error handling here
    return err;
}

```

By creating a function that has pretty much the same prototype as the LabJack Driver, all error handling can be taken care of in one place. This is especially useful to just display or log an error message. Since this routine returns the error code as well, the calling function can do further handling if needed.

A generic result parser can be created with the GetFirstResult() / GetNextResult() functions:

```

void GetResults(LJ_HANDLE Handle)
{
    LJ_ERROR err;
    long IOType,Channel;
    double value,x1,userdata;

    // do all the requests:
    GoOne(Handle);

    // get the first result:
    err = GetFirstResult(Handle,&IOType,&Channel,&value,&x1,&userdata);
    // loop while we don't have a group type error
    while (err < LJE_MIN_GROUP_ERROR)
    {
        if (err != LJE_NOERROR)
            ...do error handling for a request level error
        else
        {
            switch (IOType)
            {
                case LJ_ioANALOG_INPUT :
                    .. do stuff with analog input value read
                    break;
                case LJ_ioANALOG_OUTPUT :
                    .. do stuff with analog output value
                    break;
                ...
            }
        }
        err = GetNextResult(Handle,&IOType,&Channel,&value,&x1,&userdata);
    }
    if (err != LJE_NO_MORE_DATA_AVAILABLE)
        ...do error handling for group error
}

```

This function will call the driver to execute all the requests, and then parse them one by one doing error handling along the way. There are two types of errors: group and request level. A group error causes all further requests to stop. This is usually communication type errors. The LJE_NO_MORE_DATA_AVAILABLE error is returned when all the requests have been processed, and is considered a group type error. That is why the snippet above loops until there is an error greater than the minimum group error number. Inside the loop request level errors are handled, and if none, the request is processed with the IOType, Channel, value, x1 and userdata returned from the GetFirstResult() / GetNextResult() functions.

Here is an example of how these functions might be used. Say an application has two buttons, one for reading analog input channels 2 and 3 and the other for reading 4 and 5, and that the LabJack has already been opened elsewhere and the handle stored in `m_Handle`:

```
void OnButton23Press()
{
    AddMyRequest(m_Handle,LJ_ioANALOG_INPUT,2,0,0,0);
    AddMyRequest(m_Handle,LJ_ioANALOG_INPUT,3,0,0,0);
    GetResults(m_Handle);
}

void OnButton45Press()
{
    AddMyRequest(m_Handle,LJ_ioANALOG_INPUT,2,0,0,0);
    AddMyRequest(m_Handle,LJ_ioANALOG_INPUT,3,0,0,0);
    GetResults(m_Handle);
}
```

Since all the error handling and parsing is handled in the custom functions, the code is very simple. Here is what it would look like otherwise:

```
void OnButton23Press()
{
    long err;
    double value;
    err = AddRequest(m_Handle,LJ_ioANALOG_INPUT,2,0,0,0);
    if (err != LJE_NOERROR)
        ... put error handling here
    err = AddRequest(m_Handle,LJ_ioANALOG_INPUT,3,0,0,0);
    if (err != LJE_NOERROR)
        ... put the same error handling here again
    GoOne(m_Handle);
    err = GetResult(m_Handle,LJ_ioANALOG_INPUT,2,&Value);
    if (err != LJE_NOERROR)
        ... put the same error handling here again
    else
        ...do stuff with the analog input value read
    err = GetResult(m_Handle,LJ_ioANALOG_INPUT,3,&Value);
    if (err != LJE_NOERROR)
        ... put the same error handling here again
    else
        ...do stuff with the analog input value read
}

void OnButton45Press()
{
    long err;
    double value;
    err = AddRequest(m_Handle,LJ_ioANALOG_INPUT,4,0,0,0);
    if (err != LJE_NOERROR)
        ... put error handling here
    err = AddRequest(m_Handle,LJ_ioANALOG_INPUT,5,0,0,0);
    if (err != LJE_NOERROR)
        ... put the same error handling here again
    GoOne(m_Handle);
    err = GetResult(m_Handle,LJ_ioANALOG_INPUT,4,&Value);
    if (err != LJE_NOERROR)
        ... put the same error handling here again
    else
        ...do stuff with the analog input value read
    err = GetResult(m_Handle,LJ_ioANALOG_INPUT,5,&Value);
    if (err != LJE_NOERROR)
        ... put the same error handling here again
    else
        ...do stuff with the analog input value read
}
```

```
}
```

Obviously, the first way is much cleaner. It has less duplicated code and it is easy to change the error handling. This can be particularly important as the application gets bigger.

The UserData Parameter:

This mechanism, of course, assumes that every time a particular item is written or read the same thing will be done with it, and most applications work that way. Every time an analog input is read, it is put in a variable somewhere, displayed on the screen, graphed, or something similar. If different things are going to be done at different times, a similar structure is still recommended. This is the purpose of the UserData parameter. This parameter is not used by the driver at all, and is simply passed through to the requests. It can be used to store any sort of information with the request to allow a generic parser to determine what it should do when it receives the results.

2.4 Basic Analog Input Example

The following pseudo code shows simple reads from analog inputs. Some of the recommended methods explained above will be ignored so the general technique can be demonstrated simply.

```
//Open the first found LabJack UE9.
errorcode = OpenLabJack (LJ_dtUE9, LJ_ctUSB, 0, 1, &Handle);

//Use an e function to request and retrieve a reading from AIN5.
//Since no configuration requests have been made, defaults will
//apply for resolution, range, etc.
errorcode = eGet (Handle, LJ_ioANALOG_INPUT, 5, &Value5, 0);
```

The single eGet() call retrieves a reading from an analog input and returns it in "Value5". Now for a more involved read from a pair of analog inputs.

```
//Configure for 16-bit analog input measurements. This setting will apply to all
//further measurements until the parameter is changed or the DLL unloaded.
errorcode = AddRequest (Handle, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 16, 0, 0);

//Configure the analog input range on channels 2 and 3 for bipolar +-5 volts.
errorcode = AddRequest (Handle, LJ_ioPUT_AIN_RANGE, 2, LJ_rgBIP5V, 0, 0);
errorcode = AddRequest (Handle, LJ_ioPUT_AIN_RANGE, 3, LJ_rgBIP5V, 0, 0);

//Request AIN2 and AIN3.
errorcode = AddRequest (Handle, LJ_ioANALOG_INPUT, 2, 0, 0, 0);
errorcode = AddRequest (Handle, LJ_ioANALOG_INPUT, 3, 0, 0, 0);

//Execute the requests.
errorcode = GoOne (Handle);

//Get the configuration results just to check for errorcodes.
errorcode = GetResult (Handle, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, NULL);
errorcode = GetResult (Handle, LJ_ioPUT_AIN_RANGE, 2, NULL);
errorcode = GetResult (Handle, LJ_ioPUT_AIN_RANGE, 3, NULL);

//Get the analog input voltage readings.
errorcode = GetResult (Handle, LJ_ioANALOG_INPUT, 2, &Value2);
errorcode = GetResult (Handle, LJ_ioANALOG_INPUT, 3, &Value3);
```

Then, execution of the request list could be repeated to get another sample from AIN2 and AIN3.

```
errorcode = GoOne (Handle);
errorcode = GetResult (Handle, LJ_ioANALOG_INPUT, 2, &Value2);
errorcode = GetResult (Handle, LJ_ioANALOG_INPUT, 3, &Value3);
```

To sample a single different channel, do the following.

```

//Request AIN9. Since this is the first AddRequest() call after a Go function
//was called, the list of requests will be cleared and this will become the 1st
//and only request.
errorcode = AddRequest (Handle, LJ_ioANALOG_INPUT, 9, 0, 0, 0);

//Execute the request.
errorcode = GoOne (Handle);

//Get the analog input voltage reading.
errorcode = GetResult (Handle, LJ_ioANALOG_INPUT, 9, &Value);

```

Once again, we can repeat to get another reading.

```

errorcode = GoOne (Handle);
errorcode = GetResult (Handle, LJ_ioANALOG_INPUT, 9, &Value);

```

Reading other inputs, setting outputs or configuration information, is all done with the same basic idea, but different parameters.

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 type settings like ranges, stream settings, and such, will be done before the actual acquisition or setting of outputs. This is why the above example is acceptable, where the range is set and analog input acquired in the same Go call.

2.5 Stream Example

Stream mode is where the LabJack acquires inputs at a fixed interval controlled by the hardware clock on the device itself. The data is stored in a buffer where it can be retrieved at convenient intervals. 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.

Configuration parameters are set using the IOType LJ_ioPUT_CONFIG with special channel numbers that correspond to stream settings. Typically, scan frequency and buffer size are the only parameters required. Normal analog input parameters such as resolution and settling time also apply to streamed channels. Normal per channel settings such as range also apply to streamed channels.

Once streaming starts, scans will be acquired at the specified scan frequency, and a scan consists of a sample from each item in the scan list. The scan list is generally built by using a clear channels request first, followed by one or more add channel requests to add the desired channels. A single request with IOType LJ_ioCLEAR_STREAM_CHANNELS clears all channels from the scan list.

Start the stream using a request with IOType LJ_ioSTART_STREAM. If this is successful, the LabJack will start streaming data and the driver will use a background thread to transfer data to a RAM buffer on the PC.

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 a 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 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. Also note that 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.

Although more convenient in many applications, requesting individual channels is slightly less efficient than using LJ_chALL_CHANNELS.

Execution of a Go call will cause the desired amount of data (or less depending on wait mode) to be placed in the buffer pointed to by x1. There are currently four different 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. Since the LabJack clock could be faster than the PC clock, it is recommended to request more scans than are expected each time.
- 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.
- LJ_swPUMP and 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. 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.

The difference between PUMP and SLEEP modes is how Go command blocks. When in pump mode, the driver will constantly run the Windows message pump while waiting for the appropriate amount of data. This should be used in the main application thread or any secondary thread that contains a message pump. The sleep mode simply uses the windows Sleep() function to yield the CPU to other running threads for a given period of time. This should be used in secondary worker threads that do not have a message pump. If the wrong mode is used, an application will likely appear to hang while waiting for data. If no threads are being used, then PUMP is the appropriate choice, but not all development environments necessarily support this method.

Sample:

The following pseudo code shows a stream from two analog inputs. Once again the recommended methods explained in Section 2.3 (and all error handling) are ignored so the basic technique can be shown in a linear fashion.

First, configure the stream:

```
// Open the first found LabJack UE9.
// If the LabJack is already open, we don't need to do this
errorcode = OpenLabJack (LJ_dtUE9, LJ_ctUSB, 0, 1, &Handle);

// Configure the analog input range on channels 2 and 3 for bipolar +/- 5 volts.
errorcode = AddRequest (Handle, LJ_ioPUT_AIN_RANGE, 2, LJ_rgBIP5V, 0, 0);
errorcode = AddRequest (Handle, LJ_ioPUT_AIN_RANGE, 3, LJ_rgBIP5V, 0, 0);

// Set internal driver buffer to 40000 samples. We must set this before performing
// the first stream, as the buffer defaults to 0 to save memory.
errorcode = AddRequest (Handle, LJ_ioPUT_CONFIG, LJ_chSTREAM_BUFFER_SIZE, 40000, 0, 0);

// Configure stream for 10000 scans/second (20000 samples/second in this case).
errorcode = AddRequest (Handle, LJ_ioPUT_CONFIG, LJ_chSTREAM_SCAN_FREQUENCY, 10000, 0, 0);

// set the wait mode to wait for the requested scans automatically
errorcode = AddRequest (Handle, LJ_ioPUT_CONFIG, LJ_chSTREAM_WAIT_MODE, LJ_swPUMP, 0, 0);

// Add AIN2 and AIN3 to the scan list.
errorcode = AddRequest (Handle, LJ_ioCLEAR_STREAM_CHANNELS, 0, 0, 0, 0);
errorcode = AddRequest (Handle, LJ_ioADD_STREAM_CHANNEL, 2, 0, 0, 0);
errorcode = AddRequest (Handle, LJ_ioADD_STREAM_CHANNEL, 3, 0, 0, 0);

// Execute the requests.
errorcode = GoOne (Handle);

// Get the configuration results just to check for errorcodes.
errorcode = GetResult (Handle, LJ_ioPUT_AIN_RANGE, 2, NULL);
errorcode = GetResult (Handle, LJ_ioPUT_AIN_RANGE, 3, NULL);
errorcode = GetResult (Handle, LJ_ioPUT_CONFIG, LJ_chSTREAM_BUFFER_SIZE, NULL);
errorcode = GetResult (Handle, LJ_ioPUT_CONFIG, LJ_chSTREAM_SCAN_FREQUENCY, NULL);
errorcode = GetResult (Handle, LJ_ioCLEAR_STREAM_CHANNELS, 0, NULL);
errorcode = GetResult (Handle, LJ_ioADD_STREAM_CHANNEL, 2, NULL);
errorcode = GetResult (Handle, LJ_ioADD_STREAM_CHANNEL, 3, NULL);
```

Next, start the stream:

```
// Do this the quick way, since there is only one request:
errorcode = ePut (Handle, LJ_ioSTART_STREAM, 0, 0, 0);
```

Now read data continuously until done. This example reads 5000 scans at a time, so a read will be returned about every half second:

```
// create the arrays:
double Array2[5000];
double Array3[5000];

// cast the array pointer into a long
long pArray2 = (long)&Array2[0];
long pArray3 = (long)&Array3[0];

//Add AIN2 and AIN3 to the scan list.
errorcode = AddRequest (Handle, LJ_ioGET_STREAM_DATA, 2, 5000, pArray2, 0);
errorcode = AddRequest (Handle, LJ_ioGET_STREAM_DATA, 3, 5000, pArray3, 0);

//Read 5000 scans at a time repeatedly until done.
while(!done)
{
```

```

//Execute the requests to read data.
errorcode = GoOne (Handle);

//Find out how many points were returned.
errorcode = GetResult (Handle, LJ_ioGET_STREAM_DATA, 2, &numSamples2);
errorcode = GetResult (Handle, LJ_ioGET_STREAM_DATA, 3, &numSamples3);

// if we get nothing, then we had a timeout, which in PUMP or SLEEP mode
// more than likely means the device was disconnected, reset, or the
// stream was stopped by another thread
if (errorcode == 0)
    break;
// The data is now in Array2 and Array3. Process, display, write to file, or
// whatever else you want.

// since we are in the PUMP wait mode, we don't have to put any waits in our
// loop, as the driver will take care of it. If we were using NONE or
// ALL_OR_NONE, we'd have to either sleep or run the message pump ourselves
// or our application would completely hang.

// this code assumes the variable "done" is set elsewhere, presumably by a
// button or other user interface control.
}

```

Finally, stop the stream:

```

// Do this the quick way, since there is only one request:
errorcode = ePut (Handle, LJ_ioSTOP_STREAM, 0, 0, 0);

```

2.6 Multi-Threaded Operation

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

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

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

The one big exception to the thread safety of this driver is in the use of the Windows TerminateThread() function. As is warned in the MSDN documentation, using TerminateThread() will kill the thread without releasing any resources, and more importantly, releasing any synchronization objects. If TerminateThread() is used on a thread that is currently in the middle of a call to this driver, more than likely a synchronization object will be left open on the particular device

and access to the device will be impossible until the application is restarted. On some devices, it can be worse. On devices that have interprocess synchronization, such as the U12, calling `TerminateThread()` may kill all access to the device through this driver no matter which process is using it and even if the application is restarted. Avoid using `TerminateThread()`! All device calls have a timeout, which defaults to 1 second, but can be changed. Make sure to wait at least as long as the timeout for the driver to finish.

2.7 LabVIEW

LabVIEW VIs are available to interface with the LabJackUD Windows driver. See the `readme.txt` file in the separate download titled "LabVIEW_LJUD".

2.8 Visual Basic (VB/VBA)

There is a download available at labjack.com titled "VisualBasic_LJUD". Included in this download is a VB module for the LabJackUD driver. This module must be included in any project that will talk to the LabJackUD driver. Similar to a header file in C, this module declares the functions and constants. See the comments in the VB or VBA examples.

3. Function Reference

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

Some parameters are common to many functions:

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

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

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

3.1 ListAll()

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

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

Declaration:

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

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

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

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

Outputs:

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

3.2 OpenLabJack()

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

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

Declaration:

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

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

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

Outputs:

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

3.3 eGet() and ePut()

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

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

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

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

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

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

Declaration:

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

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

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

Outputs:

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

3.4 AddRequest()

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

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

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

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

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

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

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 ...
- **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().

Outputs:

- **None**

3.5 Go()

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

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

Declaration:

LJ_ERROR _stdcall Go()

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

- **None**

Outputs:

- **None**

3.6 GoOne()

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

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

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**

3.7 GetResult()

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

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

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

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

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

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

Declaration:

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

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

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

Outputs:

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

3.8 GetFirstResult() and GetNextResult()

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

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

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

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

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

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

Declaration:

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

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

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

Outputs:

- **pIOType** – A pointer to the IOType of this item in the list.
- **pChannel** – A pointer to the channel number of this item in the list.
- **pValue** – A pointer to the result value.
- **px1** – A pointer to the x1 parameter of this item in the list.
- **pUserData** – A pointer to data that is simply passed along with the request, and returned unmodified.

3.9 DoubleToStringAddress()

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

Declaration:

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

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

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

Outputs:

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

3.10 StringToDoubleAddress()

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

Declaration:

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

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

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

Outputs:

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

3.11 StringToConstant()

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

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

This function returns LJ_INVALID_CONSTANT if the string is not recognized.

Declaration:

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

Parameter Description:

Returns: Constant number of the passed string.

Inputs:

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

Outputs:

- **None**

3.12 ErrorToString()

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

Declaration:

```
void _stdcall ErrorToString ( LJ_ERROR ErrorCode,  
                             char *pString)
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

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

Outputs:

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

3.13 GetDriverVersion()

Returns the version number of this Windows LabJack driver.

Declaration:

```
double _stdcall GetDriverVersion();
```

Parameter Description:

Returns: Driver version.

Inputs:

- **None**

Outputs:

- **None**

3.14 ResetLabJack()

Sends a reset command to the LabJack hardware.

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

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

Declaration:

```
LJ_ERROR _stdcall ResetLabJack ( LJ_HANDLE Handle);
```

Parameter Description:

Returns: LabJack errorcodes or 0 for no error.

Inputs:

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

Outputs:

- **None**

4. Constants

This section covers various constants used by the driver.

DeviceType is used by OpenLabJack(), and specifies the LabJack model number to open.

<u>DeviceType</u>	<u>Name</u>	<u>Description</u>
3	LJ_dtU3	U3 (Future device)
9	LJ_dtUE9	UE9

Table 4-1. DeviceType Constants

ConnectionType is used by OpenLabJack(), and specifies whether to use USB or Ethernet on devices that have both.

<u>ConnectionType</u>	<u>Name</u>	<u>Description</u>
1	LJ_ctUSB	USB
2	LJ_ctETHERNET	Ethernet

Table 4-2. ConnectionType Constants

IOType is use by AddRequest() to specify what type of request is being added to the list.

<u>IOType</u>	<u>Name</u>	<u>Channel</u>	<u>Value</u>	<u>x1</u>
10	LJ_ioANALOG_INPUT	Specify	Value	Unused
20	LJ_ioANALOG_OUTPUT	Specify	Value	Unused
30	LJ_ioDIGITAL_BIT_IN	Specify	Value	Unused
35	LJ_ioDIGITAL_PORT_IN	Specify	Value	Specify
40	LJ_ioDIGITAL_BIT_OUT	Specify	Value	Unused
45	LJ_ioDIGITAL_PORT_OUT	Specify	Value	Specify
50	LJ_ioCOUNTER	Specify	Value	Unused
60	LJ_ioTIMER	Specify	Optional	Unused
100	LJ_ioRAW_OUT	Specify	Specify	Buffer
101	LJ_ioRAW_IN	Specify	Specify	Buffer
200	LJ_ioADD_STREAM_CHANNEL	Specify	Unused	Unused
201	LJ_ioCLEAR_STREAM_CHANNELS	Unused	Unused	Unused
202	LJ_ioSTART_STREAM	Unused	Unused	Unused
203	LJ_ioSTOP_STREAM	Unused	Unused	Unused
204	LJ_ioGET_STREAM_DATA	Specify	#Scans	Buffer
1000	LJ_ioPUT_CONFIG	Special	Value	Unused
1001	LJ_ioGET_CONFIG	Special	Value	Unused
2000	LJ_ioPUT_AIN_RANGE	Specify	Value	Unused
2001	LJ_ioGET_AIN_RANGE	Specify	Value	Unused
2002	LJ_ioPUT_DAC_ENABLE	Specify	Value	Unused
2003	LJ_ioGET_DAC_ENABLE	Specify	Value	Unused
2004	LJ_ioPUT_TIMER_MODE	Specify	Value	Unused
2005	LJ_ioGET_TIMER_MODE	Specify	Value	Unused
2006	LJ_ioPUT_TIMER_VALUE	Specify	Value	Unused
2007	LJ_ioGET_TIMER_VALUE	Specify	Value	Unused
2008	LJ_ioPUT_COUNTER_ENABLE	Specify	Value	Unused
2009	LJ_ioGET_COUNTER_ENABLE	Specify	Value	Unused
2010	LJ_ioPUT_COUNTER_MODE	Specify	Value	Unused
2011	LJ_ioGET_COUNTER_MODE	Specify	Value	Unused
2012	LJ_ioPUT_COUNTER_RESET	Specify	Value	Unused

Table 4-3. IOType Constants

Raw Out/In: Used to communicate with a device using its low level protocol. Channel # corresponds to the particular communication path, which depends on the device. For example, on the UE9, 0 is main communication path, and 1 is the stream communication path. Pass the number of elements (bytes) in Value, and pass a pointer to a char (U8) array in x1. The array must be initialized to the proper size. The Go call will cause the data to be sent to the device (RawOut) and/or wait until data is read from the device or timeout (RawIn). In the case of input, any available data will be placed in the x1 array, and a get result call will return the number of bytes in this array.

The following table covers special channels that have a meaning besides a simple a channel of input or output. These are generally used with the configuration IOTypes.

LJ_chALL_CHANNELS is currently only supported with the Get_Stream_Data IO Type, and specifies that all channels will be read rather than just one specific channel. All other special channels are passed with IOType LJ_ioPUT_CONFIG or LJ_ioGET_CONFIG to specify the desired configuration parameter.

An “X” in the NV column signifies a non-volatile parameter. Non-volatile memory generally has a limited lifetime (20,000 writes on the UE9), so avoid writing these parameters too often. An “RO” in the NV column signifies a read-only parameter.

<u>Special Ch.</u>	<u>Name</u>	<u>Value</u>	<u>x1</u>	<u>NV</u>
-999	LJ_INVALID_CONSTANT	Unused	Unused	
-1	LJ_chALL_CHANNELS	Unused	Unused	
0	LJ_chLOCALID	Value	Unused	X
1	LJ_chCOMM_POWER_LEVEL	Value	Unused	X
2	LJ_chIP_ADDRESS	Value	Unused	X
3	LJ_chGATEWAY	Value	Unused	X
4	LJ_chSUBNET	Value	Unused	X
5	LJ_chPORTA	Value	Unused	X
6	LJ_chPORTB	Value	Unused	X
7	LJ_chDHCP	Value	Unused	X
8	LJ_chPRODUCTID	Value	Unused	RO
9	LJ_chMACADDRESS	Value	Unused	RO
10	LJ_chHARDWARE_VERSION	Value	Unused	RO
11	LJ_chCOMM_FIRMWARE_VERSION	Value	Unused	RO
12	LJ_chSERIAL_NUMBER	Value	Unused	RO
13	LJ_chCONTROL_POWER_LEVEL	Value	Unused	X
14	LJ_chCONTROL_FIRMWARE_VERSION	Value	Unused	RO
15	LJ_chCONTROL_BOOTLOADER_VERSION	Value	Unused	RO
16	LJ_chCONTROL_RESET_SOURCE	Value	Unused	
1000	LJ_chNUMBER_TIMERS_ENABLED	Value	Unused	
1001	LJ_chTIMER_CLOCK_CONFIG	Value	Unused	
1002	LJ_chTIMER_CLOCK_DIVISOR	Value	Unused	
2000	LJ_chAIN_RESOLUTION	Value	Unused	
2001	LJ_chAIN_SETTLING_TIME	Value	Unused	
2002	LJ_chAIN_BINARY	Value	Unused	
3000	LJ_chDAC_BINARY	Value	Unused	
4000	LJ_chSTREAM_SCAN_FREQUENCY	Value	Unused	
4001	LJ_chSTREAM_BUFFER_SIZE	Value	Unused	
4002	LJ_chSTREAM_CLOCK_OUTPUT	Value	Unused	
4003	LJ_chSTREAM_EXTERNAL_TRIGGER	Value	Unused	
4004	LJ_chSTREAM_WAIT_MODE	Value	Unused	
4105	LJ_chSTREAM_BACKLOG_COMM	Value	Unused	
4106	LJ_chSTREAM_BACKLOG_CONTROL	Value	Unused	

Table 4-4. Special Channels

The following table covers other constants that are used with IOTypes or Special Channels.

<u>Other Constants</u>	<u>Name</u>	<u>UE9</u>	<u>Description</u>
1	LJ_rgBIP20V		±20 volts
2	LJ_rgBIP10V		±10 volts
3	LJ_rgBIP5V	X	±5 volts
4	LJ_rgBIP4V		±4 volts
5	LJ_rgBIP2P5V		±2.5 volts
6	LJ_rgBIP2V		±2 volts
7	LJ_rgBIP1P25V		±1.25 volts
8	LJ_rgBIP1V		±1 volts
9	LJ_rgBIPP625V		±0.625 volts
101	LJ_rgUNI20V		0-20 volts
102	LJ_rgUNI10V		0-10 volts
103	LJ_rgUNI5V	X	0-5 volts
104	LJ_rgUNI4V		0-4 volts
105	LJ_rgUNI2P5V	X	0-2.5 volts
106	LJ_rgUNI2V		0-2 volts
107	LJ_rgUNI1P25V	X	0-1.25 volts
108	LJ_rgUNI1V		0-1 volts
109	LJ_rgUNIP625V	X	0-0.625 volts
110	LJ_rgUNIP500V		0-0.5 volts
111	LJ_rgUNIP3125V		0-0.3125 volts
0	LJ_tmPWM16	X	16-bit PWM
1	LJ_tmPWM8	X	8-bit PWM
2	LJ_tmRISINGEDGES32	X	32-bit rising to rising edge measurement
3	LJ_tmFALLINGEDGES32	X	32-bit falling to falling edge measurement
4	LJ_tmDUTYCYCLE	X	Duty cycle measurement
5	LJ_tmFIRMCOUNTER	X	Firmware based rising edge counter
7	LJ_tmFREQOUT	X	Frequency output
8	LJ_tmQUAD	X	Quadrature
9	LJ_tmTIMERSTOP	X	Stops another timer after n pulses
10	LJ_tmSYSTIMERLOW	X	Read lower 32-bits of system timer
11	LJ_tmSYSTIMERHIGH	X	Read upper 32-bits of system timer
12	LJ_tmRISINGEDGES16	X	16-bit rising to rising edge measurement
13	LJ_tmFALLINGEDGES16	X	16-bit falling to falling edge measurement
0	LJ_tc750KHZ	X	Fixed 750 kHz timer clock
1	LJ_tcSYS	X	System clock is timer clock
1	LJ_swNONE	X	No wait. Return amount available.
2	LJ_swALL_OR_NONE	X	No wait. Return amount requested or none.
11	LJ_swPUMP	X	Wait using message pump.
12	LJ_swSLEEP	X	Wait by sleeping.

Table 4-5. Other Constants

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

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

Table 4-6. Error Codes

Note: some errors are specific to a request. For example, LJE_INVALID_CHANNEL_NUMBER. If this error occurs, other requests are not affected. Other errors, such as Comm Failure will cause all pending requests for a particular Go() to fail with the same error. If you receive this type of error you can not assume the state of any of the requests. For example, if you set the A to D range and read an A to D channel with two requests in a single Go() and the A to D read fails with a comm failure, you cannot determine whether the A to D range was set to your new value or whether it is still set at the old value.