

LJM User's Guide

[Add new comment](#)

Welcome to the LJM User's Guide! This document describes the API and usage of the [LabJack LJM library](#). The LabJack LJM library is a cross platform library that allows programs to read and write Modbus registers implemented on a variety of LabJack devices. A searchable list of all of the supported registers that can be read from or written to is available on our [Modbus Map](#) page.

Supported Devices

LJM supports T-series devices:

- [T4](#)
- [T7](#)

LJM also supports the Digit, which is deprecated.

Software support for other devices can be found on the [Software page](#).

LJM Download

See here for the [LJM Installer](#).

Examples

[Example code](#) for LJM is available in Python, LabVIEW, C/C++, and other languages.

Reading This Document

Table of Contents Note

Navigating using the Table of Contents

An efficient way to navigate this online document is to use the Table of Contents button to the left.

Offline PDF Note

Offline PDF

If you are looking at a PDF, hardcopy, or other downloaded offline version of this document, realize that it is possibly out-of-date since the original is an online document. Also, this document is designed as online documentation, so the formatting of an offline version might be less than perfect.

To make a PDF of this entire document including all child pages, click "Save as PDF" towards the bottom-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. Make sure you have a current browser (we mostly test in Firefox and Chrome) and the current version of Acrobat Reader. If it is not working for you, rather than a normal click of "Save as PDF" do a right-click and select "Save link as" or similar.

Then wait 20-30 seconds and a dialog box will pop up asking you where to save the PDF. Then you can open it in the real Acrobat Reader rather than embedded in a browser.

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

- We can quickly improve and update content.
- Click-able links to further or related details throughout the online document.
- The [site search](#) includes this document, the 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 page and post a comment. When appropriate we can then immediately add/change content on that page to address the question.

Periodically we use the "Save as PDF" feature to export a PDF and attach it to this page (below).

File Attachment:

 [LabJack-LJM-Library-Docs-Export-20160914.pdf](#)

1 - Overview

The LJM Library is a set of functions used to easily communicate with several of our devices using a simple Modbus interface. The goal is to be easy to use and understand, yet flexible.

All important values & data from the device can be read and/or written by using the associated register(s). Thus, the process for reading the serial number, an analog input, or a timer is all functionally the same, you simply provide a different address. We have also included functionality so that values can be specified using a name instead of numerical value, such as "AIN6" to represent the register(s) that will read analog input number 6.

<p>Begin with the open function.</p>	<p>Each device is represented by a handle which is obtained by using one of the open functions to open that device. That handle is passed to the other functions to specify that those functions should operate on that device.</p> <p style="text-align: right;">More on Opening</p>
<p>Use the read/write functions.</p>	<p>The read/write functions provide a simple interface for reading/writing data by either specifying a name or an address.</p> <p style="text-align: right;">More on Reading/Writing</p>
<p>Use the stream functions.</p>	<p>The stream functions allow data to be collected at a constant and/or fast rate.</p> <p style="text-align: right;">More on streaming</p>
<p>More functions.</p>	<p>There are also utility functions provided to do things like debugging and making unit conversions easier, as well as a few functions that let you configure the library itself.</p> <p style="text-align: right;">More on Other Functions</p>
<p>Error codes describe the problem.</p>	<p>A set of constants are defined for specific use within the library to represent various values, and a robust set of error codes that are returned when something goes wrong.</p> <p style="text-align: right;">More on Error Codes</p>

2 - Function Reference

2.1 - Opening and Closing

Summary

Open a LabJack device to communicate with it. Close it to allow other processes to open it.

Opening

Either `LJM_Open` or `LJM_OpenS` must be used to open a LabJack. The `LJM_Open` and `LJM_OpenS` functions will detect a LabJack connected to the computer, and create a device handle. The device handle is then passed as an input to other functions.

- `LJM_Open` - Open a device based on integer filter parameters.
- `LJM_OpenS` - Open a device based on string filter parameters.

Use whichever is more convenient for you.

USB connections are preferred - If `ConnectionType` is equal to `LJM_ctANY` (0), `LJM_Open` will attempt to open a USB connection before trying to open a TCP connection.

Device connections are claimed - Once a LabJack device is opened on a given connection type, other processes will generally not be able to open that same device using the same connection type until the device connection is closed by using `LJM_Close` or `LJM_CloseAll`.

Subsequent calls may return the same handle - Once a device is opened, subsequent calls to `LJM_Open` / `LJM_OpenS` will return the handle to that device if the `DeviceType`, `ConnectionType`, and `Identifier` parameters describe that device. In this case, calling `LJM_Open` or `LJM_OpenS` is nearly instant because no device communication occurs. For example:

```
LJM_Open(LJM_dtANY, LJM_ctANY, LJM_idANY, ...)
// If this returns a handle to a T7, with serial number 470010729, connected via USB...
```

```
LJM_Open(LJM_dtT7, LJM_ctUSB, "470010729", ...)
// ...then this would return the same handle, without opening a new device connection
```

```
// ...and these would attempt to open a new device connection and return a different handle:
LJM_Open(LJM_dtANY, LJM_ctANY, "470010999", ...) // Different serial number
LJM_Open(LJM_dtANY, LJM_ctETHERNET, "470010729", ...) // Different connection type
```

Devices like the T7 may have multiple device connections at once. The following calls would each return a different device handle:

- `LJM_Open(LJM_dtT7, LJM_ctUSB, "470010729", ...)`
- `LJM_Open(LJM_dtT7, LJM_ctETHERNET, "470010729", ...)`
- `LJM_Open(LJM_dtT7, LJM_ctWIFI, "470010729", ...)`

Reconnection is automatic - If LJM detects a broken connection, it will automatically attempt to reconnect. If the reconnect is unsuccessful, LJM will return the errorcode `LJME_RECONNECT_FAILED`. When this happens, many applications can simply retry calling the function that returned `LJME_RECONNECT_FAILED` with the same parameters. For more details, see the [reconnection](#) page.

Using an IP Address as the Identifier parameter is efficient - In order to open a TCP device with a specific serial number or name, LJM will communicate with all LabJack devices on the network, asking each for their serial number or name. In contrast, opening a TCP LabJack device by IP address will only send packets to that IP address, which is faster and more efficient. Setting up static IP addresses can be troublesome, so use what is appropriate for you application.

Specific IPs will be checked - [Specific IP addresses](#) may be defined, which LJM will try to open during every Ethernet- or WiFi-based Open or [ListAll](#) call.

Closing

Closing a handle will remove the associated device connection from the LJM library. Closing will both free the device to be opened again, and free allocated system resources.

- `LJM_Close` - Close a [specific](#) device connection, based on the handle number.
- `LJM_CloseAll` - Close [any/all](#) open device connections.

Devices are closed automatically when LJM is unloaded - When the LJM library (.dll, .dylib, or .so) is unloaded, it will automatically close all devices.

Device Not Found?

If a device isn't properly being found, some important debugging information can be found in our [Device Not Found?](#) App Note. If a network connection needs to be debugged, read through our [Setup WiFi and Ethernet](#) App Note as well.

2.1.1 - Open

Opens a desired LabJack and associates a device handle number (connection ID). The device handle may then be passed as an input to other functions.

Syntax

```
LJM_ERROR_RETURN LJM_Open(
    int DeviceType,
    int ConnectionType,
    const char * Identifier,
    int * Handle)
```

Parameters

DeviceType [in]

An integer containing the type of the device to be connected:

`LJM_dtANY` (0) - Open any supported LabJack device type

`LJM_dtT4` (4) - Open a T4 device

`LJM_dtT7` (7) - Open a T7-series device

`LJM_dtTSERIES` (84) - Open a T4 or T7

`LJM_dtDIGIT` (200) - Open a Digit-series device

For other LabJack devices, see [What driver/library should I use with my LabJack?](#)

ConnectionType [in]

An integer containing the type of connection desired:

`LJM_ctANY` (0), `LJM_ctUSB` (1), `LJM_ctTCP` (2), `LJM_ctETHERNET` (3), `LJM_ctWIFI` (4)

Other UDP options:

`LJM_ctNETWORK_UDP` (5), `LJM_ctETHERNET_UDP` (6), `LJM_ctWIFI_UDP` (7)

Other TCP or UDP options:

`LJM_ctNETWORK_ANY` (8), `LJM_ctETHERNET_ANY` (9), `LJM_ctWIFI_ANY` (10)

Identifier [in]

A string that may identify the device to be connected. To open any device, use LJM_idANY (or the string "ANY"). To specify an identifier, use a serial number, IP address, or device name. See [Identifier Parameter](#) for more information. If you don't have a device available, you can use [LJM_DEMO_MODE](#) (or the string "-2") to open a fake device.

Handle [out]

The new handle that represents the device connection.

Returns

[LJM_errorcode](#) or 0 for no error.

Related Functions

LJM_Open and [LJM_OpenS](#) are essentially the same, except that [LJM_OpenS](#) uses string parameters for DeviceType and ConnectionType rather than integer parameters.

See [LJM_GetHandleInfo](#) to retrieve information about a handle.

See [LJM_ListAll](#) or [LJM_ListAllS](#) to find multiple devices.

Remarks

See the notes in [Opening and Closing](#).

When the ConnectionType parameter of this function is network-based, this function will check the IP addresses listed in [LJM_SPECIAL_ADDRESSES_FILE](#).

Examples

[C/C++] Opening the first found T7, using LJM_Open

```
int LJMErr;
int handle;
LJMErr = LJM_Open(7, 0, "ANY", &handle);
```

2.1.2 - OpenS

Opens a desired LabJack and associates a device handle number (connection ID). The device handle may then be passed as an input to other functions.

Syntax

```
LJM_ERROR_RETURN LJM_OpenS(
    const char * DeviceType,
    const char * ConnectionType,
    const char * Identifier,
    int * Handle)
```

Parameters**DeviceType** [in]

A string containing the type of the device to be connected:

"ANY" - Open any supported LabJack device type

"T4" - Open a T4 device

"T7" - Open a T7-series device

"TSERIES" - Open a T4 or T7

"DIGIT" - Open a Digit-series device

For other LabJack devices, see [What driver/library should I use with my LabJack?](#)

ConnectionType [in]

A string containing the type of connection desired (USB or TCP):

"ANY", "USB", "TCP", "ETHERNET", or "WIFI"

Additional UDP options:

"NETWORK_UDP", "ETHERNET_UDP", "WIFI_UDP"

Additional TCP or UDP options:

"NETWORK_ANY", "ETHERNET_ANY", "WIFI_ANY"

Identifier [in]

A string that may identify the device to be connected. To open any device, use LJM_idANY (or the string "ANY"). To specify an identifier, use a serial number, IP address, or device name. See [Identifier Parameter](#) for more information. If you don't have a device available, you can use [LJM_DEMO_MODE](#) (or the string "-2") to open a fake device.

Handle [out]

The new handle that represents the device connection.

Returns

[LJM_errorcode](#) or 0 for no error.

Relevant Functions

[LJM_OpenS](#) and [LJM_Open](#) are essentially the same, except that [LJM_Open](#) uses integer parameters for DeviceType and ConnectionType rather than string parameters.

See [LJM_GetHandleInfo](#) to retrieve information about a handle.

See [LJM_ListAll](#) or [LJM_ListAllS](#) to find multiple devices.

Remarks

See the notes in [Opening and Closing](#).

When the ConnectionType parameter of this function is network-based, this function will check the IP addresses listed in [LJM_SPECIAL_ADDRESSES_FILE](#).

Examples

[C#] Opening a T7 via TCP using LJM_OpenS

```
int LJLError;
int handle;
LJLError = LabJack.LJM.OpenS("T7", "Ethernet", "ANY", ref handle);
```

[C/C++] Opening a T7 via TCP using LJM_OpenS

```
int LJLError;
int handle;
LJLError = LJM_OpenS("T7", "TCP", "ANY", &handle);
```

2.1.2.1 - Identifier Parameter

This page describes the Identifier parameter that is used in [LJM_Open](#) and [LJM_OpenS](#).

Common Usage

- **To Open Any Device (Ignore Identifier):**
 - Use the constant `LJM_idANY` or the string "ANY".
- **To Open By Serial Number:**
 - Specify the serial number with numbers only. For example: "470010103".
- **To Open By IP Address:**
 - Specify the decimal-dot IP address numbers and periods only. For example: "192.168.1.207".
 - To open by hexadecimal IP address, see **'Hex vs. Decimal IP Address'** below.
- **To Open By Device Name:**
 - Specify the device name with letters and numbers only. For example: "My Number 1 Favorite DAQ Device". Device names may be up to 49 characters in length, not including the null-terminator character, and must contain at least one letter (to prevent ambiguity between device name and serial number).

Network-Specific

Specifying the Port

TCP/UDP port can be specified by appending a colon and the port number. For example, if your Identifier is the IP address 192.168.1.42 and you want to connect to port 502, your Identifier would be "192.168.1.42:502".

Most users should not need to specify the TCP/UDP port.

Hex vs. Decimal IP Address

LJM assumes IP addresses to be decimal, unless they specifically look hexadecimal. To use a hex IP address, append "0x" to the IP address and write it with a dot between each byte.

For example, for the hexadecimal IP address C0A8012A can be passed as Identifier as "0xC0.A8.01.2A". (An Identifier like "C0.A8.01.2A" would be interpreted as a hex IP address because it contains letters, but it is highly recommended to use the "0x" prefix anyway.)

Any port specified will still be interpreted as a decimal number.

USB-Specific

Specifying the Pipe

For USB, pipe can be specified by appending a colon and the pipe number. For example, if your Identifier is the serial number 470010117 and you want to connect to pipe 0, your Identifier would be "470010117:0".

Most users should not need to specify the USB pipe.

No Device

If you don't have a device available, you can use LJM_DEMO_MODE (or the string "-2") to open a fake device. For more details, see [Demo Mode](#).

2.1.3 - Close

[Add new comment](#)

Closing a handle will remove the associated device connection from the LJM library, free the device to be opened again, and free allocated system resources.

Syntax

```
LJM_ERROR_RETURN LJM_Close(int Handle)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#), or [LJM_OpenS](#).

Returns

[LJM_errorcode](#) or 0 for no error.

Remarks

LJM_Close is useful when multiple LabJack device connections are open and only a single connection needs to be closed. Use [LJM_CloseAll](#) to close every device in a single function call.

Examples

[C#] Closing one device

```
int LJMErr;
//handle from LJM_Open()
LJMErr = LabJack.LJM.Close(handle);
```

[C/C++] Closing one device

```
int LJMErr;
//handle from LJM_Open()
LJMErr = LJM_Close(handle);
```

2.1.4 - CloseAll

Closing all devices will remove all device handles from the LJM library, free all previously open devices to be opened again, and free allocated system resources.

Syntax

```
LJM_ERROR_RETURN LJM_CloseAll()
```

Parameters

None

Returns

[LJM_errorcode](#) or 0 for no error.

Remarks

LJM_CloseAll is useful on program exit. Use [LJM_Close](#) to close an individual device handle.

Examples

[C/C++] Closing all open devices

```
int LJMErr;
LJMErr = LJM_CloseAll();
```

2.2 - Single Value Functions

Summary

To access device data one value at a time, use the following functions. See the [Modbus map](#) to see what values can be accessed.

Name Functions

Name functions access data by a name, such as "AIN5" for analog input 5.

- [LJM_eWriteName](#) - Write one value specified, by name.
- [LJM_eReadName](#) - Read one value specified, by name.
- [LJM_eWriteNameString](#) - Write one string value, specified by name.
- [LJM_eReadNameString](#) - Read one string value, specified by name.

Address Functions

Address functions access data by an address and data type. For example, analog input 5 ("AIN5") would have address 10 and a 32-bit floating point type.

- [LJM_eWriteAddress](#) - Write one value, specified by address/type.
- [LJM_eReadAddress](#) - Read one value, specified by address/type.
- [LJM_eWriteAddressString](#) - Write one string value, specified by address.
- [LJM_eReadAddressString](#) - Read one string value, specified by address.

Need to read or write multiple values at once? Use the [Multiple Value Functions](#).

2.2.1 - eWriteName

[Add new comment](#)

Write one value, specified by name.

Syntax

```
LJM_ERROR_RETURN LJM_eWriteName(
    int Handle,
    const char * Name,
    double Value)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Name [in]

The name that specifies the Modbus register(s) to write. Names can be found throughout the device datasheet or in the [Modbus Map](#).

Value [in]

The value to send to the device.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

For an alternate function using an address rather than name, see [LJM_eWriteAddress](#). More code examples coming soon.

Example

[C/C++] Write a value of 2.5V to the DAC0 analog output

```
int LJLError;

// handle comes from LJM_Open()
LJLError = LJM_eWriteName(handle, "DAC0", 2.5);
if (LJLError != LJME_NOERROR) {
    // Deal with error
}
```

2.2.2 - eReadName

Read one value, specified by name.

Syntax

```
LJM_ERROR_RETURN LJM_eReadName(
    int Handle,
    const char * Name,
    double * Value)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Name [in]

The name that specifies the Modbus register(s) to read. Names can be found throughout the device datasheet or in the [Modbus Map](#).

Value [out]

The value returned from the device.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

For an alternate function using an address rather than name, see [LJM_eReadAddress](#). More code examples coming soon.

Example

[C/C++] Read the serial number of the device.

```
int LJMErr;
double newValue;

// handle comes from LJM_Open()
LJMErr = LJM_eReadName(handle, "SERIAL_NUMBER", &newValue);
if (LJMErr != LJME_NOERROR) {
    // Deal with error
}
printf("SERIAL_NUMBER: %f\n", newValue);
```

2.2.3 - eWriteAddress

Write one value, specified by address.

Syntax

```
LJM_ERROR_RETURN LJM_eWriteAddress(
    int Handle,
    int Address,
    int Type,
    double Value)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Address [in]

The address that specifies the Modbus register(s) to write. Addresses can be found throughout the device datasheet or in the [Modbus Map](#).

Type [in]

The data type of the value:

Type	LJM Constant Name	LJM Constant Value
unsigned 16-bit integer	LJM_UINT16	0
unsigned 32-bit integer	LJM_UINT32	1
signed 32-bit integer	LJM_INT32	2
floating point 32-bit	LJM_FLOAT32	3

Value [in]

The value to send to the device. The input data type is a double, and will be converted according to the Type input.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

For an alternate function using a name rather than address, see [LJM_eWriteName](#).

Examples

[C/C++] Write a value of 2.5V to the DAC0 analog output.

```
int LJMErr;

// handle comes from LJM_Open()
LJMErr = LJM_eWriteAddress(handle, 1000, 3, 2.5);
if (LJMErr != LJME_NOERROR) {
    // Deal with error
}
```

2.2.4 - eReadAddress

Read one value, specified by address.

Syntax

```
LJM_ERROR_RETURN LJM_eReadAddress(
    int Handle,
    int Address,
    int Type,
    double * Value)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Address [in]

The address that specifies the Modbus register(s) to read. Addresses can be found throughout the device datasheet or in the [Modbus Map](#).

Type [in]

The data type of the value:

Type	LJM Constant Name	LJM Constant Value
unsigned 16-bit integer	LJM_UINT16	0
unsigned 32-bit integer	LJM_UINT32	1
signed 32-bit integer	LJM_INT32	2
floating point 32-bit	LJM_FLOAT32	3

Value [out]

The value returned from the device. The output data type is a double, and will be converted according to the Type input.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

For an alternate function using a name rather than address, see [LJM_eReadName](#). More code examples coming soon.

Examples

[C/C++] Read the serial number of the device.

```
int LJMErr;
double newValue;

// handle comes from LJM_Open()
LJMErr = LJM_eReadAddress(handle, 60028, 1, &newValue);
if (LJMErr != LJME_NOERROR) {
    // Deal with error
}
printf("SERIAL_NUMBER: %f\n", newValue);
```

2.2.5 - eWriteNameString

Writes a string, specified by name.

Syntax

```
LJM_ERROR_RETURN LJM_eWriteNameString(
    int Handle,
    const char * Name,
    const char * String)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Name [in]

The name that specifies the Modbus string-type register to write. Names can be found throughout the device datasheet or in the [Modbus Map](#).

String [in]

The string to write. Must null-terminate at size LJM_STRING_ALLOCATION_SIZE (50) or less.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

See also [LJM_eReadNameString](#). This is a convenience function that uses [LJM_eNames](#). Only for use with Modbus register(s) listed as type LJM_STRING (98). More code examples coming soon.

Examples

[C/C++] Change the device name.

```
int LJMErr;
// LJM_STRING_ALLOCATION_SIZE is 50
char newName[LJM_STRING_ALLOCATION_SIZE] = "My Favorite DAQ Device";

// handle comes from LJM_Open()
LJMErr = LJM_eWriteNameString(handle, "DEVICE_NAME_DEFAULT", newName);
if (LJMErr != LJME_NOERROR) {
    // Deal with error
}
```

2.2.6 - eReadNameString

Reads a string, specified by name.

Syntax

```
LJM_ERROR_RETURN LJM_eReadNameString(
    int Handle,
    const char * Name,
    char * String)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Name [in]

The name that specifies the Modbus string-type register to read. Names can be found throughout the device datasheet or in the [Modbus Map](#).

String [out]

A string that is updated to contain the result of the read. Must be allocated to size LJM_STRING_ALLOCATION_SIZE (50) or greater prior to calling this function.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

See also [LJM_eWriteNameString](#). This is a convenience function that uses [LJM_eNames](#). Only for use with Modbus register(s) listed as type LJM_STRING (98).

Examples

[C/C++] Read the device name.

```
int LJMError;
// LJM_STRING_ALLOCATION_SIZE is 50
char devName[LJM_STRING_ALLOCATION_SIZE];

// handle comes from LJM_Open()
LJMError = LJM_eReadNameString(handle, "DEVICE_NAME_DEFAULT", devName);
if (LJMError != LJME_NOERROR) {
    // Deal with error
}
printf ("%s \n", devName);
```

2.2.7 - eWriteAddressString

Writes a string, specified by address.

Syntax

```
LJM_ERROR_RETURN LJM_eWriteAddressString(
    int Handle,
    int Address,
    const char * String)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Address [in]

The address that specifies the Modbus string-type register to write. Addresses can be found throughout the device datasheet or in the [Modbus Map](#).

String [in]

The string to write. Must null-terminate at size LJM_STRING_ALLOCATION_SIZE (50) or less.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

See also [LJM_eReadAddressString](#). This is a convenience function that uses [LJM_eAddresses](#). Only for use with Modbus registers listed as type LJM_STRING (98). More code examples coming soon.

Examples

[C/C++] Change the device name.

```
int LJMError;
// LJM_STRING_ALLOCATION_SIZE is 50
char newName[LJM_STRING_ALLOCATION_SIZE] = "My Favorite DAQ Device";

// handle comes from LJM_Open()
LJMError = LJM_eWriteAddressString(handle, 60500, newName);
if (LJMError != LJME_NOERROR) {
    // Deal with error
}
```

2.2.8 - eReadAddressString

Reads a string, specified by address.

Syntax

```
LJM_ERROR_RETURN LJM_eReadAddressString(
    int Handle,
    int Address,
    char * String)
```

Parameters

Handle [in] A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Name [in]

The address that specifies the Modbus string-type register to read. Addresses can be found throughout the device datasheet or in the [Modbus Map](#).

String [out]

A string that is updated to contain the result of the read. Must be allocated to size LJM_STRING_ALLOCATION_SIZE (50) or greater prior to calling this function.

Returns

LJM errorcodes or 0 for no error.

Remarks

See also [LJM_eWriteAddressString](#). This is a convenience function that uses [LJM_eAddresses](#). Only for use with Modbus registers listed as type LJM_STRING (98).

Examples

[C/C++] Read the device name.

```
int LJMErr;
// LJM_STRING_ALLOCATION_SIZE is 50
char devName[LJM_STRING_ALLOCATION_SIZE];

// handle comes from LJM_Open()
LJMErr = LJM_eReadAddressString(handle, 60500, devName);
if (LJMErr != LJME_NOERROR) {
    // Deal with error
}
printf ("%s \n", devName);
```

2.3 - Multiple Value Functions

Summary

To access device data multiple values at once, use the following functions. See the [Modbus map](#) to see what values can be accessed.

Name Functions

Name functions access data through a name, such as "AIN5" for analog input 5.

- [LJM_eWriteNames](#) - Write one value each to multiple names.
- [LJM_eReadNames](#) - Read one value each from multiple names.
- [LJM_eWriteNameArray](#) - Write consecutive values, specified by name.
- [LJM_eReadNameArray](#) - Read consecutive values, specified by name.
- [LJM_eWriteNameByteArray](#) - Write consecutive bytes, specified by name.
- [LJM_eReadNameByteArray](#) - Read consecutive bytes, specified by name.
- [LJM_eNames](#) - Write/read values, specified by names and array sizes.

Address Functions

Address functions access data through an address, such as 10 for analog input 5.

- [LJM_eWriteAddresses](#) - Write one value each to multiple addresses.
- [LJM_eReadAddresses](#) - Read one value each from multiple addresses.
- [LJM_eWriteAddressArray](#) - Write consecutive values specified, by address.
- [LJM_eReadAddressArray](#) - Read consecutive values specified, by address.
- [LJM_eWriteAddressByteArray](#) - Write consecutive bytes, specified by address.
- [LJM_eReadAddressByteArray](#) - Read consecutive bytes, specified by address.
- [LJM_eAddresses](#) - Write/Read values, specified by addresses and array sizes.

LJM automatically splits operations into multiple packets as needed

If an LJM multiple value function is passed a large enough number of read and/or write operations, it will split the reads and writes into separate packets. By default, LJM uses the custom Modbus Feedback command, which can perform both reads and writes.

If a multiple value function is split into multiple packets, it can affect the timing of samples and/or commands. It can also increase the latency of operations. For approximate data rates, see [T-series Data Rates](#).

Maximum packet sizes per connection type:

- [T-series USB](#)
- [T-series Ethernet](#)
- [T-series WiFi](#)

Checking how many packets are required for a particular set of operations

You can use the [LJM debug log](#) with LJM_DEBUG_LOG_LEVEL set to LJM_PACKET or lower to check how LJM sends operations to the device.

To manually calculate how many packets will be sent:

- Modbus Feedback takes 8 packet header bytes—for the command (to the device) as well as for the response (from the device).
- Each Modbus Feedback command packet contains one or more frames:
 - Each command Feedback frame has 4 header bytes.
 - Within each command Feedback frame, each address being written has an appropriate number data bytes.
 - (Each 16-bit value being written consists of 2 data bytes; each 32-bit value being written consists of 4 bytes; etc.)
- Each response has an appropriate number of data bytes.
 - (Each 16-bit address that was read consists of 2 data bytes; each 32-bit value that was read consists of 4 data bytes; etc.)
- More information:
 - [Modbus Feedback protocol](#)

LJM compresses reads/writes of consecutive addresses

In addition, LJM will automatically "compress" multiple consecutive reads or writes. For example, if you read analog inputs 0 through 13 (in the ordering of AIN0, AIN1, ..., AIN13) and a write to DAC0, LJM will compress the reads into one Modbus Feedback read frame, which takes 4 frame header bytes. For a USB connection to T4s and T7s—which has a 64-byte maximum packet size—this set of operations would take one packet:

- Command packet:
 - 8 header bytes
 - 4-byte frame header for reading 14 32-bit values starting at AIN0's address (AIN0 through AIN13)
 - 4-byte frame header for writing one 32-bit value at DAC0's address
 - 4-bytes of data for the value being written to DAC0
- Response packet:
 - 8 header bytes
 - 56 bytes of data for the values read from AIN0 through AIN13

Without compression and a 64-byte maximum packet size, the above operations would take multiple packets—the 14 frames for reading AIN0 through AIN13 would need 56 bytes. After the header's 8 bytes, there would be no room left for the DAC0 bytes.

Large responses can cause multiple packets

If the response is larger than the maximum packet size, it causes the command to be split. For example, reading 15 contiguous analog inputs from a T4 via USB requires two command packets, each with a response:

- First command/response:
 - Command packet:
 - 8 header bytes
 - 4-byte frame header for reading 14 32-bit values starting at AIN0's address (AIN0 through AIN13)
 - Response packet - 64 bytes total:
 - 8 header bytes
 - 56 bytes of data for the values read from AIN0 through AIN13
- Second command/response:
 - Command packet:
 - 8 header bytes
 - 4-byte frame header for reading one 32-bit value at AIN14's address
 - Response packet:
 - 8 header bytes
 - 4 bytes of data for the value read from AIN14

LJM sends one command at a time

Each commands-response is synchronous with the next, so LJM sends one command packet and waits for the response before sending another command packet.

LJM follows the specified order of reads/writes

LJM will not reorder operations to minimize the number of packets sent. If AIN1 is read then AIN0 is read, LJM will not compress them into a single command frame because that would cause AIN0 to be read before AIN1.

2.3.1 - eWriteNames

[Add new comment](#)

Write multiple values, specified by name.

Syntax

```
LJM_ERROR_RETURN LJM_eWriteNames(
    int Handle,
    int NumFrames,
    const char ** aNames,
    const double * aValues,
    int * ErrorAddress)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

NumFrames [in]

The total number of frames to access. A frame consists of one value, so the number of frames is the size of the aNames array.

aNames [in]

An array of names that specify the Modbus register(s) to write. Names can be found throughout the device datasheet or in the [Modbus Map](#).

aValues [in]

An array of values to send to the device. The array size should be the same as the aNames array. The input data type of each value is a double, and will be converted into the correct data type automatically.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

This function is commonly used to write a handful of values at once, and is more convenient than [LJM_eWriteAddresses](#) because it is not necessary to know the data type. More code examples coming soon.

Examples

[C/C++] Change digital I/O 0, 1, and 2 to output high, and digital I/O 6 to output low.

```
int LJMErr;
int errorAddress;
const char * aNames[4] = {"FIO0", "FIO1", "FIO2", "FIO6"};
double aValues[4] = {1, 1, 1, 0};

// handle comes from LJM_Open()
LJMErr = LJM_eWriteNames(handle, 4, aNames, aValues, &errorAddress);
if (LJMErr != LJME_NOERROR) {
    // Deal with error
}
```

2.3.2 - eReadNames

Read multiple values, specified by name.

Syntax

```
LJM_ERROR_RETURN LJM_eReadNames(
    int Handle,
    int NumFrames,
    const char ** aNames,
    double * aValues,
    int * ErrorAddress)
```

Parameters**Handle** [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

NumFrames [in]

The total number of frames to access. A frame consists of one value, so the number of frames is the size of the aNames array.

aNames [in]

An array of names that specify the Modbus register(s) to write. Names can be found throughout the device datasheet or in the [Modbus Map](#).

aValues [out]

An array of values received from the device. The array size should be same as the aNames array. The values will be converted into doubles automatically.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

This function is commonly used to read a handful of values at once, and is more convenient than [LJM_eReadAddresses](#) because it is not necessary to know the data type. More code examples coming soon.

Examples

[C/C++] Reading analog inputs 5, 6, and 10.

```
int LJMErr;
int errorAddress;
const char * aNames[3] = {"AIN5", "AIN6", "AIN10"};
double ainValues[3];
double ain5;
double ain6;
double ain10;

// handle comes from LJM_Open()
LJMErr = LJM_eReadNames(handle, 3, aNames, ainValues, &errorAddress);
if (LJMErr != LJME_NOERROR) {
    // Deal with error
}
ain5 = ainValues[0];
ain6 = ainValues[1];
ain10 = ainValues[2];
```

2.3.3 - eWriteAddresses

Write multiple values, specified by address.

Syntax

```
LJM_ERROR_RETURN LJM_eWriteAddresses(
    int Handle,
    int NumFrames,
    const int * aAddresses,
    const int * aTypes,
    const double * aValues,
    int * ErrorAddress)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

NumFrames [in]

The total number of frames to access. A frame consists of one value, so the number of frames is the size of the *aAddresses* array.

aAddresses [in]

An array of addresses that specify the Modbus register(s) to write. Addresses can be found throughout the device datasheet or in the [Modbus Map](#).

aTypes [in]

An array containing the data type of each value:

Type	LJM Constant Name	LJM Constant Value
unsigned 16-bit integer	LJM_UINT16	0
unsigned 32-bit integer	LJM_UINT32	1
signed 32-bit integer	LJM_INT32	2
floating point 32-bit	LJM_FLOAT32	3

aValues [in]

An array of values to send to the device. The array size should be the size of the *aAddresses* array. The input data type of each value is a double, and they will be converted into the data type entered in *aTypes*.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

This function is used to write a handful of values at once, and is useful in programming languages that are not friendly towards the usage of strings. For programming languages that can use strings easily, see [LJM_eWriteNames](#). More code examples coming soon.

Examples

[C/C++] Change digital I/O 0, 1, and 2 to output high, and digital I/O 6 to output low.

```
int LJMErr;
int errorAddress;
int aAddresses[4] = {2000, 2001, 2002, 2006};
int aTypes[4] = {0, 0, 0, 0};
double aValues[4] = {1, 1, 1, 0};

// handle comes from LJM_Open()
LJMErr = LJM_eWriteAddresses(handle, 4, aAddresses, aTypes, aValues, &errorAddress);
if (LJMErr != LJME_NOERROR) {
    // Deal with error
}
```

2.3.4 - eReadAddresses

Read multiple values, specified by address.

Syntax

```
LJM_ERROR_RETURN LJM_eReadAddresses(
    int Handle,
    int NumFrames,
    const int * aAddresses,
    const int * aTypes,
    double * aValues,
    int * ErrorAddress)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

NumFrames [in]

The total number of frames to access. A frame consists of one value, so the number of frames is the size of the *aAddresses* array.

aAddresses [in]

An array of addresses that specify the Modbus register(s) to read. Addresses can be found throughout the device datasheet or in the [Modbus Map](#).

aTypes [in]

An array containing the data type of each value:

Type	LJM Constant Name	LJM Constant Value
unsigned 16-bit integer	LJM_UINT16	0
unsigned 32-bit integer	LJM_UINT32	1
signed 32-bit integer	LJM_INT32	2
floating point 32-bit	LJM_FLOAT32	3

aValues [out]

An array of values received from the device. The array size should be the size of the *aAddresses* array. The output data type of each value is a double, and they will be converted from the data type entered in *aTypes*.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

This function is used to read a handful of values at once, and is useful in programming languages that are not friendly towards the usage of strings. For programming languages that can use strings easily, see [LJM_eReadNames](#). More code examples coming soon.

Examples

[C/C++] Reading analog inputs 5, 6, and 10.

```
int LJMErr;
int errorAddress;
int aAddresses[3] = {10, 12, 20};
int aTypes[3] = {3, 3, 3};
double ainValues[3];
double ain5;
double ain6;
```

```
double ain10;
```

```
// handle comes from LJM_Open()
LJMError = LJM_eReadAddresses(handle, 3, aAddresses, aTypes, ainValues, &errorAddress);
if (LJMError != LJME_NOERROR) {
    // Deal with error
}
ain5 = ainValues[0];
ain6 = ainValues[1];
ain10 = ainValues[2];
```

2.3.5 - eNames

Write/Read multiple values, specified by name. This function is designed to condense communication into arrays. Moreover, consecutive values can be accessed by specifying a starting name, and a number of values.

Syntax

```
LJM_ERROR_RETURN LJM_eNames(
    int Handle,
    int NumFrames,
    const char ** aNames,
    const int * aWrites,
    const int * aNumValues,
    double * aValues,
    int * ErrorAddress)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

NumFrames [in]

The total number of frames to access. A frame consists of one or more values, based on NumValues.

aNames [in]

An array of names that specify the Modbus register(s) to write/read. To access a consecutive group of values, populate an element of the aNames array with the starting name, and then increase the NumValues parameter according to the group size.

aWrites [in]

An array containing the desired type of access, which is either read(0) or write(1). The array size should be the same as the aNames array.

aNumValues [in]

An array that contains the per-name number of consecutive values. NumValues is 1 for non-consecutive. The array size should be the same as the aNames array.

aValues [in/out]

An array of values transferred to/from the device. The array size should be the sum of all elements in the aNumValues array. For values in the array that are being sent, data is automatically converted into the correct data type. For incoming values, data is converted to a double automatically.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

LJM_eNames is for programs that need to write and read multiple, arbitrary values in a single function call. For example, a PID loop may read inputs and set outputs in a single function call. In this way, the communication overhead is reduced. There are simpler [single value functions](#) and simpler [multiple value functions](#) for programs that do not need such a level of communication optimization.

To use LJM_eNames:

1. List of values that need to be accessed.
2. Decide how each value will be accessed: read or write.
3. Note values which are consecutive and have the same access (read or write). Only use the first name in a consecutive group and increase the NumValues parameter according to the group size.
4. Define NumFrames by counting all the values to access (step 1), then subtract consecutive values already accounted for in NumValues (step 3).
5. Insert data to be written to the device at the appropriate index of aValues.
6. Read data from aValues array after function is executed.

As an example of the above use-case, consider reading AIN0-2, setting DAC0 to 4.6V, and reading the state of DIO4.

1. Values that need to be accessed: [AIN0, AIN1, AIN2, DAC0, DIO4]
2. Reading the first 3 values, writing to the 4th value, and reading from the 5th [R, R, R, W, R]
3. Seeing that the first 3 values are consecutive and are all being read, the situation can be simplified. The array size of aNames, aTypes, and aWrites is

reduced.

1. aNames = [AIN0, DAC0, DIO4]
2. aWrites = [LJM_READ, LJM_WRITE, LJM_READ] - LJM_READ is 0; LJM_WRITE is 1.
3. aNumValues = [3, 1, 1] - The number of values is increased for the first frame only.
4. The number of frames is 3
5. Set the analog output voltage to 4.6V by setting the fourth value. aValues = [0, 0, 0, 4.6, 0]
6. Read the first three values of aValues to get AIN0 through AIN2; read the fifth value to get FIO4

Examples

[C/C++] Read analog inputs 0 through 2, set DAC0 to 4.6V, and read FIO4

```
int LJMErr;
int errorAddress;
const char * aNames[3] = {"AIN0", "DAC0", "FIO4"};
int aWrites[3] = {LJM_READ, LJM_WRITE, LJM_READ};
int aNumValues[3] = {3, 1, 1};
double aValues[5];
aValues[3] = 4.6;

// handle comes from LJM_Open()
LJMErr = LJM_eNames(handle, 3, aNames, aWrites, aNumValues, aValues, &errorAddress);
if (LJMErr != LJME_NOERROR) {
    // Deal with error
}
printf("AIN0: %f\n", aValues[0]);
printf("AIN1: %f\n", aValues[1]);
printf("AIN2: %f\n", aValues[2]);
printf("FIO4: %f\n", aValues[4]);
```

2.3.6 - eAddresses

Write/Read multiple values, specified by address. This function is designed to condense communication into arrays. Moreover, consecutive values can be accessed by specifying a starting address, and a number of values.

Syntax

```
LJM_ERROR_RETURN LJM_eAddresses(
    int Handle,
    int NumFrames,
    const int * aAddresses,
    const int * aTypes,
    const int * aWrites,
    const int * aNumValues,
    double * aValues,
    int * ErrorAddress)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

NumFrames [in]

The total number of frames to access. A frame consists of one or more values, based on NumValues.

aAddresses [in]

An array of addresses that specify the Modbus register(s) to write/read. To access a consecutive group of values, populate an element of aAddresses array with the starting address, and then increase the NumValues parameter according to the group size.

aTypes [in]

An array of size NumFrames, containing the data type of each value sent/received:

Type	LJM Constant Name	LJM Constant Value
unsigned 16-bit integer	LJM_UINT16	0
unsigned 32-bit integer	LJM_UINT32	1
signed 32-bit integer	LJM_INT32	2
floating point 32-bit	LJM_FLOAT32	3

aWrites [in]

An array containing the desired type of access, which is either read(0) or write(1). The array size should be the same as the aAddresses array.

aNumValues [in]

An array that contains the per-address number of consecutive values. NumValues is 1 for non-consecutive. The array size should be the same as the aAddresses array.

aValues [in/out]

An array of values to be transferred to/from the device. The array size should be the sum of all elements in the aNumValues array. Each value will be converted to/from those defined in aTypes.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

LJM_eAddresses is equivalent to [LJM_eNames](#).

LJM_eAddresses is for programs that need to write and read multiple, arbitrary values in a single function call. For example, a PID loop may read inputs and set outputs in a single function call. In this way, the communication overhead is reduced. There are simpler [single value functions](#) and simpler [multiple value functions](#) for programs that do not need such a level of communication optimization.

To use LJM_eAddresses:

1. Create a list of values that need to be accessed.
2. Decide how each value will be accessed: read or write.
3. Note values which are consecutive and have the same access (read or write). Only use the first name in a consecutive group and increase the NumValues parameter according to the group size.
4. Define NumFrames by counting all the values to access (step 1), then subtract consecutive values already accounted for in NumValues (step 3).
5. Insert data to be written to the device at the appropriate index of aValues.
6. Read data from aValues array after function is executed.

As an example of the above use-case, consider reading AIN0-2 (addresses 0, 2, 4), setting DAC0 (address 1000) to 4.6V, and reading the state of DIO4 (address 2004).

1. Addresses that need to be accessed: [0, 2, 4, 1000, 2004]
2. Reading the first 3 values, writing to the 4th value, and reading from the 5th [R, R, R, W, R]
3. Seeing that the first 3 values are consecutive and are all being read, the situation can be simplified. The array size of aNames, aTypes, and aWrites is reduced.
 1. aAddresses = [0, 1000, 2004]
 2. aTypes = [LJM_FLOAT32, LJM_FLOAT32, LJM_UINT16] - LJM_FLOAT32 is 3; LJM_UINT16 is 0.
 3. aWrites = [LJM_READ, LJM_WRITE, LJM_READ] - LJM_READ is 0; LJM_WRITE is 1.
 4. aNumValues = [3, 1, 1] - The number of values is increased for the first frame only.
4. The number of frames is 3
5. Set the analog output voltage to 4.6V by setting the fourth value. aValues = [0, 0, 0, 4.6, 0]
6. Read the first three values of aValues to get AIN0 through AIN2; read the fifth value to get FIO4

Examples

[C/C++] Read analog inputs 0 through 7, set DAC0 to 4.6V, read FIO4

```
int LJMEError;
int errorAddress;
int aAddresses[3] = {0, 1000, 2004};
int aTypes[3] = {LJM_FLOAT32, LJM_FLOAT32, LJM_UINT16};
int aWrites[3] = {LJM_READ, LJM_WRITE, LJM_READ};
int aNumValues[3] = {3, 1, 1};
double aValues[5];
aValues[3] = 4.6;

// handle comes from LJM_Open()
LJMEError = LJM_eAddresses(handle, 3, aAddresses, aTypes, aWrites, aNumValues, aValues, &errorAddress);
if (LJMEError != LJME_NOERROR) {
    // Deal with error
}
printf("AIN0: %f\n", aValues[0]);
printf("AIN1: %f\n", aValues[1]);
printf("AIN2: %f\n", aValues[2]);
printf("FIO4: %f\n", aValues[4]);
```

2.3.7 - eReadAddressArray

[Add new comment](#)

Read consecutive values, specified by an address and type.

Syntax

```
LJM_ERROR_RETURN LJM_eReadAddressArray(
    int Handle,
    int Address,
```

```
int Type,
int NumValues,
double * aValues,
int * ErrorAddress)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Address [in]

The address that specifies the Modbus register(s) to read. Addresses can be found throughout the device datasheet or in the [Modbus Map](#).

Type [in]

The data type of the value(s):

Type	LJM Constant Name	LJM Constant Value
unsigned 16-bit integer	LJM_UINT16	0
unsigned 32-bit integer	LJM_UINT32	1
signed 32-bit integer	LJM_INT32	2
floating point 32-bit	LJM_FLOAT32	3

NumValues [in]

The number of consecutive values to read.

aValues [out]

An array of values to be transferred from the device. The array size should be equal to NumValues. Each value will be converted according to Type.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

The Name version of this function is [LJM_eReadNameArray](#).

If NumValues is large enough, these functions will automatically split reads into multiple packets based on the current device's effective data packet size. Using both non-buffer and [buffer](#) registers in one function call is not supported.

Examples

[C/C++] Read analog inputs 0 through 7

```
int LJMErr;
int errorAddress;
double newValues[8];

// handle comes from LJM_Open()
LJMErr = LJM_eReadAddressArray(handle, 0, LJM_FLOAT32, 8, newValues, &errorAddress);
if (LJMErr != LJME_NOERROR) {
    // Deal with error
}
```

2.3.8 - eReadAddressByteArray

Read consecutive byte values, specified by an address.

Syntax

```
LJM_ERROR_RETURN LJM_eReadAddressByteArray(
    int Handle,
    int Address,
    int NumBytes,
    char * aBytes,
    int * ErrorAddress)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Address [in]

The address that specifies the Modbus register(s) to read. Addresses can be found throughout the device datasheet or in the [Modbus Map](#).

NumBytes [in]

The number of consecutive bytes.

aBytes [out]

An array of bytes transferred from the device. The array size should be equal to NumBytes.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

The Name version of this function is [LJM_eReadNameByteArray](#).

This function will append a 00 byte to aBytes for odd-numbered NumBytes.

If NumBytes is large enough, these functions will automatically split reads into multiple packets based on the current device's effective data packet size. Using both non-buffer and [buffer](#) registers in one function call is not supported.

Examples

[C] Read Lua output from LUA_DEBUG_DATA

```
int bytelter, err;
double numBytes;
char * aBytes;
int errorAddress;
numBytes = 0;

// handle comes from LJM_Open()
err = LJM_eReadName(handle, "LUA_DEBUG_NUM_BYTES", &numBytes);
if (err != LJME_NOERROR) {
    // Deal with error
}
aBytes = malloc(sizeof(char) * (int)numBytes);
errorAddress = -2; // Something impossible for a Modbus address
err = LJM_eReadAddressByteArray(
    handle,
    6024, // LUA_DEBUG_DATA
    numBytes,
    aBytes,
    &errorAddress
);
if (err == LJME_NOERROR) {
    printf("LUA_DEBUG_DATA: ");
    for (bytelter = 0; bytelter < numBytes; bytelter++) {
        printf("%c", aBytes[bytelter]);
    }
    printf("\n");
}
free(aBytes);
if (err != LJME_NOERROR) {
    // Deal with error
}
```

2.3.9 - eReadNameArray

Read consecutive values, specified by a name.

Syntax

```
LJM_ERROR_RETURN LJM_eReadNameArray(
    int Handle,
    const char * Name,
    int NumValues,
    double * aValues,
    int * ErrorAddress)
```

Parameters**Handle** [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Name [in]

The name that specifies the Modbus register(s) to read. Names can be found throughout the device datasheet or in the [Modbus Map](#).

NumValues [in]

The number of consecutive values to read.

aValues [out]

An array of values to be transferred from the device. The array size should be equal to NumValues. Each value will be converted according to the type of Name.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

The Address version of this function is [LJM_eReadAddressArray](#).

If NumValues is large enough, these functions will automatically split reads into multiple packets based on the current device's effective data packet size. Using both non-buffer and [buffer](#) registers in one function call is not supported.

Examples

[C/C++] Read analog inputs 0 through 7

```
int LJMErr;
int errorAddress;
double newValues[8];

// handle comes from LJM_Open()
LJMErr = LJM_eReadNameArray(handle, "AIN0", 8, newValues, &errorAddress);
if (LJMErr != LJME_NOERROR) {
    // Deal with error
}
```

2.3.10 - eReadNameByteArray

Read consecutive byte values, specified by a name.

Syntax

```
LJM_ERROR_RETURN LJM_eReadNameByteArray(
    int Handle,
    const char * Name,
    int NumBytes,
    char * aBytes,
    int * ErrorAddress)
```

Parameters**Handle** [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Name [in]

The name that specifies the Modbus register(s) to read. Names can be found throughout the device datasheet or in the [Modbus Map](#).

NumBytes [in]

The number of consecutive bytes.

aBytes [out]

An array of bytes transferred from the device. The array size should be equal to NumBytes.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

[LJM_eWriteNameByteArray](#) is the reverse of [LJM_eReadNameByteArray](#).

The Address version of this function is [LJM_eReadAddressByteArray](#).

This function will append a 00 byte to aBytes for odd-numbered NumBytes.

If NumBytes is large enough, these functions will automatically split reads into multiple packets based on the current device's effective data packet size. Using

both non-buffer and [buffer](#) registers in one function call is not supported.

Examples

[C] Read Lua output from LUA_DEBUG_DATA

```
int bytelter, err;
double numBytes;
char * aBytes;
int errorAddress;
numBytes = 0;

// handle comes from LJM_Open()
err = LJM_eReadName(handle, "LUA_DEBUG_NUM_BYTES", &numBytes);
if (err != LJME_NOERROR) {
    // Deal with error
}
aBytes = malloc(sizeof(char) * (int)numBytes);
errorAddress = -2; // Something impossible for a Modbus address
err = LJM_eReadNameByteArray(
    handle,
    "LUA_DEBUG_DATA",
    numBytes,
    aBytes,
    &errorAddress
);
if (err == LJME_NOERROR) {
    printf("LUA_DEBUG_DATA: ");
    for (bytelter = 0; bytelter < numBytes; bytelter++) {
        printf("%c", aBytes[bytelter]);
    }
    printf("\n");
}
free(aBytes);
if (err != LJME_NOERROR) {
    // Deal with error
}
```

2.3.11 - eWriteAddressArray

Write consecutive values, specified by an address and type.

Syntax

```
LJM_ERROR_RETURN LJM_eWriteAddressArray(
    int Handle,
    int Address,
    int Type,
    int NumValues,
    double * aValues,
    int * ErrorAddress)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Address [in]

The address that specifies the Modbus register(s) to write. Addresses can be found throughout the device datasheet or in the [Modbus Map](#).

Type [in]

The data type of the value(s):

Type	LJM Constant Name	LJM Constant Value
unsigned 16-bit integer	LJM_UINT16	0
unsigned 32-bit integer	LJM_UINT32	1
signed 32-bit integer	LJM_INT32	2
floating point 32-bit	LJM_FLOAT32	3

NumValues [in]

The number of consecutive values.

aValues [in]

An array of values to be transferred to the device. The array size should be equal to NumValues. Each value will be converted according to Type.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

The Name version of this function is [LJM_eWriteNameArray](#).

If NumValues is large enough, these functions will automatically split writes into multiple packets based on the current device's effective data packet size. Using both non-buffer and [buffer](#) registers in one function call is not supported.

Examples

[C/C++] Write DAC0 and DAC1 as an array

```
int LJMEError;
int errorAddress;
double newValues[2] = {1.2, 3.4};

// handle comes from LJM_Open()
LJMEError = LJM_eWriteAddressArray(handle, 1000, LJM_FLOAT32, 2, newValues, &errorAddress);
if (LJMEError != LJME_NOERROR) {
    // Deal with error
}
```

2.3.12 - eWriteAddressByteArray

Write consecutive byte values, specified by an address.

Syntax

```
LJM_ERROR_RETURN LJM_eWriteAddressByteArray(
    int Handle,
    int Address,
    int NumBytes,
    const char * aBytes,
    int * ErrorAddress)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Address [in]

The address that specifies the Modbus register(s) to write. Address can be found throughout the device datasheet or in the [Modbus Map](#).

NumBytes [in]

The number of consecutive bytes.

aBytes [in]

An array of bytes to be transferred to the device. The array size should be equal to NumBytes.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM_errorcodes](#) or 0 for no error.

This function will append a 00 byte to aBytes for odd-numbered NumBytes.

If NumBytes is large enough, these functions will automatically split writes into multiple packets based on the current device's effective data packet size. Using both non-buffer and [buffer](#) registers in one function call is not supported.

Remarks

The Name version of this function is [LJM_eWriteNameByteArray](#).

Examples

[C/C++] Write a Lua script to LUA_SOURCE_WRITE

```
int LJMEError;
int ErrorAddress;
const char * luaScript =
    "LJ.IntervalConfig(0, 1000)\n"
    "while true do\n"
    " if LJ.CheckInterval(0) then\n"
```

```

" print(LJ.Tick())\n"
" end\n"
"end\n"
"0";
const unsigned scriptLength = strlen(luaScript) + 1;

// handle comes from LJM_Open()
LJMEError = LJM_eWriteAddressByteArray(
    handle,
    LUA_SOURCE_WRITE_ADDRESS,
    scriptLength,
    luaScript,
    &ErrorAddress
);
if (LJMEError != LJME_NOERROR) {
    // Deal with error
}

```

2.3.13 - eWriteNameArray

Write consecutive values, specified by a name.

Syntax

```

LJM_ERROR_RETURN LJM_eWriteNameArray(
    int Handle,
    const char * Name,
    int NumValues,
    double * aValues,
    int * ErrorAddress)

```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Name [in]

The name that specifies the Modbus register(s) to write. Names can be found throughout the device datasheet or in the [Modbus Map](#).

NumValues [in]

The number of consecutive values.

aValues [in]

An array of values to be transferred to the device. The array size should be equal to NumValues. Each value will be converted according to the type of Name.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

The Address version of this function is [LJM_eWriteAddressArray](#).

If NumValues is large enough, these functions will automatically split writes into multiple packets based on the current device's effective data packet size. Using both non-buffer and [buffer](#) registers in one function call is not supported.

Examples

[C/C++] Write DAC0 and DAC1 as an array

```

int LJMEError;
int errorAddress;
double newValues[2] = {1.2, 3.4};

// handle comes from LJM_Open()
LJMEError = LJM_eWriteNameArray(handle, "DAC0", 2, newValues, &errorAddress);
if (LJMEError != LJME_NOERROR) {
    // Deal with error
}

```

2.3.14 - eWriteNameByteArray

Write consecutive byte values, specified by a name.

Syntax

```
LJM_ERROR_RETURN LJM_eWriteNameByteArray(
    int Handle,
    const char * Name,
    int NumBytes,
    const char * aBytes,
    int * ErrorAddress)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Name [in]

The name that specifies the Modbus register(s) to write. Names can be found throughout the device datasheet or in the [Modbus Map](#).

NumBytes [in]

The number of consecutive bytes.

aBytes [in]

An array of bytes to be transferred to the device. The array size should be equal to NumBytes.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

[LJM_eReadNameByteArray](#) is the reverse of [LJM_eWriteNameByteArray](#).

The Address version of this function is [LJM_eWriteAddressByteArray](#).

This function will append a 00 byte to aBytes for odd-numbered NumBytes.

If NumBytes is large enough, these functions will automatically split writes into multiple packets based on the current device's effective data packet size. Using both non-buffer and [buffer](#) registers in one function call is not supported.

Examples

[C/C++] Write a Lua script to LUA_SOURCE_WRITE

```
int LJMErr;
int ErrorAddress;
const char * luaScript =
    "LJ.IntervalConfig(0, 1000)\n"
    "while true do\n"
    "  if LJ.CheckInterval(0) then\n"
    "    print(LJ.Tick())\n"
    "  end\n"
    "end\n"
    "\0";
const unsigned scriptLength = strlen(luaScript) + 1;

// handle comes from LJM_Open()
LJMErr = LJM_eWriteNameByteArray(
    handle,
    "LUA_SOURCE_WRITE",
    scriptLength,
    luaScript,
    &ErrorAddress
);
if (LJMErr != LJME_NOERROR) {
    // deal with error
}
```

2.4 - Stream Functions

[Add new comment](#)

To stream an unlimited number of scans from a device:

1. Start a stream using [LJM_eStartStream](#)
2. In a loop, read from the device stream using [LJM_eStreamRead](#)
 - Optionally, use [LJM_SetStreamCallback](#) instead of [LJM_eStreamRead](#)
3. End the stream using [LJM_eStreamStop](#)

To stream a finite number of scans, use [LJM_StreamBurst](#):

- [LJM_StreamBurst](#) is a convenience function that internally performs [LJM_eStreamStart](#), [LJM_eStreamRead](#), and [LJM_eStreamStop](#).

2.4.1 - eStreamStart

[Add new comment](#)

Initializes a stream object and begins streaming. This function creates a buffer in memory that holds data from the device, so that higher data throughput can be achieved.

Syntax

```
LJM_ERROR_RETURN LJM_eStreamStart(
    int Handle,
    int ScansPerRead,
    int NumAddresses,
    const int * aScanList,
    double * ScanRate)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

ScansPerRead [in]

The number of scans returned by each call to the [LJM_eStreamRead](#) function.

High-throughput applications should use a large *ScansPerRead* to get more data per call of [LJM_eStreamRead](#). A typical value would be equal to *ScanRate* or $1/2$ *ScanRate*, which results in a read once or twice per second, respectively.

[Low-latency](#) applications should use a smaller *ScansPerRead* to get less data per call of [LJM_eStreamRead](#). This will cause [eStreamRead](#) to be called more frequently.

The frequency [LJM_eStreamRead](#) returns data at is:

```
eStreamRead frequency = ScanRate / ScansPerRead
```

NumAddresses [in]

The number of addresses in *aScanList*. The size of the *aScanList* array.

aScanList [in]

An array of addresses to stream. The scan list is the list of data addresses that are to be buffered by LJM and returned with [LJM_eStreamRead](#). Find addresses in the [Modbus Map](#). For example, to stream AIN3, add the address 6 to the scan list.

ScanRate [in/out]

A pointer that sets the desired number of scans per second. Upon successful return of this function, *ScanRate* will be updated to the actual scan rate that the device will use. Keep in mind that [data rate limits](#) are specified in Samples/Second which is equal to $\text{NumAddresses} * \text{Scans/Second}$, or $\text{NumAddresses} * \text{ScanRate}$.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

To read data from stream, use [LJM_eStreamRead](#). To stop stream, use [LJM_eStreamStop](#).

Configuration

Channel configuration such as range and resolution must be handled elsewhere. Check the device datasheet for details about analog inputs and which connection types are capable of stream.

This function writes to the following registers, which cannot be set manually when using [LJM_eStreamStart](#):

- [STREAM_SCANRATE_HZ](#)
- [STREAM_NUM_ADDRESSES](#)
- [STREAM_SAMPLES_PER_PACKET](#)
- [STREAM_AUTO_TARGET](#)
- [STREAM_SCANLIST_ADDRESS#\(0:127\)](#)
- [STREAM_ENABLE](#)

Note that there are other [T-series stream configurations](#).

How samples are moved from the device to your application_

When stream is running, there are two sample buffers to consider: the device buffer and the LJM buffer. After starting stream:

1. The device acquires a scan of stream data at every scan interval according to its own clock and puts that data into the device buffer.
2. At the same time, LJM is running a background thread that moves data from the device buffer to the LJM buffer.
3. At the same time, the user application needs to move data from the LJM buffer to the user application—this is done using `LJM_eStreamRead` or `LJM_eStreamBurst`.

Both the device buffer and LJM buffer are first in, first out (FIFO) buffers.

Maximum Samples Per Packet

Each connection type has a maximum number of bytes per Modbus TCP packet sent between the device and host. That determines the maximum number of samples (given as Max Samples Per Packet below). A stream data packet has 16 bytes of overhead and each sample needs 2 bytes.

For the T4 and T7:

	Max Samples Per Packet	Max bytes per packet
USB	24	64
Ethernet	512	1040
WiFi	242	500

Actual Samples Per Packet

The actual samples per USB or TCP packet is configured by LJM. This will be the smaller of either `ScansPerRead * NumAddresses` (the parameters of `eStreamStart`) or the Max Samples Per Packet.

Samples Per Transfer

The samples per transfer is the number of samples transferred per read call. This is done automatically in the LJM background thread. This sets the interval for how often LJM does a read. An LJM read might consist of multiple USB or TCP packets.

The samples per transfer is limited by `ScansPerRead * NumAddresses` and the Max Samples Per Packet.

Normally, `ScansPerRead` is set to a higher value because it is more efficient to move bigger chunks of data less often, but if you want minimum latency between input/output you can set `ScansPerRead` as low as 1.

Externally clocked stream _

Externally clocked stream is when a signal is received on CIO3, for which each rising edge a single scan is triggered. This allows for variable scan rates and synchronized streaming.

When using externally clocked stream, `LJM_eStreamStart`'s `ScanRate` should be set to the fastest scan rate that the externally clocked stream will occur at. This ensures that LJM will collect data fast enough, without making LJM perform busy waits for stream data. This passed `ScanRate` has no effect on hardware in this case, but rather is just useful for LJM so it has an idea of expected data rate.

If externally clocked stream occurs at a variable scan rate or does not occur immediately after calling `LJM_eStreamStart`, you'll probably need to use an LJM configuration to prevent LJM from throwing an error when scans are not received by LJM within the expected timeframe since LJM normally expects a constant scan rate, and it will throw an error if it does not receive data from the device fast enough. Here are the options for setting LJM's configs for variable speed and/or delayed start streaming, which should be set using `LJM_WriteLibraryConfigS`:

- Set `LJM_STREAM_SCANS_RETURN` to `LJM_STREAM_SCANS_RETURN_ALL`. This will cause `LJM_eStreamRead` to return immediately with the error code `LJME_NO_SCANS_RETURNED` if LJM has not received `ScansPerRead` scans worth of data. LJM wrappers like `LJM Python` that throw exceptions instead of returning error codes will throw their language-specific version of `LJME_NO_SCANS_RETURNED`.
- Set `LJM_STREAM_RECEIVE_TIMEOUT_MS` to a safe timeout for your expected stream speeds. This should be the longest expected timeout, plus a small amount extra. If you're not sure how long of a timeout you need, 0 is the infinite timeout and will never time out.

The configs mentioned above are defined in `LabJackM.h`.

Triggered stream _

Hardware-Triggered

Hardware-triggered stream could also be called "delayed start stream". It is when `LJM_eStreamStart` has been called but stream does not start until a signal is received on FIO0 or FIO1, after which stream continues to collect data as normal.

To use triggered stream with LJM, you must enable some LJM configurations and also some T-series configurations.

LJM triggered stream configurations that should be set using `LJM_WriteLibraryConfigS`:

- Set `LJM_STREAM_SCANS_RETURN` to `LJM_STREAM_SCANS_RETURN_ALL`. This will cause `LJM_eStreamRead` to return immediately with the error code `LJME_NO_SCANS_RETURNED` if LJM has not received `ScansPerRead` scans worth of data. LJM wrappers like `LJM Python` that throw exceptions instead of returning error codes will throw their language-specific version of `LJME_NO_SCANS_RETURNED`.

- Set `LJM_STREAM_RECEIVE_TIMEOUT_MS` to a safe timeout for how long you expect it to take before stream starts. If you're not sure how long of a timeout you need, 0 is the infinite timeout and will never time out.

The configs mentioned above are defined in `LabJackM.h`.

Software-Triggered - For collecting pre-trigger data

Software-triggered stream is appropriate when pre-trigger data is required.

Software-triggered stream is when `LJM_eStreamStart` is called and stream immediately begins collecting data. The "trigger" point is then defined by software. Since this depends on the application, you would essentially iterate through the `aData` of `LJM_eStreamRead` until the desired trigger criteria is met. Data before that point is considered pre-trigger data.

Burst Stream

Burst stream is when a limited number of scans are collected by the device, after which the device stops streaming. `LJM_StreamBurst` may be used to automatically perform a burst stream or burst stream may be performed manually using the following steps:

- Write the desired number of scans to collect to `STREAM_NUM_SCANS`.
- Call `LJM_eStreamStart`
- Call `LJM_eStreamRead` until all `STREAM_NUM_SCANS` have been read or until the return code is `STREAM_BURST_COMPLETE` (2944)
- Call `LJM_eStreamStop` (to clean up memory)

Auto-recovery Precaution _

Auto-recovery mode is entered when the stream buffer on the LabJack device is too full. At that point, the device will discard further scans but will keep track of how many scans have been discarded. Once stream read has caught up and emptied the device stream buffer sufficiently, the device will stop discarding scans, and will send a data packet that reports the number of discarded scans and also has a scan of all `0xFFFF` that denotes the border between pre and post auto-recovery data. In the LJM stream buffer that is read by the user's application, LJM inserts the proper number of dummy scans, filled with all `-9999.0s`, to maintain proper timing of stream data through auto-recovery.

If auto-recovery is enabled (the default), and there is any risk of auto-recovery happening (typically happens at the highest data rates only), the first channel in the stream scan list should always be an analog input (AIN#). If you only want to stream other channels besides analog inputs and cannot add a dummy analog input because you need maximum speed, then you must either ensure that the first channel in the scan list will never return `0xFFFF` as a normal valid reading and enable `LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED` or see the [Stream Speed](#) section to reduce the chances of auto-recovery.

LJM actually looks at the first value in each scan to detect the border scan. Once auto-recovery starts, LJM watches the first channel in a scan and if it sees `0xFFFF` it assumes that is the border scan that marks the start of post auto-recovery data. `0xFFFF` is a special reserved value that analog inputs never return under normal circumstances.

Low-Latency _

Low-latency stream is for when you need to move data from the device to the host application as soon as possible after acquisition (i.e. with minimum latency). For example, you might be trying to correlate stream data with the host's clock, or perhaps are doing feedback control.

To do low-latency stream, set the `ScansPerRead` parameter (of `LJM_eStreamStart`) to a small number. This will cause `LJM_eStreamRead` or the callback of `LJM_SetStreamCallback` to read smaller amounts more often. LJM will also attempt to transfer data from the device according to how small `ScansPerRead` is:

- If `ScansPerRead` is large, LJM may transfer many samples at once. This is efficient.
- If `ScansPerRead` is 1 (the minimum), LJM will transfer one sample at a time from the device. This is less efficient, but allows for minimal latency.

Low-latency stream is typically limited to a scan rate of few kscans/second or less due to the increased overhead per scan. This increased overhead can cause the symptoms mentioned in the [Stream Speed](#) section.

To get stream latency as low as possible, avoid doing unnecessary work in either the thread that calls `LJM_eStreamRead` or in the callback set by `LJM_SetStreamCallback`. File I/O, hardware I/O, and screen updates can all take substantial time.

A real-world example:

During one test, we achieved reliable low-latency stream with:

- Up to 7250 Hz ScanRate
- 1 to 4 AIN channels
- 1 ScansPerRead
- USB connection
- LJM 1.2100
- macOS 10.13.6
- Processor: 2.7 GHz Intel Core i7
- Memory: 16 GB 1600 MHz DDR3

Stream Speed _

To ensure the host application is able to stream quickly, see our [troubleshooting guide](#) for the following situations:

- If `LJM_eStreamRead` is returning error 1301 (`LJME_LJM_BUFFER_FULL`)

- If LJM_eStreamRead gives many -9999 values in aData

Reducing Stream Startup Speed

As of LJM 1.1405: If stream has been successfully started for a given device handle without returning a LJME_USING_DEFAULT_CALIBRATION warning, the next stream start for that device handle will not load the stream calibration. This has been shown to reduce the stream startup time by about 6 milliseconds via USB or 10 or more milliseconds via TCP.

Example

[C/C++] Start stream with a scan rate of 10 kHz and 2 channels.

```
char ErrorString[LJM_MAX_NAME_SIZE];
int LJMErr;
const int numAddresses = 2;
const int aScanList[] = {0, 2}; // AIN0 and AIN1
const int initScanRate = 10000;
double scanRate = initScanRate;
const int scansPerRead = initScanRate / 2;
double aData[numAddresses * scansPerRead];
int DeviceScanBacklog;
int LJMScanBacklog;
int iteration = 0;
// Start stream
// handle from LJM_Open() or LJM_OpenS()
LJMErr = LJM_eStreamStart(handle, scansPerRead, numAddresses, aScanList, &scanRate);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_eStreamStart error: %s\n", ErrorString);
}
else {
    printf("Stream started at %f Hz\n", scanRate);
}
// Read stream
for (iteration = 0; iteration < 10; iteration++) {
    LJMErr = LJM_eStreamRead(handle, aData, &DeviceScanBacklog, &LJMScanBacklog);
    if (LJMErr != 0) {
        LJM_ErrorToString(LJMErr, ErrorString);
        printf("LJM_eStreamRead error: %s\n", ErrorString);
    }

    // Process the stream data in whatever way suits your application
    MyDataProcessingFunction(aData, DeviceScanBacklog, LJMScanBacklog);
}
// Stop stream
LJMErr = LJM_eStreamStop(handle);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_eStreamStop error: %s\n", ErrorString);
}
```

2.4.2 - eStreamRead

[Add new comment](#)

Returns data from an initialized and running LJM stream buffer. Waits for data to become available, if necessary.

Syntax

```
LJM_ERROR_RETURN LJM_eStreamRead(
    int Handle,
    double * aData,
    int * DeviceScanBacklog,
    int * LJMScanBacklog)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

aData [out]

An array that contains the values being read. Returns all channels interleaved. Must be large enough to hold (ScansPerRead * NumAddresses) values, where ScansPerRead and NumAddresses are values passed to [LJM_eStreamStart](#). The data returned is removed from the LJM stream buffer. This is done according to first in, first out (FIFO) order.

DeviceScanBacklog [out]

The number of scans left in the device buffer, as measured from when data was last collected from the device. DeviceScanBacklog should usually be near zero and not growing.

LJMScanBacklog [out]

The number of scans left in the LJM buffer, which does not include concurrent data sent in the aData array. LJMScanBacklog should usually be near

zero and not growing.

Returns

[LJM_errorcodes](#) or 0 for no error.

If LJM_eStreamRead returns an error, LJM nearly always attempts to command the device to stop streaming. LJM does not do so for the case when LJME_NO_SCANS_RETURNED is returned (due to [LJM_STREAM_SCANS_RETURN_ALL_OR_NONE](#)).

Remarks

Before calling this function, create a data buffer using [LJM_eStreamStart](#). To stop stream, use [LJM_eStreamStop](#).

If LJM_eStreamRead gives error 1301 (LJME_LJM_BUFFER_FULL) or many -9999 values in aData, here are some [strategies to help](#).

Example

See the [LJM_eStreamStart](#) page for an example using LJM_eStreamRead.

2.4.3 - eStreamStop

Stops the device from collecting data in stream mode and stops LJM from storing any more data from the device.

Syntax

```
LJM_ERROR_RETURN LJM_eStreamStop(int Handle)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Returns

[LJM_errorcodes](#) or 0 for no error.

Having trouble with stream? See our [troubleshooting guide](#).

Remarks

This function writes to the following registers:

- STREAM_ENABLE

Use LJM_eStreamStop when the data streaming session is completed, or to temporarily pause new data from being stored. To begin streaming again, use [LJM_eStreamStart](#).

When LJM_eStreamStop is called, LJM will maintain any previously collected data in the buffer to be read. Starting a new stream will clear the buffer from the old stream.

Example

See the [LJM_eStreamStart](#) page for an example using LJM_eStreamStop.

2.4.4 - StreamBurst

Initializes a [stream burst](#) and collects data. This function combines [LJM_eStreamStart](#), [LJM_eStreamRead](#), and [LJM_eStreamStop](#), as well as some other device initialization.

Syntax

```
LJM_ERROR_RETURN LJM_StreamBurst(
    int Handle,
    int NumAddresses,
    const int * aScanList,
    double * ScanRate,
    unsigned int NumScans,
    double * aData)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

NumAddresses [in]

The number of addresses in aScanList. The size of the aScanList array.

aScanList [in]

An array of addresses to stream. The scan list is the list of data addresses that are to be buffered by LJM and returned in aData. Find addresses in the [Modbus Map](#). For example, to stream AIN3 add the address 6 to the scan list.

ScanRate [in/out]

A pointer that sets the desired number of scans per second. Upon successful return of this function, ScanRate will be updated to the actual scan rate that the device used. Keep in mind that [data rate limits](#) are specified in Samples/Second which is equal to NumAddresses * Scans/Second, or NumAddresses * ScanRate .

NumScans [in]

The number of scans to collect. This is how many burst scans are collected and may not be zero.

aData [out]

An array that contains the values being read. Returns all channels interleaved. Must be large enough to hold NumScans * NumAddresses values.

Returns

LJM errorcodes or 0 for no error.

Remarks

LJM_StreamBurst does not write to STREAM_BUFFER_SIZE_BYTES.

Address configuration such as range, resolution, and differential voltages must be handled by writing to the device.

This function will block for NumScans / ScanRate seconds or longer.

For more details about stream, see the [LJM_eStreamStart](#) page.

Example

[C/C++] Stream AIN0 and FIO_STATE for 10 scans at 2kHz.

```
int err;
int numChannels = 2;
int aScanList[2] = {0, 2500};
double scanRate = 2000;
int numScans = 10;

unsigned int numSamples = numChannels * numScans;
double * aBurstSamples = malloc(sizeof(double) * numSamples);

err = LJM_StreamBurst(handle, numChannels, aScanList, &scanRate, numScans,
aBurstSamples);
if (err != LJME_NOERROR) {
// Handle error
}

// Print the scans in aBurstSamples

free(aBurstSamples);
```

2.4.5 - SetStreamCallback

Sets a callback that is called by LJM when the stream has collected ScansPerRead scans. The callback should call [LJM_eStreamRead](#).

LJM_SetStreamCallback is alternative to manually calling [LJM_eStreamRead](#). Manually calling [LJM_eStreamRead](#) is more straightforward than using LJM_SetStreamCallback, but LJM_SetStreamCallback allocates a work thread within LJM that calls the callback. Because of this, it's more appropriate for applications that need to perform other work, such as updating a GUI.

Syntax

```
typedef void (*LJM_StreamReadCallback)(void *);
LJM_ERROR_RETURN LJM_SetStreamCallback(
int Handle,
LJM_StreamReadCallback Callback,
void * Arg)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Callback [in]

The callback function for LJM's stream thread to call when ScansPerRead scans of stream data are ready, which should call [LJM_eStreamRead](#) to acquire data. ScansPerRead is a value of [LJM_eStreamStart](#).

Arg [in]

The user-defined argument that is passed to Callback when it is invoked.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

Before calling this function, start stream using [LJM_eStreamStart](#).

The Callback function is executed by a dedicated thread within LJM. [LJM_SetStreamCallback](#) should not be called from within Callback. In versions of LJM before 1.1405, [LJM_eStreamStop](#) should not be called from within Callback.

To disable the previous callback for stream reading, pass 0 or NULL as Callback.

As an alternative to [LJM_SetStreamCallback](#), the LJM configuration [LJM_STREAM_SCANS_RETURN](#) may also be useful. It controls the block on [LJM_eStreamRead](#), allowing [LJM_eStreamRead](#) to return immediately with an error if a full read of data is not ready.

2.4.6 - GetStreamTCPReceiveBufferStatus

Requires LJM 1.2000 or later

For handles running stream over the network, gets the current state of the receive buffer.

The Receive Buffer

All TCP streams have a receive buffer. In the context of receiving stream data from a LabJack device, the receive buffer is a buffer on the host computer that is managed by the operating system. The LabJack device sends packets to this buffer, which LJM then reads from.

As the receive buffer fills up with bytes, the host computer will report to the LabJack device that it has a reduced receive window. For example, if there are 1040 bytes of free space remaining in the computer's receive buffer, TCP protocol will report to the device that the computer has a receive window of 1040. The device will then be able to send 1040 bytes and no more until the computer sends another packet declaring that it has freed some space from the receive buffer.

Once the receive window is completely full, the LabJack device is temporarily not able to send more packets. Once this happens, it begins filling its own device scan buffer. Once the device scan buffer is completely full, it will stop collecting scans and [auto-recovery](#) will occur.

During stream, LJM has a dedicated thread that does nothing but read from the receive window. This usually prevents the receive window from being completely filled, but some computers may be, in effect, slower than the LabJack device—such that the device sends bytes faster than the host computer reads bytes from the receive window. For this reason, it is important to monitor the state of the receive buffer if auto-recovery is occurring.

Syntax

```
LJM_ERROR_RETURN LJM_GetStreamTCPReceiveBufferStatus(
    int Handle,
    unsigned int * ReceiveBufferBytesSize,
    unsigned int * ReceiveBufferBytesBacklog)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

ReceiveBufferBytesSize [out]

Returns the current maximum number of bytes that can be stored in the receive buffer before it is full.

ReceiveBufferBytesBacklog [out]

Returns the current number of bytes stored in the receive buffer.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

Before calling this function, start stream using [LJM_eStreamStart](#).

If the device handle's connection type is Ethernet or WiFi, this function should be called after [LJM_eStreamRead](#) if auto-recovery is occurring.

If the device handle's connection type is USB, this function returns meaningless values.

Use [LJM_GetHandleInfo](#) to get the handle's connection type.

If the ratio of `ReceiveBufferBytesBacklog / ReceiveBufferBytesSize` is consistently increasing, auto-recovery is likely to happen. If the `ReceiveBufferBytesBacklog` is staying relatively low but auto-recovery is still occurring, there may be network reliability issues or other issues. In either case, see [strategies for reducing auto-](#)

[recovery.](#)

2.5 - Device Information Functions

[Add new comment](#)

LJM_GetHandleInfo

[LJM_GetHandleInfo](#) returns details about a device handle.

Device Discovery

The ListAll functions search for available LabJack device connections. [LJM_ListAll](#) and [LJM_ListAllS](#) are the same function, except that [LJM_ListAll](#) uses integer parameters to filter what connections are searched for, while [LJM_ListAllS](#) uses string parameters to do the same. [LJM_ListAllExtended](#) allows arbitrary device registers to be queried.

Device Reconnection

In a real-world setup, device connections can intermittently fail. LJM does its best to reconnect to devices and [LJM_RegisterDeviceReconnectCallback](#) can notify client code when reconnect has occurred.

2.5.1 - GetHandleInfo

Returns details about a device handle, which is simply a connection ID to an active device.

Syntax

```
LJM_ERROR_RETURN LJM_GetHandleInfo(
    int Handle,
    int * DeviceType,
    int * ConnectionType,
    int * SerialNumber,
    int * IPAddress,
    int * Port,
    int * MaxBytesPerMB)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

DeviceType [out]

The device type corresponding to a constant:

[LJM_dtT7](#) (7) for T7 series device

[LJM_dtDIGIT](#) (200) for Digit series device

ConnectionType [out]

The connection type corresponding to a constant:

[LJM_ctUSB](#) (1) for USB

[LJM_ctETHERNET](#) (3) for Ethernet

[LJM_ctWIFI](#) (4) for WiFi

SerialNumber [out]

The serial number of the device.

IPAddress [out]

The integer representation of the device's IP address when *ConnectionType* is TCP-based. Unless *ConnectionType* is TCP-based, this will be [LJM_NO_IP_ADDRESS](#). Note that this can be converted to a human-readable string with the [LJM_NumberToIP](#) function.

Port [out]

The port number when the device connection is TCP-based, or the pipe if the device connection is USB-based.

MaxBytesPerMB [out]

The maximum packet size in number of bytes that can be sent/received from this device. Note that this can change, depending on connection type and device type. This information is important if you are using the low-level functions, or when passing large arrays into other functions.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

This function is useful to quickly see information about a device handle, such as the serial number, or maximum packet size. In the event that "ANY"-type parameters were used in opening, this function can be used to see which device was actually opened.

Examples

[C/C++] Retrieve information about a handle and print it.

```
int LJMErr;
int portOrPipe, ipAddress, serialNumber, packetMaxBytes;
int deviceType = LJM_dtANY;
int connectionType = LJM_ctANY;
char string[LJM_STRING_ALLOCATION_SIZE];
char ipAddressString[LJM_IPv4_STRING_SIZE];

LJMErr = LJM_GetHandleInfo(handle, &deviceType, &connectionType, &serialNumber, &ipAddress,
    &portOrPipe, &packetMaxBytes);
if (LJMErr) {
    LJM_ErrorToString(LJMErr, string);
    printf("LJM_GetHandleInfo error: %s (%d)\n", string, LJMErr);
}
else {
    printf("deviceType: %d\n", deviceType);
    printf("connectionType: %d\n", connectionType);
    printf("serialNumber: %d\n", serialNumber);

    if (connectionType == LJM_ctETHERNET || connectionType == LJM_ctWIFI) {
        LJM_NumberToIP(ipAddress, ipAddressString);
        printf("IP address: %s\n", ipAddressString);
    }

    if (connectionType == LJM_ctUSB) {
        printf("pipe: %d\n", portOrPipe);
    }
    else {
        printf("port: %d\n", portOrPipe);
    }

    printf("The maximum number of bytes you can send to or receive from this device in one packet is %d bytes.\n",
        packetMaxBytes);
}
```

2.5.2 - ListAll

Scans for LabJack devices, returning arrays describing the devices found. The use of "ANY"-type for DeviceType and ConnectionType allow this function to perform a broad search.

Syntax

```
LJM_ERROR_RETURN LJM_ListAll(
    int DeviceType,
    int ConnectionType,
    int * NumFound,
    int * aDeviceTypes,
    int * aConnectionTypes,
    int * aSerialNumbers,
    int * aIPAddresses)
```

Parameters

DeviceType [in]

Filter device type to find. 4 for T4 devices, 7 for T7 devices, 84 for T-series devices, 200 for Digit devices, etc. 0 for ANY is allowed.

For other LabJack devices, see [What driver/library should I use with my LabJack?](#)

ConnectionType [in]

Filter connection type to scan. 1 for USB, 2 for TCP, 3 for Ethernet, 4 for WIFI. 0 for ANY is allowed.

NumFound [out]

A pointer that returns the number of devices found.

aDeviceTypes [out]

An array of device types, one for each of the NumFound devices. Must be preallocated to size LJM_LIST_ALL_SIZE.

aConnectionTypes [out]

An array of connection types, one for each of the NumFound devices. Must be preallocated to size LJM_LIST_ALL_SIZE.

aSerialNumbers [out]

An array of serial numbers, one for each of the NumFound devices. Must be preallocated to size LJM_LIST_ALL_SIZE.

aIPAddresses [out]

An array of IP Addresses, one for each (if applicable) of the NumFound devices. When the device is not TCP capable, IP address will be

LJM_NO_IP_ADDRESS.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

This function is useful for big programs that open multiple kinds of devices, especially when the device type and connection type are unknown. This function only shows what devices could be opened. To actually open a device, use [LJM_Open](#) or [LJM_OpenS](#).

When the ConnectionType parameter of this function is network-based, this function will check the IP addresses listed in [LJM_Specific_IPs](#).

Examples

See examples/utilities/list_all.c, available in the [LJM C++ examples](#).

2.5.3 - ListAllS

Scans for LabJack devices, returning arrays describing the devices found. The use of "ANY" for DeviceType and ConnectionType allow this function to perform a broad search.

Syntax

```
LJM_ERROR_RETURN LJM_ListAllS(
    const char * DeviceType,
    const char * ConnectionType,
    int * NumFound,
    int * aDeviceTypes,
    int * aConnectionTypes,
    int * aSerialNumbers,
    int * aIPAddresses)
```

Parameters

DeviceType [in]

Filter device type to find. "T4" for T4 devices, "T7" for T7 devices, "TSERIES" for T-series devices, "Digit" for Digit devices, etc. "ANY" is allowed.

For other LabJack devices, see [What driver/library should I use with my LabJack?](#)

ConnectionType [in]

Filter connection type to scan. "USB" for USB connection, "TCP" for TCP connection, etc. "ANY" is allowed.

NumFound [out]

A pointer that returns the number of devices found.

aDeviceTypes [out]

An array of device types, one for each of the NumFound devices. Must be preallocated to size LJM_LIST_ALL_SIZE.

aConnectionTypes [out]

An array of connection types, one for each of the NumFound devices. Must be preallocated to size LJM_LIST_ALL_SIZE.

aSerialNumbers [out]

An array of serial numbers, one for each of the NumFound devices. Must be preallocated to size LJM_LIST_ALL_SIZE.

aIPAddresses [out]

An array of IP Addresses, one for each (if applicable) of the NumFound devices. When the device is not TCP capable, IP address will be LJM_NO_IP_ADDRESS.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

This function is useful for big programs that open multiple kinds of devices, especially when the device type and connection type are unknown. This function only shows what devices could be opened. To actually open a device, use [LJM_Open](#) or [LJM_OpenS](#).

When the ConnectionType parameter of this function is network-based, this function will check the IP addresses listed in [LJM_SPECIAL_ADDRESSES_FILE](#).

Examples

See examples/utilities/list_all.c, available in the [LJM C++ examples](#).

Device Not Found?

If a device isn't properly being found, some important debugging information can be found in our [Device Not Found?](#) App Note. If a network connection needs to be debugged, read through our [Setup WiFi and Ethernet](#) App Note as well.

2.5.4 - ListAllExtended

Advanced version of [LJM_ListAll](#) that performs an additional query of arbitrary registers on the device.

Syntax

```
LJM_ERROR_RETURN LJM_ListAllExtended(
    int DeviceType,
    int ConnectionType,
    int NumAddresses,
    const int * aAddresses,
    const int * aNumRegs,
    int MaxNumFound,
    int * NumFound,
    int * aDeviceTypes,
    int * aConnectionTypes,
    int * aSerialNumbers,
    int * aIPAddresses,
    unsigned char * aBytes)
```

Parameters

DeviceType [in]

Filter device type to find. 4 for T4 devices, 7 for T7 devices, 84 for T-series devices, 200 for Digit devices, etc. 0 for ANY is allowed.

ConnectionType [in]

Filter connection type to scan. 1 for USB, 2 for TCP, 3 for Ethernet, 4 for WIFI. 0 for ANY is allowed.

NumAddresses [in]

The number of addresses to query. Also the size of *aAddresses* and *aNumRegs*.

aAddresses [in]

The addresses to query for each device that is found.

aNumRegs [in]

The addresses to query for each device that is found. Each *aNumRegs*[*i*] corresponds to *aAddresses*[*i*].

MaxNumFound [in]

The maximum number of devices to find. Also the size of *aDeviceTypes*, *aConnectionTypes*, *aSerialNumbers*, and *aIPAddresses*.

NumFound [out]

A pointer that returns the number of devices found.

aDeviceTypes [out]

An array of device types, one for each of the *NumFound* devices. Must be preallocated to size *MaxNumFound*.

aConnectionTypes [out]

An array of connection types, one for each of the *NumFound* devices. Must be preallocated to size *MaxNumFound*.

aSerialNumbers [out]

An array of serial numbers, one for each of the *NumFound* devices. Must be preallocated to size *MaxNumFound*.

aIPAddresses [out]

An array of IP Addresses, one for each (if applicable) of the *NumFound* devices. When the device is not TCP capable, IP address will be `LJM_NO_IP_ADDRESS`.

aBytes [out]

An array that must be preallocated to size:
 $\text{MaxNumFound} * \langle \text{the sum of } a\text{NumRegs} \rangle * \text{LJM_BYTES_PER_REGISTER}$,
 which will contain the query bytes sequentially. A device represented by index *i* would have an *aBytes* index of:
 $(i * \langle \text{the sum of } a\text{NumRegs} \rangle * \text{LJM_BYTES_PER_REGISTER})$.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

When the *ConnectionType* parameter of this function is network-based, this function will check the IP addresses listed in [LJM_SPECIAL_ADDRESSES_FILE](#).

Examples

See [examples/utilities/list_all_extended.c](#), available in the [LJM C++ examples](#).

2.5.5 - RegisterDeviceReconnectCallback

Sets a callback that is called by LJM after the device is found to be disconnected (resulting in a read/write error) and the device is then reconnected.

Syntax

```
typedef void (*LJM_DeviceReconnectCallback)(int);
LJM_ERROR_RETURN LJM_RegisterDeviceReconnectCallback(
    int Handle,
    LJM_DeviceReconnectCallback Callback)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Callback [in]

The callback function which will receive the device Handle as a parameter.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

For more information on device reconnection, see "[2.1 - Opening and Closing](#)".

To disable the previous callback for reconnection, pass 0 or NULL as Callback.

LJM_RegisterDeviceReconnectCallback may not be called from within a LJM_DeviceReconnectCallback.

2.6 - Utility Functions

2.6.1 - ErrorToString

Get the name of an error code.

Syntax

```
LJM_VOID_RETURN LJM_ErrorToString(
    int ErrorCode,
    char * ErrorString)
```

Parameters

ErrorCode [in]

The error code number.

ErrorString [out]

A pointer to the char array which will be populated with the null-terminated error name/description. Allocate to size LJM_MAX_NAME_SIZE.

Returns

None.

Remarks

If the constants file that has been loaded does not contain the error code, this returns a null-terminated message saying so. If the constants file could not be opened, this returns a null-terminated string containing the expected file path.

Examples

[C/C++] Get the name of error code 1230

```
char errorName[LJM_MAX_NAME_SIZE]; // LJM_MAX_NAME_SIZE is 256
LJM_ErrorToString(1230, errorName);
printf ("%s\n", errorName);
// prints "LJME_COULD_NOT_CLAIM_DEVICE"
```

2.6.2 - TCVoltsToTemp

Converts thermocouple voltage to temperature.

Syntax

```
LJM_ERROR_RETURN LJM_TCVoltsToTemp(
    int TCType,
    double TCVolts,
    double CJTempK,
    double * pTCTempK)
```

Parameters

TCType [in]

The thermocouple type. Constants are (alphabetically):

B = 6001

C = 6009

E = 6002

J = 6003

K = 6004

N = 6005

R = 6006

S = 6007

T = 6008

TCVolts [in]

The voltage reported by the thermocouple.

CJTempK [in]

The cold-junction temperature in degrees Kelvin.

pTCTempK [out]

The calculated thermocouple temperature in degrees Kelvin.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

Use this function to easily calculate thermocouple temperatures from voltage readings. When thermocouples are connected to the AIN0-AIN3 screw terminals, the cold-junction temperature can be obtained from AIN14, which maps to a nearby temp sensor inside most LabJack devices.

Examples

[C/C++] Get the temperature of a K type thermocouple.

```
int LJLError;
double TCTempKelvin;
```

```
LJLError = LJM_TCVoltsToTemp(6004, 0.00134, 299.039, &TCTempKelvin);
printf("%f\n", TCTempKelvin);
```

2.6.3 - NumberToIP

Takes an integer representing an IPv4 address and outputs the corresponding decimal-dot IPv4 address as a null-terminated string.

Syntax

```
LJM_ERROR_RETURN LJM_NumberToIP(
    unsigned int Number,
    char * IPv4String)
```

Parameters

Number [in]

A number representing an IPv4 address.

IPv4String [out]

A character array which will be updated to contain the null-terminated string representation of an IPv4 address. Must be allocated to size LJM_IPv4_STRING_SIZE.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

Use this function to get a more human-readable interpretation of an IP address.

Examples

[C/C++] Convert a number into an IPv4 string.

```
int LJMError;
char IPv4String[16];
```

```
LJMError = LJM_NumberToIP(3232235983, IPv4String);
printf ("%s \n", IPv4String);
// Prints "192.168.1.207"
```

2.6.4 - NumberToMAC

Takes an integer representing a MAC address and outputs the corresponding hex-colon MAC address as a null-terminated string.

Syntax

```
LJM_ERROR_RETURN LJM_NumberToMAC(
    unsigned long long Number,
    char * MACString)
```

Parameters

Number [in]

The number representing a MAC address.

MACString [out]

A character array which will be updated to contain the null-terminated string representation of the MAC address. Must be allocated to size LJM_MAC_STRING_SIZE.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

Use this function to get a more human-readable interpretation of a MAC address.

Examples

[C/C++] Convert a number into a MAC string.

```
int LJMError;
char MACString[18];
```

```
LJMError = LJM_NumberToIP(81952921372024, MACString);
printf ("%s \n", MACString);
// prints "78:45:C4:26:89:4A"
```

2.6.5 - IPToNumber

Takes a decimal-dot IPv4 string representing an IPv4 address and outputs the corresponding integer version of the address.

Syntax

```
LJM_ERROR_RETURN LJM_IPToNumber(
    const char * IPv4String,
    unsigned int * Number)
```

Parameters

IPv4String [in]

A decimal dot string representing an IPv4 address.

Number [out]

The numerical representation of IPv4String.

Returns

[LJM errorcodes](#) or 0 for no error. Returns LJME_INVALID_PARAMETER if IPv4String could not be parsed as an IPv4 address.

Remarks

Use this function to convert a human-readable interpretation of an IP address into an integer value.

Examples

[C/C++] Convert an IPv4 string into a number.

```
int LJMError;
unsigned int Number;

LJMError = LJM_IPToNumber("192.168.1.207", &Number);
printf ("%u \n", Number);
// prints 3232235983
```

2.6.6 - MACToNumber

Takes a hex-colon string representing a MAC address and outputs the corresponding integer version of the address.

Syntax

```
LJM_ERROR_RETURN LJM_MACToNumber(
    const char * MACString,
    unsigned long long * Number)
```

Parameters

MACString [in]

A hex-colon string representing a MAC address.

Number [out]

The numerical representation of MACString.

Returns

[LJM errorcodes](#) or 0 for no error. Returns LJME_INVALID_PARAMETER if MACString could not be parsed as a MAC address.

Remarks

Use this function to convert a human-readable interpretation of a MAC address into an integer value.

Examples

[C/C++] Convert a MAC string into a number.

```
int LJMError;
unsigned long long Number;

LJMError = LJM_MACToNumber("78:45:C4:26:89:4A", &Number);
printf ("%llu \n", Number);
// prints 81952921372024
```

2.6.7 - NameToAddress

[Add new comment](#)

Takes a Modbus register name as input and produces the corresponding Modbus address and type. These two values can serve as input to functions that have Address and Type as input parameters.

Syntax

```
LJM_ERROR_RETURN LJM_NameToAddress(
    const char * Name,
    int * Address,
    int * Type)
```

Parameters

Name [in]

A null-terminated c-string register identifier. This register identifiers can be a register name or a register alternate name.

Address [out]

Output parameter containing the address specified by Name.

Type [out]

Output parameter containing the type specified by Name.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

If Name is not a valid register identifier, Address will be set to LJM_INVALID_NAME_ADDRESS.

Examples

[C/C++] Get the address and type of "AIN3".

```
int LJLError;
int Address;
int Type;
LJLError = LJM_NameToAddress("AIN3", &Address, &Type);
printf("%d\n", Address);
// prints 6
printf("%d\n", Type);
// prints 3 for LJM_FLOAT32
```

2.6.8 - NamesToAddresses

Takes a list of Modbus register names as input and produces two lists as output - the corresponding Modbus addresses and types. These two lists can serve as input to functions that have Address/aAddresses and Type/aTypes as input parameters.

Syntax

```
LJM_ERROR_RETURN LJM_NamesToAddresses(
    int NumFrames,
    const char ** aNames,
    int * aAddresses,
    int * aTypes)
```

Parameters

NumFrames [in]

The number of names in aNames and the allocated size of aAddresses and aTypes.

aNames [in]

An array of null-terminated c-string register identifiers. These register identifiers can be register names or register alternate names.

aAddresses [out]

An output array containing the addresses specified by aNames in the same order, must be allocated to the size NumFrames before calling LJM_NamesToAddresses.

aTypes [out]

An output array containing the types specified by aNames in the same order, must be allocated to the size NumFrames before calling LJM_NamesToAddresses.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

For each register identifier in aNames that is invalid, the corresponding aAddresses value will be set to LJM_INVALID_NAME_ADDRESS.

Examples

[C/C++] Get the addresses and types of "AIN3" and "DAC0".

```
int LJLError;
const char * aNames[2] = {"AIN3", "DAC0"};
int aAddresses[2];
int aTypes[2];
int AIN3Address;
int AIN3Type;
int DAC0Address;
int DAC0Type;

LJLError = LJM_NamesToAddresses(2, aNames, aAddresses, aTypes);
AIN3Address = aAddresses[0];
AIN3Type = aTypes[0];
DAC0Address = aAddresses[1];
DAC0Type = aTypes[1];
```

2.6.9 - AddressToType

Retrieves the data type for a given Modbus register address.

Syntax

```
LJM_ERROR_RETURN LJM_AddressToType(
```

```
int Address,
int * Type)
```

Parameters

Address [in]

A Modbus address.

Type [out]

Output parameter containing the data type of Address.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

Convenience function to programmatically discover the type of a Modbus address. Most people can simply look in the [Modbus Map](#).

Examples

[C/C++] Get the type of Modbus address 6, which is AIN3.

```
int LJMErr;
int Type;
LJMErr = LJM_AddressesToTypes(6, &Type);
printf("%d\n", Type);
// 3 for LJM_FLOAT32
```

2.6.10 - AddressesToTypes

Retrieves multiple data types for given Modbus register addresses.

Syntax

```
LJM_ERROR_RETURN LJM_AddressesToTypes(
    int NumAddresses,
    int * aAddresses,
    int * aTypes)
```

Parameters

NumAddresses [in]

The number of addresses to process.

aAddresses [in]

The array of address numbers to process. Set to size NumAddresses.

aTypes [out]

An output array containing the data types of each address specified in aAddresses. Set to size NumAddresses.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

Convenience function to programmatically discover the types of many Modbus addresses. Most people can simply look in the [Modbus Map](#). For each aAddresses[i] that is not found, the corresponding entry aTypes[i] will be set to LJM_INVALID_NAME_ADDRESS and this function will return LJM_INVALID_ADDRESS.

Examples

[C/C++] Get the type of Modbus address 6, which is AIN3, and Modbus address 2002, which is FIO2.

```
int LJMErr;
int aAddresses[2] = {6, 2002};
int aTypes[2];
int TypeAIN3;
int TypeFIO2;
LJMErr = LJM_AddressesToTypes(2, aAddresses, aTypes);
TypeAIN3 = aTypes[0]; //LJM_FLOAT32
TypeFIO2 = aTypes[1]; //LJM_UINT16
```

2.7 - Timing Functions

Use `LJM_GetHostTick` or `LJM_GetHostTick32Bit` to get the current system time.

Use `LJM_StartInterval`, `LJM_WaitForNextInterval`, and `LJM_CleanInterval` to perform periodic operations.

2.7.1 - GetHostTick

Queries the host system's steady (monotonic) clock, preferentially with high precision.

Syntax

```
LJM_LONG_LONG_RETURN LJM_GetHostTick()
```

Returns

The current clock tick in microseconds, as a 64-bit integer. Resolution may vary.

On Windows, sleep states (such as standby, hibernate, or connected standby) do not affect `LJM_GetHostTick`. Other platforms should be manually tested.

Compatibility

Some development environments can not handle the 64-bit return value. In such cases use [LJM_GetHostTick32Bit](#) which uses two 32-bit parameters to return the clock tick.

Example

[C/C++] Measure how long an operation takes.

```
double value;
long long time0 = LJM_GetHostTick();
err = LJM_eReadName(handle, "SERIAL_NUMBER", &value);
long long time1 = LJM_GetHostTick();
ErrorCheck(err, "LJM_eReadName");
printf("LJM_eReadName took %lld microseconds.\n", time1 - time0);
```

Device I/O times can vary greatly, but this prints something like:

```
LJM_eReadName took 338 microseconds.
```

Determining What's Slow

There are a number of spots in your code that could be incurring a delay. These include:

- Communication with a LabJack device
- Communication with other devices
- Other I/O
- Graphics
- CPU-intensive tasks

If your loop is too slow, try timing which part of the loop is slow. A good way to do this is to measure two sections: how long a section of code takes and how long the entire loop takes. Then, output both times and compare.

With pseudo-C code this could look like the following:

```
long long tloop = 0;
for (...) {
    long long t0 = LJM_GetHostTick();
    // The section under test goes here
    long long t1 = LJM_GetHostTick();

    if (tloop != 0) {
        printf("The code section took %lld microseconds;", t1 - t0);
        printf(" the whole loop took %lld microseconds\n", t1 - tloop);
    }
    tloop = t1;
}
```

After the first loop, this uses the second `GetHostTick` and a temporary variable to measure total loop time.

Each line shows how long the code section took compared to the whole loop:

```
The code section took 370 microseconds; the whole loop took 378 microseconds
The code section took 325 microseconds; the whole loop took 333 microseconds
...
```

2.7.2 - GetHostTick32Bit

Gets the current clock tick, exactly like [LJM_GetHostTick](#), except it uses two 32-bit numbers.

Syntax

```
LJM_VOID_RETURN LJM_GetHostTick32Bit(
    unsigned int * TickUpper,
    unsigned int * TickLower)
```

Parameters

TickUpper [out]

The upper (most significant) 32-bits of the clock tick.

TickLower [out]

The lower (least significant) 32-bits of the clock tick.

Returns

None.

Compatibility

Most modern development environments can use the normal [GetHostTick](#) function which provides the clock tick as a 64-bit return value. This 32-bit function is provided for environments that can not handle 64-bit values, since all environments can pass 32-bit parameters. For example, LabVIEW 7.1 does not support 64-bit return values but has no problem with the 32-bit parameters, so our applicable [LabVIEW VI](#) uses LJM_GetHostTick32Bit. The clock tick values can be kept as two separate 32-bit values or combined—as done in LJM_GetHostTick32Bit.vi, where two 32-bit integers are combined into one extended precision float since that is a LabVIEW 7.1 data type that will hold a 64-bit integer.

Example

[C/C++] Get the current host clock tick and combine it into a single 64-bit number.

```
unsigned int TickUpper;
unsigned int TickLower;
long long time;
LJM_GetHostTick32Bit(&TickUpper, &TickLower);
time = ((long long)TickUpper << 32) + TickLower;
printf("The host tick is %lld\n", time);
```

2.7.3 - StartInterval

Sets up a reoccurring interval timer.

Syntax

```
LJM_ERROR_RETURN LJM_StartInterval(
    int IntervalHandle,
    int Microseconds)
```

Parameters

IntervalHandle [in]

The user-generated interval identifier. This can be anything.

Microseconds [in]

The number of microseconds in the interval.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

This function allocates memory for the given `IntervalHandle` and begins a reoccurring interval timer. This function does not perform any waiting.

In many applications an interval timer is preferable to using a normal delay (sleep), as it is difficult to get an exact interval using a delay. For example, say a loop makes an I/O call that takes 5 to 15 milliseconds (ms) each time, and this is followed by a 90 ms delay. The loop will take 95 to 105 ms per iteration, and the average time per iteration over time could be anywhere in that range. Using interval timers avoids this problem because the loop will always take the specified number of milliseconds (or else LJM_WaitForNextInterval will notify you of skipped intervals).

Jitter and clock skew may occur, but the average interval period over time (ignoring skipped intervals) will be equal to the number of specified `Microseconds`.

For more precise timing, use [Lua scripting](#) or [streaming](#).

Examples

[C/C++] Read AIN0 once per second, 100 times.

Given a `handle` to a device, a function named `ReadAndOutput` that reads and outputs a registers, and an error-checking function called `ErrorCheck`, the following example reads `AIN0` once per second, 100 times:

```
int err;
int SkippedIntervals;
const int INTERVAL_HANDLE = 1;
err = LJM_StartInterval(INTERVAL_HANDLE, 1000 * 1000);
ErrorCheck(err, "LJM_StartInterval");

for (int loop = 0; loop < 100; loop++) {
    ReadAndOutput(handle, "AIN0");

    err = LJM_WaitForNextInterval(INTERVAL_HANDLE, &SkippedIntervals);
    ErrorCheck(err, "LJM_WaitForNextInterval");
    if (SkippedIntervals > 0) {
        printf("SkippedIntervals: %d\n", SkippedIntervals);
    }
}

err = LJM_CleanInterval(INTERVAL_HANDLE);
ErrorCheck(err, "LJM_CleanInterval");
```

2.7.4 - WaitForNextInterval

Waits (blocks/sleeps) until the next interval occurs. If intervals are skipped, this function still waits until the next complete interval.

Syntax

```
LJM_ERROR_RETURN LJM_WaitForNextInterval(
    int IntervalHandle,
    int * SkippedIntervals)
```

Parameters

IntervalHandle [in]

The user-generated interval identifier. This can be anything.

SkippedIntervals [out]

The number of skipped intervals that occurred since the last time this function was called.

Returns

[LJM errorcodes](#) or 0 for no error.

Returns `LJME_INVALID_INTERVAL_HANDLE` if `IntervalHandle` was not set up using [LJM_StartInterval](#).

Examples

See [LJM_StartInterval](#) for an example.

2.7.5 - CleanInterval

Cleans/deallocates memory for the given `IntervalHandle`.

Syntax

```
LJM_ERROR_RETURN LJM_CleanInterval(
    int IntervalHandle)
```

Parameters

IntervalHandle [in]

The user-generated interval identifier. This can be anything.

Returns

[LJM errorcodes](#) or 0 for no error.

Returns `LJME_INVALID_INTERVAL_HANDLE` if `IntervalHandle` was not set up using [LJM_StartInterval](#).

Examples

See [LJM_StartInterval](#) for an example.

2.8 - Debugging Functions

Debugging Functions Overview

LJM contains debug logging functionality that is disabled by default.

Once enabled, LJM will write messages to the [LJM debug log file](#). These messages can contain error messages, warnings, raw packets, and other information.

LJM's debug logging can be enabled using one of two methods – via `ljm_startup_configs.json` or via `LJM_WriteLibraryConfigS`. Both methods (described below) require changing the following LJM configuration parameters:

- Change `LJM_DEBUG_LOG_MODE` to `LJM_DEBUG_LOG_MODE_CONTINUOUS` (equal to 2).
- Change `LJM_DEBUG_LOG_LEVEL` to one of the possible debug levels. Use `LJM_STREAM_PACKET` (equal to 1) if you want to ensure that all debug messages are output.
- Change `LJM_DEBUG_LOG_FILE_MAX_SIZE` to a large number like 123456789 if your program will be running for a long time.

Enabling via `ljm_startup_configs.json`

The file `ljm_startup_configs.json` contains the LJM configurations that are used by LJM when LJM is loaded.

Note that `ljm_startup_configs.json` is installed / replaced every time the LJM installer is run, so this method is temporary. LJM installs `ljm_startup_configs.json` to the following locations:

- Windows Vista and later - `C:\ProgramData\LabJack\LJM\ljm_startup_configs.json`
- Windows XP - `C:\Documents and Settings\All Users\Application Data\LabJack\LJM\ljm_startup_configs.json`
- Mac OS X and Linux - `/usr/local/share/LabJack/LJM/ljm_startup_configs.json`

After opening `ljm_startup_configs.json` with a text editor, change the parameters as described above.

Enabling via `LJM_WriteLibraryConfigS`

Use the function `LJM_WriteLibraryConfigS` to change the LJM parameters as described above.

2.8.1 - Log

Saves a message of the specified severity level to the [LJM debug log file](#).

Syntax

```
LJM_ERROR_RETURN LJM_Log(
    int Level,
    const char * String)
```

Parameters

Level [in]

The error severity level of this new log entry. See constants file `LJM_DEBUG_LOG_LEVEL`.

- `LJM_STREAM_PACKET = 1`
- `LJM_TRACE = 2`
- `LJM_DEBUG = 4`
- `LJM_INFO = 6`
- `LJM_PACKET = 7`
- `LJM_WARNING = 8`
- `LJM_USER = 9`
- `LJM_ERROR = 10`
- `LJM_FATAL = 12`

String [in]

The debug message to write to the log file.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

By default, `LJM_DEBUG_LOG_MODE` is to never log, so LJM does not output any log messages, even from this function. Users must first use `LJM_WriteLibraryConfigS` to change the log mode. See the library configuration functions.

Examples

[C/C++] Write "Beginning stream..." to the log file with severity level: Debug.

```
int LJLError;
LJLError = LJM_Log(4, "Beginning stream...");
```

2.8.2 - ResetLog

Clears all characters from the debug log file.

Syntax

```
LJM_ERROR_RETURN LJM_ResetLog()
```

Parameters

None.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

See the LJM configuration properties for Log-related properties.

Examples

[C/C++] Clear the log file.

```
int LJMErr;
LJMErr = LJM_ResetLog();
```

2.9 - Low-level Functions

2.9.1 - Feedback Functions

Summary

The low-level Feedback functions [LJM_AddressesToMBFB](#), [LJM_MBFBComm](#), and [LJM_UpdateValues](#) may be used together or separately to perform Modbus Feedback (MBFB) operations on the LabJack device.

- [LJM_AddressesToMBFB](#) creates a MBFB packet
- [LJM_MBFBComm](#) sends a MBFB packet to the device and receives a MBFB response packet
- [LJM_UpdateValues](#) converts read values in a MBFB response packet

Want Something Simpler?

For simpler functions that do the same thing as the low-level Feedback functions, see the [Single Value Functions](#) or the [Multiple Value Functions](#).

Examples

[C/C++] Write to DAC0, read from AIN0 using [LJM_AddressesToMBFB](#), [LJM_MBFBComm](#), and [LJM_UpdateValues](#)

```
/**
 * Name: stepwise_feedback.c
 * Desc: Shows how to read from a few analog inputs
 *       using a LabJack and the cross platform LabJackM
 *       Library using the C-style API
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <LabJackM.h>

#include "LJM_Uilities.h"

int main()
{
    int handle, err;

    // Just something that isn't positive so we know
    // errAddress has or hasn't been modified
    int errAddress = -2;

    // Set up the data
    enum { NUM_FRAMES = 2 };
    int numFrames = NUM_FRAMES;

    const char * ADDRESS_STRINGS[NUM_FRAMES] =
        {"DAC0", "AIN0"};
    const double VALUE_0 = 1.23;
```

```

const double VALUE_1 = 0.0;

int aAddresses[NUM_FRAMES];
int aTypes[NUM_FRAMES];
int aWrites[NUM_FRAMES] = {LJM_WRITE, LJM_READ };
int aNumValues[NUM_FRAMES] = {1, 1 };

enum { NUM_VALUES = 2 };
double aValues[NUM_VALUES] = {VALUE_0, VALUE_1 };

int MaxBytesPerMBFB = LJM_DEFAULT_FEEDBACK_ALLOCATION_SIZE;
unsigned char aMBFB[LJM_DEFAULT_FEEDBACK_ALLOCATION_SIZE];

unsigned char UnitID = LJM_DEFAULT_UNIT_ID;

// Fill out aAddresses
err = LJM_NamesToAddresses(NUM_FRAMES, ADDRESS_STRINGS,
    aAddresses, aTypes);
ErrorCheck(err, "LJM_NamesToAddresses");

// Open first found LabJack
err = LJM_Open(LJM_dtANY, LJM_ctANY, "LJM_idANY", &handle);
ErrorCheck(err, "LJM_Open");

PrintDeviceInfoFromHandle(handle);

err = LJM_AddressesToMBFB(MaxBytesPerMBFB, aAddresses,
    aTypes, aWrites, aNumValues, aValues, &numFrames,
    aMBFB);
ErrorCheck(err, "LJM_AddressesToMBFB");

printf("\nLJM_MBFBComm will overwrite the Transaction ID and Unit ID of the following command\n");
PrintFeedbackCommand(aMBFB, "Feedback command");

// Send the command and receive the response
err = LJM_MBFBComm(handle, UnitID, aMBFB, &errAddress);
ErrorCheckWithAddress(err, errAddress, "LJM_MBFBComm");

PrintFeedbackResponse(aMBFB, "Feedback response");

// Get the data back in a readable form
err = LJM_UpdateValues(aMBFB, aTypes, aWrites,
    aNumValues, numFrames, aValues);
ErrorCheck(err, "LJM_UpdateValues");

// Print results
printf("%s: %fn", ADDRESS_STRINGS[1], aValues[1]);

// Close
err = LJM_Close(handle);
ErrorCheck(err, "LJM_Close");

WaitForUserIfWindows();

return LJME_NOERROR;
}

```

Possible output:

```

deviceType: LJM_dtT7
connectionType: LJM_ctUSB
serialNumber: 470010117
pipe: 0
The maximum number of bytes you can send to or receive from this device in one packet is 64 bytes.

```

```

LJM_MBFBComm will overwrite the Transaction ID and Unit ID of the following command
Feedback command:
Header: 0x00 0x00 0x00 0x00 0x00 0x0e 0x01 0x4c
frame 00: 0x01 0x03 0xe8 0x02 0x3f 0x9d 0x70 0xa4
frame 01: 0x00 0x00 0x00 0x02
Feedback response:
Header: 0x01 0x89 0x00 0x00 0x00 0x06 0x01 0x4c
data: 0x40 0x9a 0xfd 0x5f
AINO: 4.843429

```

2.9.1.1 - AddressesToMBFB

Creates a Modbus Feedback (MBFB) packet. This packet can be sent to the device using [LJM_MBFBComm](#).

Syntax

```

LJM_ERROR_RETURN LJM_AddressesToMBFB(
    int MaxBytesPerMBFB,
    int * aAddresses,
    int * aTypes,
    int * aWrites,

```

```
int * aNumValues,
double * aValues,
int * NumFrames,
unsigned char * aMBFBCommand)
```

Parameters

MaxBytesPerMBFB [in]

The maximum number of bytes that the Feedback command is allowed to consist of. It is highly recommended to pass the size of the `aMBFBCommand` buffer as `MaxBytesPerMBFB` to prevent buffer overflow. See `LJM_DEFAULT_FEEDBACK_ALLOCATION_SIZE`.

aAddresses [in]

An array of size `NumFrames` representing the register addresses to read from or write to for each frame.

aTypes [in]

An array of size `NumFrames` containing the data type of each value sent/received for each frame.
Types - 0:UINT16, 1:UINT32, 2:INT32, 3:FLOAT32

aWrites [in]

An array of size `NumFrames` containing the desired type of access for each frame, which is either `read(0)` or `write(1)`.

aNumValues [in]

An array of size `NumFrames` that contains the number of consecutive values for each frame. Use 1 for a single (non-consecutive) value.

aValues [in]

An array of values to be transferred to/from the device. This array contains all the values for all the frames, so its size should be the sum of all elements in the `aNumValues` array. Each write value will be converted to its corresponding frame type defined in `aTypes`. `LJM_UpdateValues` can be used later to update this array with read values.

NumFrames [in/out]

Input as the number of frames being created. Output as the number of frames that were successfully put into the `aMBFBCommand` buffer, which may be less or equal to the input number. If `NumFrames` is altered by this function, `LJM_AddressesToMBFB` will return the warning code `LJME_FRAMES_OMITTED_DUE_TO_PACKET_SIZE`.

aMBFBCommand [out]

The resultant Modbus Feedback command. Transaction ID and Unit ID will be blanks that `LJM_MBFBComm` will fill in.

Returns

[LJM_errorcodes](#) or 0 for no error.

Remarks

The [LJM_NamesToAddresses](#) and [LJM_NameToAddress](#) functions may be used to initialize the `aAddresses` and `aTypes` parameters from register names.

Examples

Please see the [Low-level Feedback Functions overview page](#) for an example.

2.9.1.2 - MBFBComm

Sends a Feedback command and receives a Feedback response, parsing the response for obvious errors. The Feedback command may be generated using [LJM_AddressesToMBFB](#) and the Feedback response may be parsed with the [LJM_UpdateValues](#) function.

Syntax

```
LJM_ERROR_RETURN LJM_MBFBComm(
    int Handle,
    unsigned char UnitID,
    unsigned char * aMBFB,
    int * ErrorAddress)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

UnitID [in]

The ID of the specific unit that the Feedback command should be sent to. Use `LJM_DEFAULT_UNIT_ID` (1) unless the device documentation instructs otherwise.

aMBFB [in/out]

As an input parameter, it is a valid Feedback command, which may be generated using [LJM_AddressesToMBFB](#).

As an output parameter, it is a Feedback response, which may be an error response.

ErrorAddress [out]

If error, the address responsible for causing an error.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

LJM_MBFBComm provides device synchronization within LJM.

Examples

Please see the [Low-level Feedback Functions overview page](#) for an example.

2.9.1.3 - UpdateValues

Converts read values in a Modbus Feedback response packet.

Syntax

```
LJM_ERROR_RETURN LJM_UpdateValues(
    unsigned char * aMBFBResponse,
    const int * aTypes,
    const int * aWrites,
    const int * aNumValues,
    int NumFrames,
    double * aValues)
```

Parameters

Note that some of these parameters may be used directly from [LJM_AddressesToMBFB](#) and/or [LJM_MBFBComm](#).

aMBFBResponse [in]

A valid Feedback response. May be used directly from LJM_MBFBComm.

aTypes [in]

An array of size NumFrames containing the data type of each value sent/received for each frame. May be used directly from LJM_AddressesToMBFB.
Types - 0:UINT16, 1:UINT32, 2:INT32, 3:FLOAT32

aWrites [in]

An array of size NumFrames containing the desired type of access for each frame, which is either read(0) or write(1). May be used directly from LJM_AddressesToMBFB.

aNumValues [in]

An array of size NumFrames that contains the number of consecutive values for each frame. Use 1 for a single (non-consecutive) value. May be used directly from LJM_AddressesToMBFB.

NumFrames [in]

The number of frames. May be used from LJM_AddressesToMBFB after dereferencing.

aValues [out]

An array that contains all the values for all the frames, so its size should be the sum of all elements in the aNumValues array. Each read value is converted to its corresponding frame type defined in aTypes.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

The [Type Conversion functions](#) may also be used to read bytes to types.

Examples

Please see the [Low-level Feedback Functions overview page](#) for an example.

2.9.2 - Raw Byte Functions

Raw Byte Functions Overview

These functions do not provide full device synchronization within LJM. They are provided for advanced users familiar with Modbus that want to control the exact bytes sent/received.

2.9.2.1 - ReadRaw

Attempts to receive unaltered bytes from a device.

Syntax

```
LJM_ERROR_RETURN LJM_ReadRaw(
    int Handle,
    unsigned char * Data,
    int NumBytes)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Data [Out]

Buffer to be filled with bytes from the read

NumBytes [out]

The number of bytes that were read, or -1 for error.

Returns

[LJM_errorcode](#) or 0 for no error.

Remarks

Must be used in conjunction with [LJM_WriteRaw](#).

2.9.2.2 - WriteRaw

Attempts to send unaltered bytes to a device.

Syntax

```
LJM_ERROR_RETURN LJM_WriteRaw(
    int Handle,
    const unsigned char * Data,
    int NumBytes)
```

Parameters

Handle [in]

A device handle. The handle is a connection ID for an active device. Generate a handle with [LJM_Open](#) or [LJM_OpenS](#).

Data [in]

Buffer of bytes to send to the device.

NumBytes [in]

The number of bytes in Data.

Returns

[LJM_errorcode](#) or 0 for no error.

Remarks

Must be used in conjunction with [LJM_ReadRaw](#).

2.9.3 - Type Conversion Functions

Type Conversion Functions Overview

Byte/Value Conversion Functions

These functions are not needed except for applications performing raw bytes manipulation, as with the [Raw Byte functions](#).

Byte-to-value

- [LJM_ByteArrayToFLOAT32](#)
- [LJM_ByteArrayToINT32](#)
- [LJM_ByteArrayToUINT32](#)
- [LJM_ByteArrayToUINT16](#)

Value-to-byte

- [LJM_FLOAT32ToByteArray](#)
- [LJM_INT32ToByteArray](#)
- [LJM_UINT32ToByteArray](#)
- [LJM_UINT16ToByteArray](#)

2.9.3.1 - ByteArrayToFLOAT32

Converts an array of bytes to 32-bit float values, performing automatic endian conversions if necessary.

This function is not needed except for applications performing raw byte manipulation, as with the [Raw Byte functions](#).

Syntax

```
LJM_ERROR_RETURN LJM_ByteArrayToFLOAT32(
    const unsigned char * aBytes,
    int RegisterOffset,
    int NumFLOAT32,
    float * aFLOAT32)
```

Parameters

aBytes [in]

The bytes to be converted.

RegisterOffset [in]

The register offset to get the values from in *aBytes*.

NumFLOAT32 [in]

The number of values to convert.

aFLOAT32 [out]

The converted bytes in 32-bit float value form.

Returns

[LJM errorcode](#) or 0 for no error.

Remarks

See also: [LJM_FLOAT32ToByteArray](#)

2.9.3.2 - ByteArrayToINT32

Converts an array of bytes to 32-bit integer values, performing automatic endian conversions if necessary.

This function is not needed except for applications performing raw byte manipulation, as with the [Raw Byte functions](#).

Syntax

```
LJM_ERROR_RETURN LJM_ByteArrayToINT32(
    const unsigned char * aBytes,
    int RegisterOffset,
    int NumINT32,
    int * aINT32)
```

Parameters

aBytes [in]

The bytes to be converted.

RegisterOffset [in]

The register offset to get the values from in *aBytes*.

NumINT32 [in]

The number of values to convert.

aINT32 [out]

The converted bytes in 32-bit integer value form.

Returns

[LJM errorcode](#) or 0 for no error.

Remarks

See also: [LJM_INT32ToByteArray](#)

2.9.3.3 - ByteArrayToUINT16

Converts an array of bytes to 16-bit unsigned integer values, performing automatic endian conversions if necessary.

This function is not needed except for applications performing raw byte manipulation, as with the [Raw Byte functions](#).

Syntax

```
LJM_ERROR_RETURN LJM_ByteArrayToUINT16(
    const unsigned char * aBytes,
    int RegisterOffset,
    int NumUINT16,
    unsigned int * aUINT16)
```

Parameters

aBytes [in]

The bytes to be converted.

RegisterOffset [in]

The register offset to get the values from in *aBytes*.

NumUINT16 [in]

The number of values to convert.

aUINT16 [out]

The converted bytes in 16-bit unsigned integer value form.

Returns

[LJM errorcode](#) or 0 for no error.

Remarks

See also: [LJM_UINT16ToByteArray](#)

2.9.3.4 - ByteArrayToUINT32

Converts an array of bytes to 32-bit unsigned integer values, performing automatic endian conversions if necessary.

This function is not needed except for applications performing raw byte manipulation, as with the [Raw Byte functions](#).

Syntax

```
LJM_ERROR_RETURN LJM_ByteArrayToUINT32(
    const unsigned char * aBytes,
    int RegisterOffset,
    int NumUINT32,
    unsigned int * aUINT32)
```

Parameters

aBytes [in]

The bytes to be converted.

RegisterOffset [in]

The register offset to get the values from in *aBytes*.

NumUINT32 [in]

The number of values to convert.

aUINT32 [out]

The converted bytes in 32-bit unsigned integer value form.

Returns

[LJM_errorcode](#) or 0 for no error.

Remarks

See also: [LJM_UINT32ToByteArray](#)

2.9.3.5 - FLOAT32ToByteArray

Converts an array of 32-bit float values to bytes, performing automatic endian conversions if necessary.

This function is not needed except for applications performing raw byte manipulation, as with the [Raw Byte functions](#).

Syntax

```
LJM_ERROR_RETURN LJM_FLOAT32ToByteArray(
    const float * aFLOAT32,
    int RegisterOffset,
    int NumFLOAT32,
    unsigned char * aBytes)
```

Parameters

aFLOAT32 [in]

The array of values to be converted to bytes.

RegisterOffset [in]

The register offset to put the converted values in aBytes.

NumFLOAT32 [in]

The number of values to convert.

aBytes [out]

The converted values in byte form.

Returns

[LJM_errorcode](#) or 0 for no error.

Remarks

See also: [LJM_ByteArrayToFLOAT32](#)

2.9.3.6 - INT32ToByteArray

Converts an array of 32-bit integer values to bytes, performing automatic endian conversions if necessary.

This function is not needed except for applications performing raw byte manipulation, as with the [Raw Byte functions](#).

Syntax

```
LJM_ERROR_RETURN LJM_INT32ToByteArray(
    const int * aINT32,
    int RegisterOffset,
    int NumINT32,
    unsigned char * aBytes)
```

Parameters

aINT32 [in]

The array of values to be converted to bytes.

RegisterOffset [in]

The register offset to put the converted values in aBytes.

NumINT32 [in]

The number of values to convert.

aBytes [out]

The converted values in byte form.

Returns

[LJM_errorcode](#) or 0 for no error.

Remarks

See also: [LJM_ByteArrayToINT32](#)

2.9.3.7 - UINT16ToByteArray

Converts an array of 16-bit unsigned integer values to bytes, performing automatic endian conversions if necessary.

This function is not needed except for applications performing raw byte manipulation, as with the [Raw Byte functions](#).

Syntax

```
LJM_ERROR_RETURN LJM_UINT16ToByteArray(
    const unsigned int * aUINT16,
    int RegisterOffset,
    int NumUINT16,
    unsigned char * aBytes)
```

Parameters

aUINT16 [in]

The array of values to be converted to bytes.

RegisterOffset [in]

The register offset to put the converted values in *aBytes*.

NumUINT16 [in]

The number of values to convert.

aBytes [out]

The converted values in byte form.

Returns

[LJM_errorcode](#) or 0 for no error.

Remarks

See also: [LJM_ByteArrayToUINT16](#)

2.9.3.8 - UINT32ToByteArray

Converts an array of 32-bit unsigned int values to bytes, performing automatic endian conversions if necessary.

This function is not needed except for applications performing raw byte manipulation, as with the [Raw Byte functions](#).

Syntax

```
LJM_ERROR_RETURN LJM_UINT32ToByteArray(
    const unsigned int * aUINT32,
    int RegisterOffset,
    int NumUINT32,
    unsigned char * aBytes)
```

Parameters

aUINT32 [in]

The array of values to be converted to bytes.

RegisterOffset [in]

The register offset to put the converted values in *aBytes*.

NumUINT32 [in]

The number of values to convert.

aBytes [out]

The converted values in byte form.

Returns

[LJM_errorcode](#) or 0 for no error.

Remarks

See also: [LJM_ByteArrayToUINT32](#)

2.10 - Library Configuration Functions

Library Configuration Functions Overview

This section describes how to configure the LJM library's behavior and interpretation constants. Whenever LJM is started up, it is loaded with default values, so any desired configurations must be applied each time LJM is started.

Behavioral Parameters

The functions `LJM_WriteLibraryConfigS` and `LJM_WriteLibraryConfigStringS` are good for setting a few configurations, while `LJM_LoadConfigurationFile` is more convenient for setting many configurations or for sharing a configuration between programs.

Interpretation Constants

LJM also provides interpretation of Modbus map constants and error constants.

The Modbus map constants affect Name functions that use Name interpretation, such as `LJM_eReadName`. These Name functions have "Name" in their name and have a "const char * Name" parameter or "const char ** aNames" parameter. For each name passed to one of these functions, it is interpreted according to the loaded constants. The Modbus map constants affect the behavior of other functions as well, such as when `LJM_OLD_FIRMWARE_CHECK` is enabled.

The Error constants affects the `LJM_ErrorToString` function. The loaded Error constants will be interpreted from an integer error code to a error string. Custom application error codes can be loaded in the 3900-3999 range.

Configuration Parameters

The following parameters are constants defined in the header file `LabJackM.h`. Each parameter's name is the same as it's value; e.g. `LJM_LIBRARY_VERSION` is a constant of the string "LJM_LIBRARY_VERSION".

Informational parameters:

- `LJM_LIBRARY_VERSION` - The current LJM version. (Read-only)

Timeout parameters:

- `LJM_SEND_RECEIVE_TIMEOUT_MS` - How long LJM waits for a packet to be sent or received.
- `LJM_OPEN_TCP_DEVICE_TIMEOUT_MS` - How long LJM waits for a device to respond to a connection request.

Device operation parameters:

- `LJM_ALLOWS_AUTO_MULTIPLE_FEEDBACKS` - Whether or not LJM will automatically perform multiple Feedback commands when the desired operations would exceed the maximum packet length.
- `LJM_ALLOWS_AUTO_CONDENSE_ADDRESSES` - Whether or not LJM will automatically condense congruent address reads/writes into array reads/writes.
- `LJM_RETRY_ON_TRANSACTION_ID_MISMATCH` - Whether or not LJM automatically retries an operation if an `LJME_TRANSACTION_ID_ERR` occurs.
- `LJM_OLD_FIRMWARE_CHECK` - Whether or not LJM will check the Modbus constants file (see `LJM_MODBUS_MAP_CONSTANTS_FILE`) to make sure the firmware of the current device is compatible with the Modbus register(s) being read from or written to.

Constants file parameters:

- `LJM_MODBUS_MAP_CONSTANTS_FILE` - The absolute or relative path of the constants file to use for that use the LJM Name functionality.
- `LJM_ERROR_CONSTANTS_FILE` - The absolute or relative path of the constants file to use for `LJM_ErrorToString`.
- `LJM_CONSTANTS_FILE` - For setting both `LJM_MODBUS_MAP_CONSTANTS_FILE` and `LJM_ERROR_CONSTANTS_FILE`. (Write-only)

Debug logger parameters:

- `LJM_DEBUG_LOG_FILE` - The absolute or relative path of the file to output log messages to.
- `LJM_DEBUG_LOG_MODE` - What conditions will cause LJM to output log messages.
- `LJM_DEBUG_LOG_LEVEL` - What priority of log messages are output.
- `LJM_DEBUG_LOG_BUFFER_MAX_SIZE` - How many messages the log buffer can hold.
- `LJM_DEBUG_LOG_SLEEP_TIME_MS` - How often the logger thread will run.
- `LJM_DEBUG_LOG_FILE_MAX_SIZE` - The maximum size of the log file in number of characters.

Stream parameters:

- `LJM_STREAM_RECEIVE_TIMEOUT_MODE` - Sets how stream data collection should time out.
- `LJM_STREAM_RECEIVE_TIMEOUT_MS` - Manually sets the timeout for LJM's stream data collection.
- `LJM_STREAM_SCANS_RETURN` - How `LJM_eStreamRead` behaves when the full read amount is not ready.
- `LJM_STREAM_TRANSFERS_PER_SECOND` - How often stream threads attempt to read from the stream.

Altering Configurations

Functions for setting the behavioral configurations of LJM:

- [LJM_WriteLibraryConfigS](#) - Set one numerical configuration value.
- [LJM_WriteLibraryConfigStringS](#) - Set one string-based configuration value.
- [LJM_LoadConfigurationFile](#) - Load all the configuration values in a specified file.

Altering Modbus Map Constants or Error Constants

Functions for setting the Modbus and/or Error interpretation maps:

- [LJM_LoadConstants](#) - Manually loads or reloads the constants file.
- [LJM_LoadConstantsFromFile](#) - Loads a specified file as the Modbus map and Error constants.
- [LJM_LoadConstantsFromString](#) - Loads a string as the Modbus map and/or Error constants.

Reading Configurations

Functions for getting the behavioral configurations of LJM:

- [LJM_ReadLibraryConfigS](#) - Read one numerical configuration value.
- [LJM_ReadLibraryConfigStringS](#) - Read one string-based configuration value.

2.10.1 - GetSpecificIPsInfo

Get information about whether the [specific IPs](#) file was parsed successfully.

Syntax

```
LJM_ERROR_RETURN LJM_GetSpecificIPsInfo(
    int * InfoHandle,
    const char ** Info)
```

Parameters

InfoHandle [out]

A handle to Info that should be passed to [LJM_CleanInfo](#) after Info has been read.

Info [out]

A pointer to a JSON char * (allocated by LJM) describing the state of the specific IPs.

Info Semantics

```
{
  "errorCode": Integer LJME_ error code. 0 indicates no error
  "IPs": Array of strings - the presentation-format IPs
  "message": Human-readable string description of success/failure
  "filePath": String absolute or relative file path
  "invalidIPs": Array of strings - the unparsable lines
}
```

Returns

[LJM_errorcodes](#) or 0 for no error. This may be [LJME_CONFIG_PARSING_ERROR](#) even if some addresses in the specific IP file were parsed without error.

2.10.2 - LoadConfigurationFile

Load all the configuration values in a specified file.

For the full list of possible parameters, see [Configuration Parameters](#). Alternately, see the [ljm_startup_configs.json](#) file that was installed along with LJM.

[LJM_LoadConfigurationFile](#) parses files according to the semantics of [ljm_startup_configs.json](#). [ljm_startup_configs.json](#) can be found in the default constants file location (see [Constants file parameters](#)).

Syntax

```
LJM_ERROR_RETURN LJM_LoadConfigurationFile(
    const char * FileName)
```

Parameters

FileName [in]

A relative or absolute file location. Must null-terminate. "default" maps to the default configuration file [ljm_startup_config.json](#) in the constants file location (see [Constants file parameters](#)).

Returns

[LJM_errorcodes](#) or 0 for no error.

LJME_CONFIG_FILE_NOT_FOUND (1289): FileName did not give the location to a readable file.

LJME_CONFIG_PARSING_ERROR (1290): The file at location FileName did not contain a valid configuration file.

Remarks

To write a numerical configuration parameter, see [LJM_WriteLibraryConfigS](#).

To write a string-based configuration parameter, see [LJM_WriteLibraryConfigStringS](#).

Example

[C/C++] Load the configuration file "ljm_verbose_debug_log_configs.json"

```
int LJMEError = LJM_LoadConfigurationFile("ljm_verbose_debug_log_configs.json");
```

2.10.3 - LoadConstants

Manually loads or reloads the constants file.

This step is automatic. This function does not need to be called before any functions that use LJM's Modbus interpretation or Error interpretation. LJM_LoadConstantsFromFile is provided so that changes to the constants file can be loaded.

Syntax

```
LJM_VOID_RETURN LJM_LoadConstants()
```

Returns

Nothing.

Remarks

To specify a constants file to load, see [LJM_LoadConstantsFromFile](#).

2.10.4 - LoadConstantsFromFile

Loads a specified file as the Modbus map and Error constants.

Syntax

```
LJM_ERROR_RETURN LJM_LoadConstantsFromFile(
    const char * FileName)
```

Parameters

FileName [in]

The absolute or relative file path to load as the constants. Must null-terminate. "default" maps to the default configuration file ljm_startup_config.json in the constants file location (see [Constants file parameters](#)).

Returns

[LJM_errorcodes](#) or 0 for no error.

LJME_CONSTANTS_FILE_NOT_FOUND (1292): FileName did not point to a readable file location.

LJME_INVALID_CONSTANTS_FILE (1293): The file located at FileName was not a valid constants file. LJM_ErrorToString outputs a specific error message.

Remarks

LJM_LoadConstantsFromFile is an alias for executing [LJM_WriteLibraryConfigStringS](#) with LJME_CONSTANTS_FILE as the Parameter of [LJM_WriteLibraryConfigStringS](#) and FileName as the String parameter of [LJM_WriteLibraryConfigStringS](#):

```
LJM_WriteLibraryConfigStringS(LJME_CONSTANTS_FILE, FileName)
```

To load the Modbus constants and error constants separately, use [LJM_WriteLibraryConfigStringS](#) with LJM_MODBUS_MAP_CONSTANTS_FILE or LJME_ERROR_CONSTANTS_FILE as the Parameter.

Example

[C/C++] Load the constants file "custom_modbus_map.json"

```
char ErrorMessage[LJM_MAX_NAME_SIZE];
int LJMEError;
LJMEError = LJM_LoadConstantsFromFile("custom_modbus_map.json");
```

```

if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_LoadConstantsFromFile error: %s\n", ErrorString);
}

```

2.10.5 - LoadConstantsFromString

Loads a string as the Modbus map and/or Error constants.

Syntax

```

LJM_ERROR_RETURN LJM_LoadConstantsFromString(
    const char * JsonString)

```

Parameters

JsonString [in]

A string in JSON format containing the constants to load. Must null-terminate. If JsonString contains a JSON array named "registers", that array is loaded as the new Modbus map constants and the previous Modbus map constants are unloaded. If JsonString contains a JSON array named "errors", that array is loaded as the new Error constants and the previous Error constants are unloaded.

Returns

[LJM_errorcodes](#) or 0 for no error.

LJM_INVALID_CONSTANTS_FILE (1293): The constants in JsonString was not a valid constants file. LJM_ErrorToString outputs a specific error message.

Remarks

To load constants from a file, see [LJM_LoadConstantsFromFile](#).

Example

[C/C++] Load a small constants string

```

char ErrorString[LJM_MAX_NAME_SIZE];
int LJMErr = LJM_LoadConstantsFromString(
    "{
    \"registers\":
    [
    {\"address\":600, \"name\": \"STRING_AIN#(0:254)\", \"type\": \"FLOAT32\",
    \"devices\": [{\"device\": \"T7\", \"fwmin\": 1.0001}],
    \"readwrite\": \"R\", \"tags\": [\"AIN\"], \"altnames\": [\"\"],
    {\"address\":4321, \"name\": \"STRING_SERIAL_NUMBER\", \"type\": \"UINT32\",
    \"devices\": [{\"device\": \"T7\", \"fwmin\": 1.0001}],
    \"readwrite\": \"R\", \"tags\": [\"CONFIG\"], \"altnames\": [\"\"]}
    ],
    \"errors\":
    [
    {\"error\":0, \"string\": \"SUCCESS\"},
    {\"error\":3900, \"string\": \"UNKNOWN_APPLICATION_ERROR\"}
    ]
    }
");
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_LoadConstantsFromString error: %s\n", ErrorString);
}

```

2.10.6 - ReadLibraryConfigs

Read one numerical configuration value.

Syntax

```

LJM_ERROR_RETURN LJM_ReadLibraryConfigS(
    const char * Parameter,
    double * Value)

```

Parameters

Parameter [in]

The name of the configuration setting, see configuration parameters page. Not case-sensitive. Must null-terminate. For the full list of possible parameters, see [Configuration Parameters](#).

Value [out]

The return value representing the current value of Parameter.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

To write a numerical configuration parameter, see [LJM_WriteLibraryConfigS](#).

To read a string-based configuration parameter, see [LJM_ReadLibraryConfigStringS](#).

Example

[C/C++] Read LJM_SEND_RECEIVE_TIMEOUT_MS

```

char ErrorString[LJM_MAX_NAME_SIZE];
double Value;
int LJMErr = LJM_ReadLibraryConfigS(LJM_SEND_RECEIVE_TIMEOUT_MS, &Value);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
printf("LJM_SEND_RECEIVE_TIMEOUT_MS is currently %fn", Value);

```

2.10.7 - ReadLibraryConfigStringS

Read one string-based configuration value.

Syntax

```

LJM_ERROR_RETURN LJM_ReadLibraryConfigStringS(
    const char * Parameter,
    double * String)

```

Parameters

Parameter [in]

The name of the configuration setting, see configuration parameters page. Not case-sensitive. Must null-terminate. For the full list of possible parameters, see [Configuration Parameters](#).

String [out]

The return value string representing the current value of Parameter. Must be allocated to size LJM_MAX_NAME_SIZE.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

To write a string-based configuration parameter, see [LJM_WriteLibraryConfigStringS](#).

To read a numerical configuration parameter, see [LJM_ReadLibraryConfigS](#).

Example

[C/C++] Read LJM_DEBUG_LOG_FILE

```

char ErrorString[LJM_MAX_NAME_SIZE];
char String[LJM_MAX_NAME_SIZE];
int LJMErr = LJM_ReadLibraryConfigStringS(LJM_DEBUG_LOG_FILE, String);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigStringS error: %s\n", ErrorString);
}
printf("LJM_DEBUG_LOG_FILE is currently %s\n", String);

```

2.10.8 - WriteLibraryConfigS

Write a numerical LJM behavioral configuration value.

Syntax

```

LJM_ERROR_RETURN LJM_WriteLibraryConfigS(
    const char * Parameter,
    double Value)

```

Parameters

Parameter [in]

The name of the configuration setting, see configuration parameters page. Not case-sensitive. Must null-terminate. For the full list of possible parameters, see [Configuration Parameters](#).

Value [in]

The config value to apply to Parameter.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

To write a string-based configuration parameter, see [LJM_WriteLibraryConfigStringS](#).

To load all the configuration values in a specified file, see [LJM_LoadConfigurationFile](#).

Example

[C/C++] Set LJM_SEND_RECEIVE_TIMEOUT_MS to 500 milliseconds

```
int LJMError = LJM_WriteLibraryConfigS(LJM_SEND_RECEIVE_TIMEOUT_MS, 500);
```

2.10.9 - WriteLibraryConfigStringS

Write a string-based LJM behavioral configuration value.

Syntax

```
LJM_ERROR_RETURN LJM_WriteLibraryConfigStringS(
    const char * Parameter,
    const char * String)
```

Parameters**Parameter** [in]

The name of the configuration setting, see the configuration parameters page. Not case-sensitive. Must null-terminate. For the full list of possible parameters, see [Configuration Parameters](#).

String [in]

The config value string to apply to Parameter. Must null-terminate. Must be of size LJM_MAX_NAME_SIZE or less, including the null-terminator.

Returns

[LJM errorcodes](#) or 0 for no error.

Remarks

To write a numerical configuration parameter, see [LJM_WriteLibraryConfigS](#).

To load all the configuration values in a specified file, see [LJM_LoadConfigurationFile](#).

Example

[C/C++] Set LJM_DEBUG_LOG_FILE to "new_log_file.log"

```
int LJMError = LJM_WriteLibraryConfigStringS(LJM_DEBUG_LOG_FILE, "new_log_file.log");
```

2.10.10 - GetDeepSearchInfo

Get information about whether the [Deep Search](#) file was parsed successfully.

Syntax

```
LJM_ERROR_RETURN LJM_GetDeepSearchInfo(
    int * InfoHandle,
    const char ** Info)
```

Parameters**InfoHandle** [out]

A handle to Info that should be passed to [LJM_CleanInfo](#) after Info has been read.

Info [out]

A pointer to a JSON char * (allocated by LJM) describing the state of the Deep Search.

Info Semantics

```
{
  "errorCode": Integer LJME_ error code. 0 indicates no error
  "IPs": Array of strings - the presentation-format IPs
  "message": Human-readable string description of success/failure
  "filePath": String absolute or relative file path
  "invalidIPs": Array of strings - the unparseable lines
}
```

Returns

[LJM_errorcodes](#) or 0 for no error. This may be LJME_CONFIG_PARSING_ERROR even if some addresses in the Deep Search file were parsed without error.

Easy Functions

Documentation has moved. Please see:

- [Single Value Functions](#)
- [Multiple Value Functions](#)
- [Stream Functions](#)

Summary

This section describes how to obtain data from a device using the Easy functions. [Modbus registers](#) have both a name and an address, so the Easy functions are categorized accordingly. For instance, analog input 3 has an address of 6, and is named "AIN3". Most people will find the name functions quite convenient.

Additionally, there are single and multiple value access functions. The single value functions access a single data value, and the multiple value functions access an array of data values. The advantage to using the multiple value functions is that the number of function calls can be reduced, and overall communication traffic is more efficient.

The many variants of Easy functions are provided to afford users maximum flexibility. It should be noted that nearly all device functionality can be achieved with only 2 functions: [LJM_eReadName](#), and [LJM_eWriteName](#), so start with these functions.

Address Functions

Address functions access data through an address, such as 10 for analog input 5.

- [LJM_eWriteAddress](#) - Write one value specified by address.
- [LJM_eReadAddress](#) - Read one value specified by address.
- [LJM_eWriteAddresses](#) - Write values specified by addresses.
- [LJM_eReadAddresses](#) - Read values specified by addresses.
- [LJM_eWriteAddressArray](#) - Write consecutive values specified by address.
- [LJM_eReadAddressArray](#) - Read consecutive values specified by address.
- [LJM_eAddresses](#) - Write/Read values specified by addresses and array sizes.

Name Functions

Name functions access data through a name, such as "AIN5" for analog input 5.

- [LJM_eWriteName](#) - Write one value specified by name.
- [LJM_eReadName](#) - Read one value specified by name.
- [LJM_eWriteNames](#) - Write values specified by names.
- [LJM_eReadNames](#) - Read values specified by names.
- [LJM_eWriteNameArray](#) - Write consecutive values specified by name.
- [LJM_eReadNameArray](#) - Read consecutive values specified by name.
- [LJM_eNames](#) - Write/Read values specified by names and array sizes.

Choosing a function to use can depend on the programming environment, some languages use strings easily, so the name parameters might be convenient, etc.

Remarks

All of the Easy functions provide device synchronization within LJM.

3 - Constants

A note on language wrapper differences

The constants are accessed in different ways, depending on which [LJM language wrapper](#) is being used.

C/C++: LJM constants begin with the prefix `LJM_`. For example, use `LJM_READ`.

LabVIEW: LJM constants begin with the prefix `LJM_`. For example, use `LJM_READ`.

Python: LJM constants are located in the namespace `labjack.ljm.constants` and do not begin with the `LJM_` prefix. For example:

```
import labjack
print(labjack.ljm.constants.READ) # Prints 0
```

Or, for a shorter version:

```
from labjack.ljm import constants as ljmc
print(ljmc.READ) # Prints 0
```

.NET: LJM constants are located in the namespace `LabJack.LJM.CONSTANTS` and do not begin with the `LJM_` prefix. For example:

```
// C#
import LabJack;
// ...
Console.WriteLine(LJM.CONSTANTS.READ); // Prints 0

' Visual Basic
Imports LabJack
' ...
Console.WriteLine(LJM.CONSTANTS.READ) ' Prints 0
```

Delphi: LJM constants begin with the prefix `LJM_`. For example, use `LJM_READ`.

Java: LJM constants are located in the namespace `com.labjack.LJM.Constants` and do not begin with the `LJM_` prefix. For example:

```
import com.labjack.LJM;
// ...
System.out.println(LJM.Constants.READ); // Prints 0
```

Node.js: LJM constants are located in [ljswitchboard-ljm_driver_constants/lib/driver_constants.js](#) and generally begin with the prefix `LJM_`. For example,

```
var ljmc = require('ljswitchboard-ljm_driver_constants');
console.log(ljmc.LJM_READ)
```

Visual Basic 6 and VBA: LJM constants begin with the prefix `LJM_`. For example, use `LJM_READ`.

LJM Constants

Used to specify if reading or writing:

```
READ = 0
WRITE = 1
```

Used to specify data type for parameters:

```
UINT16 = 0
UINT32 = 1
INT32 = 2
FLOAT32 = 3
```

```
BYTE = 99
STRING = 98
```

The `STRING` data type has the following max size, not including the automatic null-terminator:

```
STRING_MAX_SIZE = 49
```

May be used to initialize the `ErrorAddress` parameter of the [multiple value functions](#):

```
INVALID_NAME_ADDRESS = -1
```

The maximum size of the `ErrorString` of [LJM_ErrorToString](#):

```
MAX_NAME_SIZE = 256
```

As an entry of the `aData` parameter of [LJM_eStreamRead](#), indicates a value was skipped due to auto-recovery:

```
DUMMY_VALUE = -9999
```

Device types used to specify which LabJack device to open:

```
dtANY = 0
dtUE9 = 9
dtU3 = 3
dtU6 = 6
dtT4 = 4
dtT7 = 7
```

```
dtTSERIES = 84
dtSKYMOTE_BRIDGE = 1000
dtDIGIT = 200
```

Common connection types used to specify connect to be used when opening a device:

```
ctANY = 0
ctUSB = 1
ctTCP = 2 // Ethernet or WiFi
ctETHERNET = 3 // TCP
ctWIFI = 4 // TCP
```

Uncommon connection types:

```
ctNETWORK_UDP = 5
ctETHERNET_UDP = 6
ctWIFI_UDP = 7
ctNETWORK_ANY = 8 // TCP or UDP
ctETHERNET_ANY = 9 // TCP or UDP
ctWIFI_ANY = 10 // TCP or UDP
```

Constants related to opening devices:

```
NO_IP_ADDRESS = 0
NO_PORT = 0
TCP_PORT = 502
ETHERNET_UDP_PORT = 52362
WIFI_UDP_PORT = 502
UNKNOWN_IP_ADDRESS = -1
DEMO_MODE = "-2"
idANY = 0
```

The size of the arrays of LJM_ListAll:

```
LIST_ALL_SIZE = 128
```

Maximum packet sizes:

- Max USB packet bytes = 64
- Max TCP packet bytes = 1400

Timeout constants:

```
NO_TIMEOUT = 0
DEFAULT_USB_SEND_RECEIVE_TIMEOUT_MS = 2600
DEFAULT_ETHERNET_OPEN_TIMEOUT_MS = 1000
DEFAULT_ETHERNET_SEND_RECEIVE_TIMEOUT_MS = 2600
DEFAULT_WIFI_OPEN_TIMEOUT_MS = 1000
DEFAULT_WIFI_SEND_RECEIVE_TIMEOUT_MS = 4000
```

LJM Configuration Constants

The LJM configuration constants configure LJM behavior. For more information, see the [Library Configuration Functions](#).

```
SEND_RECEIVE_TIMEOUT_MS = "LJM_SEND_RECEIVE_TIMEOUT_MS"
```

```
OPEN_TCP_DEVICE_TIMEOUT_MS = "LJM_OPEN_TCP_DEVICE_TIMEOUT_MS"
```

```
DEBUG_LOG_MODE = "LJM_DEBUG_LOG_MODE"
```

```
LJM_DEBUG_LOG_MODE_NEVER = 1.0
LJM_DEBUG_LOG_MODE_CONTINUOUS = 2.0
LJM_DEBUG_LOG_MODE_ON_ERROR = 3.0
```

```
DEBUG_LOG_LEVEL = "LJM_DEBUG_LOG_LEVEL"
```

```
LJM_STREAM_PACKET = 1.0
LJM_TRACE = 2.0
LJM_DEBUG = 4.0
LJM_INFO = 6.0
LJM_PACKET = 7.0
LJM_WARNING = 8.0
LJM_USER = 9.0
LJM_ERROR = 10.0
LJM_FATAL = 12.0
```

```
DEBUG_LOG_BUFFER_MAX_SIZE = "LJM_DEBUG_LOG_BUFFER_MAX_SIZE"
```

```
DEBUG_LOG_SLEEP_TIME_MS = "LJM_DEBUG_LOG_SLEEP_TIME_MS"
```

```
LIBRARY_VERSION = "LJM_LIBRARY_VERSION"
```

```
ALLOWS_AUTO_MULTIPLE_FEEDBACKS = "LJM_ALLOWS_AUTO_MULTIPLE_FEEDBACKS"
```

```
ALLOWS_AUTO_CONDENSE_ADDRESSES = "LJM_ALLOWS_AUTO_CONDENSE_ADDRESSES"
```

```
LJM_AUTO_IPS = "LJM_AUTO_IPS"
```

LJM_AUTO_IPS_FILE = "LJM_AUTO_IPS_FILE"
AUTO_RECONNECT_STICKY_CONNECTION = "LJM_AUTO_RECONNECT_STICKY_CONNECTION"
AUTO_RECONNECT_STICKY_SERIAL = "LJM_AUTO_RECONNECT_STICKY_SERIAL"
AUTO_RECONNECT_WAIT_MS = "LJM_AUTO_RECONNECT_WAIT_MS"
MODBUS_MAP_CONSTANTS_FILE = "LJM_MODBUS_MAP_CONSTANTS_FILE"
ERROR_CONSTANTS_FILE = "LJM_ERROR_CONSTANTS_FILE"
DEBUG_LOG_FILE = "LJM_DEBUG_LOG_FILE"
CONSTANTS_FILE = "LJM_CONSTANTS_FILE"
DEBUG_LOG_FILE_MAX_SIZE = "LJM_DEBUG_LOG_FILE_MAX_SIZE"
SPECIFIC_IPS_FILE = "LJM_SPECIFIC_IPS_FILE"
STREAM_AIN_BINARY = "LJM_STREAM_AIN_BINARY"
STREAM_RECEIVE_TIMEOUT_MODE = "LJM_STREAM_RECEIVE_TIMEOUT_MODE"
STREAM_RECEIVE_TIMEOUT_MS = "LJM_STREAM_RECEIVE_TIMEOUT_MS"
STREAM_SCANS_RETURN = "LJM_STREAM_SCANS_RETURN"
STREAM_TRANSFERS_PER_SECOND = "LJM_STREAM_TRANSFERS_PER_SECOND"
RETRY_ON_TRANSACTION_ID_MISMATCH = "LJM_RETRY_ON_TRANSACTION_ID_MISMATCH"
OLD_FIRMWARE_CHECK = "LJM_OLD_FIRMWARE_CHECK"
LJM_USE_TCP_INIT_FOR_T7_WIFI_TCP = "LJM_USE_TCP_INIT_FOR_T7_WIFI_TCP"
LJM_ZERO_LENGTH_ARRAY_MODE = "LJM_ZERO_LENGTH_ARRAY_MODE"

3.1 - Timeout Configs

Timeout Configs Overview

LJM can be configured to set separate timeouts when connecting to a device and when sending / receiving packets to / from a device. Both of these timeout types can be configured separately for USB, Ethernet, and WiFi.

The open timeouts are:

- LJM_ETHERNET_OPEN_TIMEOUT_MS
- LJM_WIFI_OPEN_TIMEOUT_MS

LJM_OPEN_TCP_DEVICE_TIMEOUT_MS may be used to set both open timeout configs at the same time.

The send / receive (command-response) timeouts are:

- LJM_USB_SEND_RECEIVE_TIMEOUT_MS
- LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS
- LJM_WIFI_SEND_RECEIVE_TIMEOUT_MS

LJM_SEND_RECEIVE_TIMEOUT_MS may be used to set all three send / receive timeouts at the same time.

All timeouts are in units of milliseconds.

Details

One command-response operation (as performed by a function like LJM_eReadName, for example) may take longer than the timeout for that connection type. This is because there are multiple timeout periods -- one to send the command and another to receive the response.

Setting a new timeout value will not affect pending timeouts.

Setting a timeout value to 0 sets an infinite timeout.

Relevant Functions

To read a timeout config, use LJM_ReadLibraryConfigS.

To write a timeout config, use LJM_WriteLibraryConfigS or LJM_LoadConfigurationFile.

Example

[C/C++] Read LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS, set LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS, then read it again

```

char ErrorString[LJM_MAX_NAME_SIZE];
double Value = 0;
int LJMError = LJM_ReadLibraryConfigS(LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS, &Value);
if (LJMError != 0) {
    LJM_ErrorToString(LJMError, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
printf("The default for LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS is %.00f milliseconds\n", Value);

Value = 3000;
printf("Setting LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS to %.00f milliseconds\n", Value);
LJMError = LJM_WriteLibraryConfigS(LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS, Value);
if (LJMError != 0) {
    LJM_ErrorToString(LJMError, ErrorString);
    printf("LJM_WriteLibraryConfigS error: %s\n", ErrorString);
}

LJMError = LJM_ReadLibraryConfigS(LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS, &Value);
if (LJMError != 0) {
    LJM_ErrorToString(LJMError, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
printf("LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS is now %.00f milliseconds\n", Value);

```

Possible output:

```

The default for LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS is 2000 milliseconds
Setting LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS to 3000 milliseconds
LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS is now 3000 milliseconds

```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_ETHERNET_OPEN_TIMEOUT_MS

Summary

LJM_ETHERNET_OPEN_TIMEOUT_MS is a numerical readable-writable LJM library configuration which sets how long in milliseconds LJM waits for a Ethernet-based TCP connection to be established.

The constant LJM_ETHERNET_OPEN_TIMEOUT_MS can be used interchangeably with the string "LJM_ETHERNET_OPEN_TIMEOUT_MS".

Details

See [Timeout Configs](#).

If the pending connection may be Ethernet or WiFi, [LJM_WIFI_OPEN_TIMEOUT_MS](#) is used.

For more LJM configurations, see [Library Configuration Functions](#).

LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS

Summary

LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS is a numerical readable-writable LJM library configuration which sets how long LJM waits for a Ethernet-connected TCP packet to be sent or received in milliseconds.

The constant LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS can be used interchangeably with the string "LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS".

Default

The default is 2600 milliseconds.

Details

See [Timeout Configs](#).

For more LJM configurations, see [Library Configuration Functions](#).

LJM_USB_SEND_RECEIVE_TIMEOUT_MS

Summary

LJM_USB_SEND_RECEIVE_TIMEOUT_MS is a numerical readable-writable LJM library configuration which sets how long LJM waits for a USB packet to be sent or received in milliseconds.

The constant LJM_USB_SEND_RECEIVE_TIMEOUT_MS can be used interchangeably with the string "LJM_USB_SEND_RECEIVE_TIMEOUT_MS".

Default

The default is 2600 milliseconds.

Details

See [Timeout Configs](#).

For more LJM configurations, see [Library Configuration Functions](#).

LJM_WIFI_OPEN_TIMEOUT_MS

Summary

LJM_WIFI_OPEN_TIMEOUT_MS is a numerical readable-writable LJM library configuration which sets how long in milliseconds LJM waits for a WiFi-based TCP connection to be established.

The constant LJMWIFI_OPEN_TIMEOUT_MS can be used interchangeably with the string "LJM_WIFI_OPEN_TIMEOUT_MS".

Details

See [Timeout Configs](#).

If the pending connection may be Ethernet or WiFi, LJMWIFI_OPEN_TIMEOUT_MS is used.

For more LJM configurations, see [Library Configuration Functions](#).

LJM_WIFI_SEND_RECEIVE_TIMEOUT_MS

Summary

LJM_WIFI_SEND_RECEIVE_TIMEOUT_MS is a numerical readable-writable LJM library configuration which sets how long LJM waits for a WiFi-connected TCP packet to be sent or received in milliseconds.

The constant LJMWIFI_SEND_RECEIVE_TIMEOUT_MS can be used interchangeably with the string "LJM_WIFI_SEND_RECEIVE_TIMEOUT_MS".

Default

The default is 4000 milliseconds.

Details

See [Timeout Configs](#).

For more LJM configurations, see [Library Configuration Functions](#).

LJM_OPEN_TCP_DEVICE_TIMEOUT_MS

Summary

LJM_OPEN_TCP_DEVICE_TIMEOUT_MS is a numerical readable-writable LJM library configuration which is a shortcut for setting the following configs:

- LJMEETHERNET_OPEN_TIMEOUT_MS
- LJMWIFI_OPEN_TIMEOUT_MS

The constant LJM_OPEN_TCP_DEVICE_TIMEOUT_MS can be used interchangeably with the string "LJM_OPEN_TCP_DEVICE_TIMEOUT_MS".

Reading LJM_OPEN_TCP_DEVICE_TIMEOUT_MS may be meaningless if it has not previously been written.

Details

See [Timeout Configs](#).

For more LJM configurations, see [Library Configuration Functions](#).

LJM_SEND_RECEIVE_TIMEOUT_MS

Summary

LJM_SEND_RECEIVE_TIMEOUT_MS is a numerical readable-writable LJM library configuration which is a shortcut for setting the following configs:

- LJMUSB_SEND_RECEIVE_TIMEOUT_MS

- LJM_ETHERNET_SEND_RECEIVE_TIMEOUT_MS
- LJM_WIFI_SEND_RECEIVE_TIMEOUT_MS

The constant LJM_SEND_RECEIVE_TIMEOUT_MS can be used interchangeably with the string "LJM_SEND_RECEIVE_TIMEOUT_MS".

Reading LJM_SEND_RECEIVE_TIMEOUT_MS may be meaningless if it has not previously been written.

Details

See [Timeout Configs](#).

For more LJM configurations, see [Library Configuration Functions](#).

3.2 - LJM Specific IPs

A Note About Auto IPs

LJM 1.1500 adds the [Auto IPs](#) feature, which provides most of the benefit of specific IPs, but does so automatically. Most users should not need to edit the `ljm_specific_ips.config` file.

LJM Specific IPs

If LJM isn't finding an Ethernet or WiFi device during a search open (i.e. a non-specific open where "ANY" is passed for some parameters), use `ljm_specific_ips.config` to configure LJM to specially check specific IP addresses. LJM tries to open each IP address in `ljm_specific_ips.config` during relevant [Open](#) calls and [ListAll](#) calls.

Example `ljm_specific_ips.config`

Regardless of your computer's IP address, the following `ljm_specific_ips.config` file would trigger 192.168.2.207 and 10.0.0.123 to be checked:

```
// Have LJM connect to the static-IP T7...
// ...in the lab:
192.168.2.207

// ...in the kitchen:
10.0.0.123
```

Syntax

File syntax:

- Whitespace is ignored.
- Empty lines and lines starting with `//` are ignored.
- All other lines are expected to contain one IP address, which should not be a broadcast address.

Default Location

Windows Vista and later:

```
C:\ProgramData\LabJack\LJM\ljm_specific_ips.config
```

Windows XP:

```
C:\Documents and Settings\All Users\Application Data\LabJack\LJM\ljm_specific_ips.config
```

Mac OS X and Linux:

```
/usr/local/share/LabJack/LJM/ljm_specific_ips.config
```

`ljm_specific_ips.config` is not overwritten by the installer.

Remarks

For programmatically setting the file path, see [LJM_SPECIFIC_IPS_FILE](#).

For checking if LJM has successfully read the specific IPs file, use [LJM_GetSpecificIPsInfo](#).

Specific IPs are treated with higher priority than normal IP addresses. They checked before or during the regular UDP broadcast search and are more likely to be an Open result.

Contact your network administrator to ensure that each of the desired IPs will be reachable via TCP/UDP as a static IP address. You can also ping each address to ensure that they are reachable.

LJM_SPECIAL_ADDRESSES_STATUS

`LJM_SPECIAL_ADDRESSES_STATUS` is deprecated. Please use [LJM_GetSpecificIPsInfo](#) instead.

Summary

Passing `LJM_SPECIAL_ADDRESSES_STATUS` to `LJM_ReadLibraryConfigStringS` returns a string that reports the current success or failure of parsing `LJM_SPECIAL_ADDRESSES_FILE`.

`LJM_SPECIAL_ADDRESSES_STATUS` is read-only.

The constant `LJM_SPECIAL_ADDRESSES_STATUS` can be used interchangeably with the string "`LJM_SPECIAL_ADDRESSES_STATUS`".

Possible Values

Below are some example values that may be reported as `LJM_SPECIAL_ADDRESSES_STATUS`. This feature is in development, so if you need to programmatically parse `LJM_SPECIAL_ADDRESSES_STATUS`, please [contact us](#).

- Success. File: `file_path`, IPs: `[comma-separated_list_of_IP_addresses]`
- File load failure: `file_path`

Remarks

See the [Special Addresses Configs overview](#).

For more LJM configurations, see [Library Configuration Functions](#).

LJM_SPECIFIC_IPS_FILE

Summary

`LJM_SPECIFIC_IPS_FILE` is an LJM configuration that describes the absolute or relative file path of the [LJM specific IPs](#) file.

To read, pass `LJM_SPECIFIC_IPS_FILE` to `LJM_ReadLibraryConfigStringS` to get a string containing the file path.

To write, pass `LJM_SPECIFIC_IPS_FILE` and string containing a file path to `LJM_WriteLibraryConfigStringS`, or use [LJM_LoadConfigurationFile](#).

The constant `LJM_SPECIFIC_IPS_FILE` can be used interchangeably with the string "`LJM_SPECIFIC_IPS_FILE`".

Remarks

To set `LJM_SPECIFIC_IPS_FILE` to the default location, set it to "default" or an empty string. For example, in C/C++:

```
int error = LJM_WriteLibraryConfigStringS(LJM_SPECIFIC_IPS_FILE, "default");
```

The default file path specified by `LJM_SPECIFIC_IPS_FILE` is only parsed on LJM startup, so changes made to the file will not affect running LJM processes unless `LJM_WriteLibraryConfigStringS(LJM_SPECIFIC_IPS_FILE, ...)` is called.

`LJM_SPECIFIC_IPS_FILE` replaces the deprecated `LJM_SPECIAL_ADDRESSES_FILE`.

For more LJM configurations, see [Library Configuration Functions](#).

3.3 - LJM Startup Configs

LJM Startup Configs

If you need to change LJM's [configs](#) either globally or quickly, you can edit the `ljm_startup_configs.json` file.

Default Location

Windows Vista and later:

```
C:\ProgramData\LabJack\LJM\ljm_startup_configs.json
```

Windows XP:

```
C:\Documents and Settings\All Users\Application Data\LabJack\LJM\ljm_startup_configs.json
```

Mac OS X and Linux:

```
/usr/local/share/LabJack/LJM/ljm_startup_configs.json
```

Parsing

- Parsed before any other configs are set, so config values that are manually set using the [programmatic config functions](#) (such as [LJM_WriteLibraryConfigS](#)) will take precedence over the values in `ljm_startup_configs.json`.
- Parsed during LJM startup, so processes that have already loaded LJM will not be altered by new changes to `ljm_startup_configs.json`.
- `ljm_startup_configs.json` (and the containing folder) may be overwritten when LJM updates, so if you'd like to make changes to the configurations that are loaded when LJM starts, use `ljm_startup_configs.json` as a template to make a different file in a different directory that you control, then load that file using the [LJM_LoadConfigurationFile](#) function at the beginning of your program.

Default Values

Use the string "default" to load the default value for each config parameter. Otherwise, see the "type" for each config parameter for what kinds of values can be accepted. Please see `LabJackM.h` for more information about what each config parameter does.

Configs that are not included in `ljm_startup_configs.json` will be given the default value.

Syntax

`ljm_startup_configs.json` is parsed by LJM case-insensitively. LJM ignores every key in the top-level JSON object except for `LJM_CONFIG_VALUES`, as well as ignoring every "description", "type", and "values" fields. Other fields in `LJM_CONFIG_VALUES` which are not recognized by LJM will generate errors/warnings. LJM will also ignore lines starting with `"/`.

Windows Paths

Windows file paths must escape `\` characters with `\\`. For example, the path:

```
C:\ljm.log
```

Must be escaped as:

```
C:\\ljm.log
```

3.4 - LJM Auto IPs

LJM Auto IPs Overview

LJM automatically stores network connection information to file to help open connections to LabJack devices during relevant `Open` calls and `ListAll` calls.

Auto IPs functionality was added in LJM 1.1500.

Remarks

For enabling or disabling LJM Auto IPs functionality, see [LJM_AUTO_IPS](#).

For setting the file path, see [LJM_AUTO_IP_FILE](#).

Auto IPs are added when a LabJack device responds to a normal LJM UDP discovery broadcast. If the device is successfully queried for its attributes, those attributes are added to the given auto IP entry. Auto IPs in the auto IPs file are made unique according to serial number combined with connection type. Auto IPs are also removed from the auto IPs file when they have not resulted in a successful open during 10 days. Auto IPs that have not recently had a successful open have a shorter open timeout.

A Note About Specific IPs

LJM has a similar feature called [Specific IPs](#) with the main differences being:

- Specific IPs must be manually added/removed from the [Specific IPs file](#).
- Specific IPs have higher priority over other IP-based opens—they are more likely to be the result of an `Open` call.
- Specific IPs always have a full-length open timeout.

LJM_AUTO_IPS

Summary

`LJM_AUTO_IPS` sets whether or not LJM uses the [auto IPs](#) feature.

The constant `LJM_AUTO_IPS` can be used interchangeably with the string `"LJM_AUTO_IPS"`.

Details

0 - false/disabled

1 - true/enabled (Default)

Relevant Functions

To read `LJM_AUTO_IPS`, use [LJM_ReadLibraryConfigS](#).

To write `LJM_AUTO_IPS`, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

For more LJM configurations, see [Library Configuration Functions](#).

LJM_AUTO_IPS_FILE

Summary

`LJM_AUTO_IPS_FILE` is an LJM configuration that describes the absolute or relative file path of the [LJM auto IPs](#) file.

To read, pass LJM_AUTO_IPS_FILE to `LJM_ReadLibraryConfigStringS` to get a string containing the file path.

To write, pass LJM_AUTO_IPS_FILE and string containing a file path to `LJM_WriteLibraryConfigStringS`, or use `LJM_LoadConfigurationFile`.

The constant LJM_AUTO_IPS_FILE can be used interchangeably with the string "LJM_AUTO_IPS_FILE".

Default Location

Windows Vista and later:

```
C:\ProgramData\LabJack\LJM\ljm_auto_ips.json
```

Windows XP:

```
C:\Documents and Settings\All Users\Application Data\LabJack\LJM\ljm_auto_ips.json
```

Mac OS X and Linux:

```
/usr/local/share/LabJack/LJM/ljm_auto_ips.json
```

Remarks

To set LJM_AUTO_IPS_FILE to the default location, set it to "default" or an empty string. For example, in C/C++:

```
int error = LJM_WriteLibraryConfigStringS(LJM_AUTO_IPS_FILE, "default");
```

LJM will attempt to create the file specified by LJM_AUTO_IPS_FILE if it does not exist.

For more LJM configurations, see [Library Configuration Functions](#).

3.5 - LJM Deep Search

LJM Deep Search

LJM Deep Search is a LJM feature that searches IP address ranges for LabJack devices. It is meant to be used to find DHCP-enabled devices on networks where UDP broadcasts do not propagate to the device, such as when the device is on a separate subnet. Most users should not need to use LJM Deep Search, since the normal UDP broadcast discovery usually works.

Deep Search adds time to device searching. Though Deep Search can find devices with static IPs, [Specific IPs](#) is more suitable for that purpose because Specific IPs is quicker.

Deep Search is enabled by writing IP address range(s) to the LJM_DEEP_SEARCH_FILE.

When LJM_DEEP_SEARCH_FILE contains IP addresses ranges(s), Deep Search is triggered after the normal UDP broadcast discovery during for either of the following situations:

1. ListAll or ListAllExtended is called
2. Open or OpenS is called with all of the following conditions:
 - A. The open call is passed a network-based ConnectionType—essentially any ConnectionType except USB
 - B. The open call is passed a Identifier that is a device name or serial number
 - C. A device with the specified name or serial number cannot be found after UDP broadcast discovery [Auto IPs](#), and [Specific IPs](#) searching

Deep Search stores results as [Auto IPs](#), to speed up future open calls.

Example ljm_deep_search.config

Regardless of you computer's IP address, the following ljm_deep_search.config file would trigger search of 155 IP addresses, including 192.168.2.100 and 192.168.2.254:

```
192.168.2.100-254
```

Syntax

File syntax:

- Whitespace is ignored.
- Empty lines and lines starting with // are ignored.
- All other lines are expected to contain one IP address range, which should not include a broadcast address. The IP address range must be an IPv4 address with the last octet being two numbers separated by a dash. For example: `192.168.2.100-254`

Default Location

Windows Vista and later:

```
C:\ProgramData\LabJack\LJM\ljm_deep_search.config
```

Windows XP:

```
C:\Documents and Settings\All Users\Application Data\LabJack\LJM\ljm_deep_search.config
```

Mac OS X and Linux:

```
/usr/local/share/LabJack/LJM/ljm_deep_search.config
```

ljm_deep_search.config is not overwritten by the installer.

Remarks

For programmatically setting the file path, see [LJM_DEEP_SEARCH_FILE](#).

For checking if LJM has successfully read the Deep Search file, use [LJM_GetDeepSearchInfo](#).

macOS Issue

macOS has a file limit, per Terminal shell. The default is low: 256. Deep Search opens many sockets in parallel, so opening any files (sockets) after 256 will fail. This can cause Deep Search to become rather useless unless the file limit is increased.

To increase the file limit to 2048 (which is typically sufficient for Deep Search), execute the following command in your shell:

```
ulimit -S -n 2048
```

This change only lasts for the current shell session. To make it permanent, you can add it to your shell's startup configurations. (E.g. `.bashrc`, `.bash_profile`, etc.)

LJM_DEEP_SEARCH_FILE

Summary

`LJM_DEEP_SEARCH_FILE` is an LJM configuration that describes the absolute or relative file path of the `LJMDeep_Search` file.

To read, pass `LJM_DEEP_SEARCH_FILE` to [LJM_ReadLibraryConfigStringS](#) to get a string containing the file path.

To write, pass `LJM_DEEP_SEARCH_FILE` and string containing a file path to [LJM_WriteLibraryConfigStringS](#), or use [LJM_LoadConfigurationFile](#).

The constant `LJM_DEEP_SEARCH_FILE` can be used interchangeably with the string `"LJM_DEEP_SEARCH_FILE"`.

Remarks

To set `LJM_DEEP_SEARCH_FILE` to the default location, set it to `"default"` or an empty string. For example, in C/C++:

```
int error = LJM_WriteLibraryConfigStringS(LJM_DEEP_SEARCH_FILE, "default");
```

The default file path specified by `LJM_DEEP_SEARCH_FILE` is only parsed on LJM startup, so changes made to the file will not affect running LJM processes unless `LJM_WriteLibraryConfigStringS(LJM_DEEP_SEARCH_FILE, ...)` is called.

For more LJM configurations, see [Library Configuration Functions](#).

3.6 - Device Search Configs

When LJM searches for devices in the following situations:

- During an open or `ListAll` call
- During reconnect

The below parameters configure device search behavior.

Retries:

- `LJM_LISTALL_NUM_ATTEMPTS_ETHERNET`
 - Determines the number of times LJM tries to discover Ethernet connections per open/`ListAll` call.
 - Default: 2
- `LJM_LISTALL_NUM_ATTEMPTS_WIFI`
 - Determines the number of times LJM tries to discover WiFi connections per open/`ListAll` call.
 - Default: 2

Concurrent searching:

- `LJM_LISTALL_THREADED_ETHERNET`
 - Determines whether LJM searches for Ethernet connections in parallel with the other connection type searches or not
 - Default: true
- `LJM_LISTALL_THREADED_USB`
 - Determines whether LJM searches for USB connections in parallel with the other connection type searches or not
 - Default: true
- `LJM_LISTALL_THREADED_WIFI`
 - Determines whether LJM searches for WiFi connections in parallel with the other connection type searches or not
 - Default: true

Timeout:

Determines how long in milliseconds LJM waits for device responses:

- LJM_LISTALL_TIMEOUT_MS_ETHERNET
 - Default: 900
- LJM_LISTALL_TIMEOUT_MS_WIFI
 - Default: 1000

Protocol:

Determines whether LJM initializes the connection via UDP (true) or TCP (false):

- LJM_LISTALL_UDP_ETHERNET
 - Default: true
- LJM_LISTALL_UDP_WIFI
 - Default: true

Relevant Functions

To read the above configs, use [LJM_ReadLibraryConfigS](#).

To write the above configs, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

Remarks

Other related device connection configurations:

- [Auto IPs](#) - Automatically stored connection information for helping to open connections.
- [Specific IPs](#) - Manual configurations for connecting to statically-addressed LabJack devices.
- [Deep Search](#) - Manual configurations for connecting to DHCP-addressed LabJack devices.

3.7 - RPC Configs

LJM uses gRPC to coordinate device connections. This allows LJM to retrieve information about USB-connected LabJack devices.

The below parameters configure LJM RPC behavior.

- LJM_RPC_HEARTBEAT_PORT
 - Determines the port that LJM uses to coordinate device connections.
 - Default: 50051
 - Windows: cmd command to check if port 50051 is in use: `netstat -na | find "50051"`
- LJM_RPC_ENABLE
 - Enables (when true) or disables (when false) RPC communications from a given instance of LJM.
 - Default: true
- LJM_RPC_TIMEOUT_MS
 - Determines the multi-purpose RPC timeout in milliseconds
 - Default: 1000

LJM_ALLOWED_AUTO_CONDENSE_ADDRESSES

Summary

LJM_ALLOWED_AUTO_CONDENSE_ADDRESSES is a numerical readable-writable LJM library configuration that determines whether or not LJM will automatically condense congruent address reads/writes into array reads/writes.

For example, if a call to [LJM_eNames](#) contains two frames next to each other that write to AIN0 then AIN1, LJM would automatically condense these frames into one frame that writes to both AIN0 and AIN1.

The constant LJM_ALLOWED_AUTO_CONDENSE_ADDRESSES can be used interchangeably with the string "LJM_ALLOWED_AUTO_CONDENSE_ADDRESSES".

Details

0 - false/disabled

1 - true/enabled (Default)

Relevant Functions

To read LJM_ALLOWED_AUTO_CONDENSE_ADDRESSES, use [LJM_ReadLibraryConfigS](#).

To write LJM_ALLOWED_AUTO_CONDENSE_ADDRESSES, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

Example

[C/C++] Read LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES, set LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES, then read it again

```

char ErrorString[LJM_MAX_NAME_SIZE];
double Value;
int LJMErr = LJM_ReadLibraryConfigS(LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES, &Value);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
if (Value == 0) {
    printf("LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES is disabled by default.\n");
}
else {
    printf("LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES is enabled by default.\n");
}

printf("Disabling LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES\n");
LJMErr = LJM_WriteLibraryConfigS(LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES, 0);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_WriteLibraryConfigS error: %s\n", ErrorString);
}

LJMErr = LJM_ReadLibraryConfigS(LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES, &Value);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
if ((int)Value == 0) {
    printf("LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES is now disabled.\n");
}
else {
    printf("LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES is now enabled.\n");
}

```

Possible output:

```

LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES is enabled by default.
Disabling LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES
LJM_ALLOWEDS_AUTO_CONDENSE_ADDRESSES is now disabled.

```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_ALLOWEDS_AUTO_MULTIPLE_FEEDBACKS

Summary

LJM_ALLOWEDS_AUTO_MULTIPLE_FEEDBACKS is a numerical readable-writable LJM library configuration that determines whether or not LJM will automatically perform multiple Feedback commands when the desired operations would exceed the maximum packet length.

For example, if a call to [LJM_eNames](#) would require a 68-byte packet but the connection can only take 64-byte packets, LJM would automatically separate the [LJM_eNames](#) call into 2 operations of less than 64 bytes each.

The constant LJM_ALLOWEDS_AUTO_MULTIPLE_FEEDBACKS can be used interchangeably with the string "LJM_ALLOWEDS_AUTO_MULTIPLE_FEEDBACKS".

Details

0 - false/disabled

1 - true/enabled (Default)

Relevant Functions

To read LJM_ALLOWEDS_AUTO_MULTIPLE_FEEDBACKS, use [LJM_ReadLibraryConfigS](#).

To write LJM_ALLOWEDS_AUTO_MULTIPLE_FEEDBACKS, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

Example

[C/C++] Read LJM_ALLOWEDS_AUTO_MULTIPLE_FEEDBACKS, set LJM_ALLOWEDS_AUTO_MULTIPLE_FEEDBACKS, then read it again

```

char ErrorString[LJM_MAX_NAME_SIZE];
double Value;
int LJMErr = LJM_ReadLibraryConfigS(LJM_ALLOWEDS_AUTO_MULTIPLE_FEEDBACKS, &Value);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}

```

```

}
if ((int)Value == 0) {
    printf("LJM_ALLOWED_AUTO_MULTIPLE_FEEDBACKS is disabled by default.\n");
}
else {
    printf("LJM_ALLOWED_AUTO_MULTIPLE_FEEDBACKS is enabled by default.\n");
}

printf("Disabling LJM_ALLOWED_AUTO_MULTIPLE_FEEDBACKS\n");
LJMErr = LJM_WriteLibraryConfigS(LJM_ALLOWED_AUTO_MULTIPLE_FEEDBACKS, 0);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_WriteLibraryConfigS error: %s\n", ErrorString);
}

LJMErr = LJM_ReadLibraryConfigS(LJM_ALLOWED_AUTO_MULTIPLE_FEEDBACKS, &Value);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
if ((int)Value == 0) {
    printf("LJM_ALLOWED_AUTO_MULTIPLE_FEEDBACKS is now disabled.\n");
}
else {
    printf("LJM_ALLOWED_AUTO_MULTIPLE_FEEDBACKS is now enabled.\n");
}

```

Possible output:

```

LJM_ALLOWED_AUTO_MULTIPLE_FEEDBACKS is enabled by default.
Disabling LJM_ALLOWED_AUTO_MULTIPLE_FEEDBACKS
LJM_ALLOWED_AUTO_MULTIPLE_FEEDBACKS is now disabled.

```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_AUTO_RECONNECT_STICKY_CONNECTION

Summary

LJM_AUTO_RECONNECT_STICKY_CONNECTION is a numerical readable-writable LJM library configuration that sets whether or not LJM attempts to reconnect disrupted / disconnected connections according to same connection type as the original handle. When LJM_AUTO_RECONNECT_STICKY_CONNECTION is disabled, the ConnectionType that was used during the initial [LJM_Open](#) or [LJM_OpenS](#) call will be used to reconnect.

The constant LJM_AUTO_RECONNECT_STICKY_CONNECTION can be used interchangeably with the string "LJM_AUTO_RECONNECT_STICKY_CONNECTION".

Details

0 - false/disabled - Reconnects according to the ConnectionType parameter that was passed to the Open call

1 - true/enabled (Default) - Reconnects via the actual connection type that was used to connect to the device

Relevant Functions

To read LJM_AUTO_RECONNECT_STICKY_CONNECTION, use [LJM_ReadLibraryConfigS](#).

To write LJM_AUTO_RECONNECT_STICKY_CONNECTION, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

See also: [LJM_AUTO_RECONNECT_STICKY_SERIAL](#).

For more LJM configurations, see [Library Configuration Functions](#).

LJM_AUTO_RECONNECT_STICKY_SERIAL

Summary

LJM_AUTO_RECONNECT_STICKY_SERIAL is a numerical readable-writable LJM library configuration that sets whether or not LJM attempts to reconnect disrupted / disconnected connections according to same serial number as the original handle. When LJM_AUTO_RECONNECT_STICKY_SERIAL is disabled, the Identifier that was used during the initial [LJM_Open](#) or [LJM_OpenS](#) call will be used to reconnect.

The constant LJM_AUTO_RECONNECT_STICKY_SERIAL can be used interchangeably with the string "LJM_AUTO_RECONNECT_STICKY_SERIAL".

Details

0 - false/disabled

1 - true/enabled (Default)

Relevant Functions

To read LJM_AUTO_RECONNECT_STICKY_SERIAL, use [LJM_ReadLibraryConfigS](#).

To write LJM_AUTO_RECONNECT_STICKY_SERIAL, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

See also: [LJM_AUTO_RECONNECT_STICKY_CONNECTION](#).

For more LJM configurations, see [Library Configuration Functions](#).

LJM_AUTO_RECONNECT_WAIT_MS

Summary

LJM_AUTO_RECONNECT_WAIT_MS is a numerical readable-writable LJM library configuration that sets how long in milliseconds LJM waits between attempts to reconnect when a device has been found to be disconnected. The default is 500 milliseconds.

The constant LJM_AUTO_RECONNECT_WAIT_MS can be used interchangeably with the string "LJM_AUTO_RECONNECT_WAIT_MS".

Relevant Functions

To read LJM_AUTO_RECONNECT_WAIT_MS, use [LJM_ReadLibraryConfigS](#).

To write LJM_AUTO_RECONNECT_WAIT_MS, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

See also: [How to deal with LJME_RECONNECT_FAILED](#).

For more LJM configurations, see [Library Configuration Functions](#).

LJM_CONSTANTS_FILE

Summary

LJM_CONSTANTS_FILE is a string-based write-only LJM library configuration which sets the absolute or relative path of an existing file to read Modbus constants and error constants from. It is a shortcut for writing to both [LJM_MODBUS_MAP_CONSTANTS_FILE](#) and [LJM_ERROR_CONSTANTS_FILE](#).

The constant LJM_CONSTANTS_FILE can be used interchangeably with the string "LJM_CONSTANTS_FILE".

Default Value

Windows Vista and later

C:\ProgramData\LabJack\LJM\ljm_constants.json

Windows XP

C:\Documents and Settings\All Users\Application Data\LabJack\LJM\ljm_constants.json

Mac OS X and Linux

/usr/local/share/LabJack/LJM/ljm_constants.json

Relevant Constants

[LJM_MODBUS_MAP_CONSTANTS_FILE](#) and [LJM_ERROR_CONSTANTS_FILE](#)

Relevant Functions

To write LJM_CONSTANTS_FILE, use [LJM_WriteLibraryConfigStringS](#) or [LJM_LoadConfigurationFile](#).

Setting LJM_CONSTANTS_FILE may affect the output of the following functions:

- [LJM_ErrorToString](#)
- [LJM_NameToAddress](#)
- [LJM_NamesToAddresses](#)
- [LJM_AddressToType](#)
- [LJM_AddressesToTypes](#)

Example

[C/C++] Set LJM_CONSTANTS_FILE to "alternate_constants.json"

```
char ErrorString[LJM_MAX_NAME_SIZE];
int LJMErr = LJM_WriteLibraryConfigStringS(LJM_CONSTANTS_FILE, "alternate_constants.json");
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_WriteLibraryConfigStringS error: %s\n", ErrorString);
}
```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_DEBUG_LOG_FILE

Default

Windows Vista and later

C:\ProgramData\LabJack\LJM\ljm.log

Windows XP

C:\Documents and Settings\All Users\Application Data\LabJack\LJM\ljm.log

Mac OS X and Linux

/usr/local/share/LabJack/LJM/ljm.log

Summary

LJM_DEBUG_LOG_FILE is a string-based readable-writable LJM library configuration which sets the absolute or relative path of the file to output log messages to.

The constant LJM_DEBUG_LOG_FILE can be used interchangeably with the string "LJM_DEBUG_LOG_FILE".

Details

The configuration value of LJM_DEBUG_LOG_FILE is meaningless if [LJM_DEBUG_LOG_MODE](#) is in LJM_DEBUG_LOG_MODE_NEVER (which is the default).

Relevant Constants

See these [debug logger parameters](#), especially [LJM_DEBUG_LOG_MODE](#).

Relevant Functions

To read LJM_DEBUG_LOG_FILE, use [LJM_ReadLibraryConfigStringS](#).

To write LJM_DEBUG_LOG_FILE, use [LJM_WriteLibraryConfigStringS](#) or [LJM_LoadConfigurationFile](#).

For more information, see [2.7 Debugging Functions](#)

Example

[C/C++] Set LJM to log to the log file "ljm_monday.log"

```
char ErrorString[LJM_MAX_NAME_SIZE];
char DebugLogFile[LJM_MAX_NAME_SIZE];
int LJLError;

printf("Setting LJM_DEBUG_LOG_MODE to LJM_DEBUG_LOG_MODE_CONTINUOUS\n");
LJLError = LJM_WriteLibraryConfigS(LJM_DEBUG_LOG_MODE, LJM_DEBUG_LOG_MODE_CONTINUOUS);
if (LJLError != 0) {
    LJM_ErrorToString(LJLError, ErrorString);
    printf("LJM_WriteLibraryConfigS error: %s\n", ErrorString);
}

printf("Setting LJM_DEBUG_LOG_FILE to ljm_monday.log\n");
LJLError = LJM_WriteLibraryConfigStringS(LJM_DEBUG_LOG_FILE, "ljm_monday.log");
if (LJLError != 0) {
    LJM_ErrorToString(LJLError, ErrorString);
    printf("LJM_WriteLibraryConfigStringS error: %s\n", ErrorString);
}

LJLError = LJM_ReadLibraryConfigStringS(LJM_DEBUG_LOG_FILE, DebugLogFile);
if (LJLError != 0) {
    LJM_ErrorToString(LJLError, ErrorString);
    printf("LJM_ReadLibraryConfigStringS error: %s\n", ErrorString);
}
printf("LJM_DEBUG_LOG_FILE is now %s\n", DebugLogFile);
```

Possible output:

```
Setting LJM_DEBUG_LOG_MODE to LJM_DEBUG_LOG_MODE_CONTINUOUS
Setting LJM_DEBUG_LOG_FILE to ljm_monday.log
LJM_DEBUG_LOG_FILE is now ljm_monday.log
```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_DEBUG_LOG_FILE_MAX_SIZE

Summary

LJM_DEBUG_LOG_FILE_MAX_SIZE is a numerical readable-writable LJM library configuration that limits the number of characters in the `debug_log` file. Once the debug log file has exceeded LJM_DEBUG_LOG_FILE_MAX_SIZE characters, it will be reset to 0 characters and debug logging will continue. The largest LJM_DEBUG_LOG_FILE_MAX_SIZE you may use is the largest file size of your operating system. The default is 50000 characters.

The constant LJM_DEBUG_LOG_FILE_MAX_SIZE can be used interchangeably with the string "LJM_DEBUG_LOG_FILE_MAX_SIZE".

Relevant Functions

To read LJM_DEBUG_LOG_FILE_MAX_SIZE, use [LJM_ReadLibraryConfigS](#).

To write LJM_DEBUG_LOG_FILE_MAX_SIZE, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

Example

[C/C++] Set LJM_DEBUG_LOG_FILE_MAX_SIZE to a larger-than-default size

```
char ErrorString[LJM_MAX_NAME_SIZE];
int LJMEError;
LJMEError = LJM_WriteLibraryConfigS(LJM_DEBUG_LOG_FILE_MAX_SIZE, 123456789);
if (LJMEError != 0) {
    LJM_ErrorToString(LJMEError, ErrorString);
    printf("LJM_WriteLibraryConfigS error: %s\n", ErrorString);
}
```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_DEBUG_LOG_LEVEL

Summary

LJM_DEBUG_LOG_LEVEL is a numerical readable-writable LJM library configuration with the following options:

- LJM_STREAM_PACKET = 1
- LJM_TRACE = 2
- LJM_DEBUG = 4
- LJM_INFO = 6
- LJM_PACKET = 7
- LJM_WARNING = 8
- LJM_USER = 9
- LJM_ERROR = 10
- LJM_FATAL = 12

The constant LJM_DEBUG_LOG_LEVEL can be used interchangeably with the string "LJM_DEBUG_LOG_LEVEL".

Details

[LJM_DEBUG_LOG_MODE](#) must allow for logging in order for LJM_DEBUG_LOG_LEVEL to have any effect.

LJM_DEBUG_LOG_LEVEL determines which log messages are output. LJM outputs the debug messages that are of the current LJM_DEBUG_LOG_LEVEL and greater.

Relevant Functions

To read LJM_DEBUG_LOG_LEVEL, use [LJM_ReadLibraryConfigS](#).

To write LJM_DEBUG_LOG_LEVEL, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

For more information, see [2.7 Debugging Functions](#)

Example

[C/C++] Read LJM_DEBUG_LOG_LEVEL then set LJM_DEBUG_LOG_LEVEL to LJM_WARNING

```
char ErrorString[LJM_MAX_NAME_SIZE];
double value;

int LJMEError = LJM_ReadLibraryConfigS(LJM_DEBUG_LOG_LEVEL, &value);
if (LJMEError != 0) {
    LJM_ErrorToString(LJMEError, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
printf("The default for LJM_DEBUG_LOG_LEVEL is %.00f\n", value);

value = LJM_WARNING;
printf("Setting LJM_DEBUG_LOG_LEVEL to %.00f\n", value);
LJMEError = LJM_WriteLibraryConfigS(LJM_DEBUG_LOG_LEVEL, value);
```

```
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_DEBUG_LOG_LEVEL error: %s\n", ErrorString);
}
```

Possible output:

```
The default for LJM_DEBUG_LOG_LEVEL is 7
Setting LJM_DEBUG_LOG_LEVEL to 8
```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_DEBUG_LOG_MODE

Summary

LJM_DEBUG_LOG_MODE is a numerical readable-writable LJM library configuration with the following options:

LJM_DEBUG_LOG_MODE_NEVER = 1

- LJM will never log messages
- The log thread will never start

LJM_DEBUG_LOG_MODE_CONTINUOUS = 2

- LJM will log all messages with priority equal to or greater than the current LJM_DEBUG_LOG_LEVEL
- The log thread will start when the first message with priority equal to or greater than the current LJM_DEBUG_LOG_LEVEL is scheduled to be output

LJM_DEBUG_LOG_MODE_ON_ERROR = 3

- LJM will collect all messages with priority equal to or greater than the current LJM_DEBUG_LOG_LEVEL, but only output those messages if a log message with priority equal to or greater than LJM_ERROR (10) is scheduled to be output
- The maximum number of messages that will be collected is LJM_DEBUG_LOG_BUFFER_MAX_SIZE
- The log thread will start when the first message with priority equal to or greater than the current LJM_DEBUG_LOG_LEVEL is scheduled to be output

The constant LJM_DEBUG_LOG_MODE can be used interchangeably with the string "LJM_DEBUG_LOG_MODE".

Relevant Functions

To read LJM_DEBUG_LOG_MODE, use [LJM_ReadLibraryConfigS](#).

To write LJM_DEBUG_LOG_MODE, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

For more information, see [2.7 Debugging Functions](#).

Example

[C/C++] Read LJM_DEBUG_LOG_MODE, set LJM_DEBUG_LOG_MODE, then read it again

```
char ErrorString[LJM_MAX_NAME_SIZE];
double Value = 0;
int LJMErr = LJM_ReadLibraryConfigS(LJM_DEBUG_LOG_MODE, &Value);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
printf("The default for LJM_DEBUG_LOG_MODE is %.00fn", Value);

Value = LJM_DEBUG_LOG_MODE_CONTINUOUS;
printf("Setting LJM_DEBUG_LOG_MODE to %.00fn", Value);
LJMErr = LJM_WriteLibraryConfigS(LJM_DEBUG_LOG_MODE, Value);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_WriteLibraryConfigS error: %s\n", ErrorString);
}

LJMErr = LJM_ReadLibraryConfigS(LJM_DEBUG_LOG_MODE, &Value);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
printf("LJM_DEBUG_LOG_MODE is now %.00fn", Value);
```

Possible output:

```
The default for LJM_DEBUG_LOG_MODE is 1
Setting LJM_DEBUG_LOG_MODE to 2
LJM_DEBUG_LOG_MODE is now 2
```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_ERROR_CONSTANTS_FILE

Summary

LJM_ERROR_CONSTANTS_FILE is a string-based readable-writable LJM library configuration which sets the absolute or relative path of an existing file to use as error constants for use with the function [LJM_ErrorToString](#).

The constant LJM_ERROR_CONSTANTS_FILE can be used interchangeably with the string "LJM_ERROR_CONSTANTS_FILE".

Default Value

- Windows Vista and later: "C:\ProgramData\LabJack\LJM\ljm_constants.json"
- Windows XP: "C:\Documents and Settings\All Users\Application Data\LabJack\LJM\ljm_constants.json"
- Mac OS X and Linux: "/usr/local/share/LabJack/LJM/ljm_constants.json"

Relevant Constants

To set LJM_ERROR_CONSTANTS_FILE and [LJM_MODBUS_MAP_CONSTANTS_FILE](#) simultaneously, use [LJM_CONSTANTS_FILE](#).

Relevant Functions

To read, use [LJM_ReadLibraryConfigStringS](#).

To write, use [LJM_WriteLibraryConfigStringS](#) or [LJM_LoadConfigurationFile](#).

Example

[C/C++] Read LJM_ERROR_CONSTANTS_FILE, then set it to "alternate_error_constants.json"

```
char ErrorString[LJM_MAX_NAME_SIZE];
char defaultErrorConstantsFile[LJM_MAX_NAME_SIZE];
char * newErrorConstantsFile = "alternate_error_constants.json";

int LJMErr = LJM_ReadLibraryConfigStringS(LJM_ERROR_CONSTANTS_FILE, defaultErrorConstantsFile);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigStringS error: %s\n", ErrorString);
}
printf("The default for LJM_ERROR_CONSTANTS_FILE is %s\n", defaultErrorConstantsFile);

printf("Setting LJM_ERROR_CONSTANTS_FILE to %s\n", newErrorConstantsFile);
LJMErr = LJM_WriteLibraryConfigStringS(LJM_ERROR_CONSTANTS_FILE, newErrorConstantsFile);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_WriteLibraryConfigStringS error: %s\n", ErrorString);
}
```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_LIBRARY_VERSION

Summary

LJM_LIBRARY_VERSION is a numerical read-only LJM library configuration which is used to read the current LJM version.

The constant LJM_LIBRARY_VERSION can be used interchangeably with the string "LJM_LIBRARY_VERSION".

Relevant Functions

To read LJM_LIBRARY_VERSION, use [LJM_ReadLibraryConfigS](#).

Example

[C/C++] Read and display LJM_LIBRARY_VERSION, then display LJM_VERSION

```
char ErrorString[LJM_MAX_NAME_SIZE];
double Value = 0;
int LJMErr = LJM_ReadLibraryConfigS(LJM_LIBRARY_VERSION, &Value);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
printf("LJM_LIBRARY_VERSION is the version of LJM that is\n");
printf("  running and it is %.04f\n", Value);
printf("LJM_VERSION is the version of LJM that this program\n");
printf("  was compiled with and it is %.04f\n", LJM_VERSION);
```

Possible output:

```
LJM_LIBRARY_VERSION is the version of LJM that is
running and it is 1.0203
LJM_VERSION is the version of LJM that this program
was compiled with and it is 1.0203
```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_MODBUS_MAP_CONSTANTS_FILE

Summary

LJM_MODBUS_MAP_CONSTANTS_FILE is a string-based readable-writable LJM library configuration which sets the absolute or relative path of an existing file to use as Modbus constants for use with the functions that use the Modbus map constants, such as [LJM_NamesToAddresses](#).

The constant LJM_MODBUS_MAP_CONSTANTS_FILE can be used interchangeably with the string "LJM_MODBUS_MAP_CONSTANTS_FILE".

Default Value

- Windows Vista and later: "C:\ProgramData\LabJack\LJM\ljm_constants.json"
- Windows XP: "C:\Documents and Settings\All Users\Application Data\LabJack\LJM\ljm_constants.json"
- Mac OS X and Linux: "/usr/local/share/LabJack/LJM/ljm_constants.json"

Relevant Constants

To set [LJM_ERROR_CONSTANTS_FILE](#) and LJM_MODBUS_MAP_CONSTANTS_FILE simultaneously, use [LJM_CONSTANTS_FILE](#).

Relevant Functions

To read, use [LJM_ReadLibraryConfigStringS](#).

To write, use [LJM_WriteLibraryConfigStringS](#) or [LJM_LoadConfigurationFile](#).

Example

[C/C++] Read LJM_MODBUS_MAP_CONSTANTS_FILE, then set to "alternate_modbus_constants.json"

```
char ErrorString[LJM_MAX_NAME_SIZE];
char defaultModbusConstantsFile[LJM_MAX_NAME_SIZE];
char * newModbusConstantsFile = "alternate_modbus_constants.json";

int LJMErr = LJM_ReadLibraryConfigStringS(LJM_MODBUS_MAP_CONSTANTS_FILE, defaultModbusConstantsFile);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigStringS error: %s\n", ErrorString);
}
printf("The default for LJM_MODBUS_MAP_CONSTANTS_FILE is %s\n", defaultModbusConstantsFile);

printf("Setting LJM_MODBUS_MAP_CONSTANTS_FILE to %s\n", newModbusConstantsFile);
LJMErr = LJM_WriteLibraryConfigStringS(LJM_MODBUS_MAP_CONSTANTS_FILE, newModbusConstantsFile);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_WriteLibraryConfigStringS error: %s\n", ErrorString);
}
```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_OLD_FIRMWARE_CHECK

Summary

LJM_OLD_FIRMWARE_CHECK is a numerical readable-writable LJM library configuration which sets whether or not LJM will check the [Modbus constants file](#) to make sure the firmware of the current device is compatible with the Modbus register(s) being read from or written to.

The constant LJM_OLD_FIRMWARE_CHECK can be used interchangeably with the string "LJM_OLD_FIRMWARE_CHECK".

Values

0 - false/disabled

1 - true/enabled (default)

It is recommended to not disable LJM_OLD_FIRMWARE_CHECK.

Details

For all [Easy functions](#) and for [LJM_MBFBComm](#), each address being read from or written to that also exists in the [Modbus constants file](#), LJM will compare the firmware version of the current device against the minimum firmware required for that address. If the device does not have adequate firmware, the function will abort before communicating with the device and return the error LJME_OLD_FIRMWARE (1307).

Relevant Functions

To read, use [LJM_ReadLibraryConfigS](#).

To write, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

Affects the error checking of all [Easy functions](#) and also [LJM_MBFBComm](#).

Example

[C/C++] Read LJM_OLD_FIRMWARE_CHECK, then disable it.

```
char ErrorString[LJM_MAX_NAME_SIZE];
double value;

int LJMErr = LJM_ReadLibraryConfigS(LJM_OLD_FIRMWARE_CHECK, &value);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
printf("The default for LJM_OLD_FIRMWARE_CHECK is %.00f\n", value);

value = 0;
printf("Setting LJM_OLD_FIRMWARE_CHECK to %.00f\n", value);
LJMErr = LJM_WriteLibraryConfigS(LJM_OLD_FIRMWARE_CHECK, value);
if (LJMErr != 0) {
    LJM_ErrorToString(LJMErr, ErrorString);
    printf("LJM_WriteLibraryConfigS error: %s\n", ErrorString);
}
```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_STREAM_AIN_BINARY

Summary

Sets whether [LJM_eStreamRead](#) will return calibrated or uncalibrated (binary) data.

Below, ScansPerRead is a parameter of [LJM_eStreamStart](#).

LJM_STREAM_AIN_BINARY = 0 (default)

- LJM_eStreamRead will return calibrated data.

LJM_STREAM_AIN_BINARY = 1

- LJM_eStreamRead will return uncalibrated (binary) data.

The constant LJM_STREAM_AIN_BINARY can be used interchangeably with the string "LJM_STREAM_AIN_BINARY".

Remarks

LJM_STREAM_AIN_BINARY does not affect currently running or already initialized streams.

Relevant Functions

To read LJM_STREAM_AIN_BINARY, use [LJM_ReadLibraryConfigS](#).

To write LJM_STREAM_AIN_BINARY, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

LJM_STREAM_AIN_BINARY affects the behavior of [LJM_eStreamRead](#).

For more LJM configurations, see [Library Configuration Functions](#).

LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABL

Summary

Requires LJM 1.2000 or later

LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED is a numerical readable and writable LJM library configuration which sets how LJM handles auto-recovery when the first channel is not firmware-protected from returning 0xFFFF as a valid data value. Channels that never return 0xFFFF during stream are the following:

- Analog inputs (AIN0, AIN1, ...)
- FIO_STATE
- EIO_STATE
- CIO_STATE
- MIO_STATE
- EIO_CIO_STATE
- CIO_MIO_STATE

The constant LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED can be used interchangeably with the string "LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED".

Values

0 (default) - Error detection is enabled:

If auto-recovery happens and the first channel is not one of the above channels, stream will be stopped and LJM_eStreamRead will return the error LJME_DIGITAL_AUTO_RECOVERY_ERROR_DETECTED (errorcode 1320).

1 - Error detection is disabled:

If auto-recovery happens and the first channel is not firmware-protected from return a read data value of 0xFFFF, LJM will treat the first channel as if it cannot return 0xFFFF and will insert the missing scans as LJM_DUMMY_VALUE values.

Details

If the device stream buffer overflows (so that samples can no longer be taken), the LabJack device will count how many scans were skipped. Once LJM has caught up reading from the device stream buffer, the device will commence collecting scans again. After this, the device will send a packet indicating that auto-recovery is complete. This auto-recovery complete packet contains the count of how many scans were skipped and uses the delimiter value of 0xFFFF within the data stream for the first channel to indicate where in the data stream LJM should insert extra LJM_DUMMY_VALUE values to represent skipped scans. To ensure that 0xFFFF can be used as a delimiter value for the first channel, some channels (such as analog inputs) never return 0xFFFF. However, digital channels not listed above can return 0xFFFF under nominal conditions. These channels by default are not assumed by LJM to be usable for the first channel during auto-recovery because a real data value of 0xFFFF is indistinguishable from a delimiter value of 0xFFFF. To allow LJM to treat your first stream channel as if it will never return a real data value of 0xFFFF, you should:

1. Make sure that your first channel will never return 0xFFFF. If your first channel is FIO_EIO_STATE, for example, you could wire FIO3 to GND.
2. Set LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED to 1.

If your first channel cannot be guaranteed to never return 0xFFFF, you can still set LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED to 1, but skipped scans may be inserted by LJM at the wrong place.

Setting a new LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED value will not affect stream sessions that are already in progress.

Relevant Functions

To read LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED, use [LJM_ReadLibraryConfigS](#).

To write LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

Example

[C/C++] Read LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED then set it to 1.

```
char ErrorString[LJM_MAX_NAME_SIZE];
double value;

int LJMEError = LJM_ReadLibraryConfigS(LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED, &value);
if (LJMEError != 0) {
    LJM_ErrorToString(LJMEError, ErrorString);
    printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
printf("The default for LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED is %.00f\n", value);

value = 1;
printf("Setting LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED to %.00f\n", value);
printf("I hope one of the digital inputs is wired to GND!\n");
LJMEError = LJM_WriteLibraryConfigS(LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED, value);
if (LJMEError != 0) {
    LJM_ErrorToString(LJMEError, ErrorString);
    printf("LJM_WriteLibraryConfigS error: %s\n", ErrorString);
}
```

Output:

```
The default for LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED is 0
Setting LJM_STREAM_DIGITAL_AUTO_RECOVERY_ERROR_DETECTION_DISABLED to 1
I hope one of the digital inputs is wired to GND!
```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_STREAM_RECEIVE_TIMEOUT_MODE

Summary

Sets how stream data collection should time out.

LJM_STREAM_RECEIVE_TIMEOUT_MODE_CALCULATED = 1 (default)

- LJM calculates how long the stream timeout should be, according to the scan rate reported by the device.

LJM_STREAM_RECEIVE_TIMEOUT_MODE_MANUAL = 2

- LJM will use the value of [LJM_STREAM_RECEIVE_TIMEOUT_MS](#) as the stream timeout.

The constant `LJM_STREAM_RECEIVE_TIMEOUT_MODE` can be used interchangeably with the string "LJM_STREAM_RECEIVE_TIMEOUT_MODE".

Remarks

`LJM_STREAM_RECEIVE_TIMEOUT_MODE` does not affect currently running or already initialized streams.

Relevant Functions

To read `LJM_STREAM_RECEIVE_TIMEOUT_MODE`, use [LJM_ReadLibraryConfigS](#).

To write `LJM_STREAM_RECEIVE_TIMEOUT_MODE`, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

`LJM_STREAM_RECEIVE_TIMEOUT_MODE` affects the behavior of [LJM_eStreamRead](#).

For more LJM configurations, see [Library Configuration Functions](#).

LJM_STREAM_RECEIVE_TIMEOUT_MS

Summary

Manually sets the timeout for LJM's stream data collection. Writing to this configuration sets `LJM_STREAM_RECEIVE_TIMEOUT_MODE` to `LJM_STREAM_RECEIVE_TIMEOUT_MODE_MANUAL`.

Writing a non-zero value to `LJM_STREAM_RECEIVE_TIMEOUT_MS` will manually set the timeout for LJM's stream data collection. Note that using `LJM_STREAM_RECEIVE_TIMEOUT_MODE` of `LJM_STREAM_RECEIVE_TIMEOUT_MODE_CALCULATED` is almost always better than using manual non-zero `LJM_STREAM_RECEIVE_TIMEOUT_MS` value.

Writing 0 to `LJM_STREAM_RECEIVE_TIMEOUT_MS` will cause LJM to never time out. This allows you to set up triggered stream or externally clocked stream on the T7. This is usually used in conjunction with [LJM_STREAM_SCANS_RETURN](#) set to `LJM_STREAM_SCANS_RETURN_ALL_OR_NONE`.

The constant `LJM_STREAM_RECEIVE_TIMEOUT_MS` can be used interchangeably with the string "LJM_STREAM_RECEIVE_TIMEOUT_MS".

Remarks

`LJM_STREAM_RECEIVE_TIMEOUT_MS` does not affect currently running or already initialized streams.

Relevant Functions

To read `LJM_STREAM_RECEIVE_TIMEOUT_MS`, use [LJM_ReadLibraryConfigS](#).

To write `LJM_STREAM_RECEIVE_TIMEOUT_MS`, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

`LJM_STREAM_RECEIVE_TIMEOUT_MS` affects the behavior of [LJM_eStreamRead](#).

Example

[C/C++] Set the `LJM_STREAM_RECEIVE_TIMEOUT_MS` mode to infinite.

```
char ErrorString[LJM_MAX_NAME_SIZE];
int LJMErr = 0;
LJMErr = LJM_WriteLibraryConfigS(
    LJM_STREAM_RECEIVE_TIMEOUT_MS,
```

```

0
);
if (LJMEError != LJME_NOERROR) {
    LJM_ErrorToString(LJMEError, ErrorString);
    printf("LJM_WriteLibraryConfigS error: %s\n", ErrorString);
}

```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_STREAM_SCANS_RETURN

Summary

Sets how [LJM_eStreamRead](#) will return data.

Below, `ScansPerRead` is a parameter of [LJM_eStreamStart](#).

LJM_STREAM_SCANS_RETURN_ALL = 1 (default)

- A mode that will cause [LJM_eStreamRead](#) to sleep until the full `ScansPerRead` scans are collected by LJM.
- This mode may not be appropriate for stream types that are not consistently timed, such as externally clocked stream mode.

LJM_STREAM_SCANS_RETURN_ALL_OR_NONE = 2

- A mode that will cause [LJM_eStreamRead](#) to never sleep, and instead either:
 - consume `ScansPerRead` scans and return `LJME_NOERROR`, or
 - consume no scans and return `LJME_NO_SCANS_RETURNED`.

The constant `LJM_STREAM_SCANS_RETURN` can be used interchangeably with the string "LJM_STREAM_SCANS_RETURN".

Remarks

`LJM_STREAM_SCANS_RETURN` does not affect currently running or already initialized streams.

Relevant Functions

To read `LJM_STREAM_SCANS_RETURN`, use [LJM_ReadLibraryConfigS](#).

To write `LJM_STREAM_SCANS_RETURN`, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

`LJM_STREAM_SCANS_RETURN` affects the behavior of [LJM_eStreamRead](#).

Example

[C/C++] Set the `LJM_STREAM_SCANS_RETURN` mode to `LJM_STREAM_SCANS_RETURN_ALL_OR_NONE`.

```

char ErrorString[LJM_MAX_NAME_SIZE];
int LJMEError = 0;
LJMEError = LJM_WriteLibraryConfigS(
    LJM_STREAM_SCANS_RETURN,
    LJM_STREAM_SCANS_RETURN_ALL_OR_NONE
);
if (LJMEError != 0) {
    LJM_ErrorToString(LJMEError, ErrorString);
    printf("LJM_WriteLibraryConfigS error: %s\n", ErrorString);
}

```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE

Summary

Requires LJM 1.2000 or later

Sets the TCP receive buffer size for LJM's stream data collection. This can reduce the frequency of [auto-recovery](#). Behavior varies by platform.

For a background on the receive buffer, see [LJM_GetStreamTCPReceiveBufferStatus](#).

Writing 0 to `LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE` will use the OS default receive buffer size, which probably includes auto-tuning. This is appropriate for most use cases. The default `LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE` is 0.

Writing a non-zero value to `LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE` will manually set the receive buffer size. Internally, `setsockopt` is called with `SO_RCVBUF` as the option parameter.

Since behavior for setting the receive buffer size varies by platform, you should try out different values of `LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE` until you find the maximum. For example, you should start with a small value like 65535 (which doesn't require window scaling) and exponentially increase it until an

error occurs. Your test program should set `LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE`, then start stream and call `LJM_GetStreamTCPReceiveBufferStatus`, ensuring that the `ReceiveBufferBytesSize` is equivalent to your intended stream receive buffer size.

You may also wish to use Wireshark to capture packets to see what the maximum window size your host computer reports the window size to be. To do so, you can start a new capture with a filter like `ip.addr == 192.168.1.207`, where 192.168.1.207 is your LabJack's IP address, then find the first packet addressed to 192.168.1.207 with port 702. The window size and window scaling of that packet determines the window size as reported to the LabJack device.

Here are some known platform differences:

- A Windows 10 computer was shown to be able to increase the receive buffer size until about 2147483647, which is the maximum number that Windows' `recv` function can take as the `len` parameter (due to the `len` parameter being a signed 32-bit integer). After this point, LJM will return `LJME_NEGATIVE_RECEIVE_BUFFER_SIZE` (errorcode 1321).
- A macOS computer was shown to be able to increase the receive buffer size until about 7400000, when `LJM_eStreamStart` returns `LJME_SOCKET_LEVEL_ERROR`.
- Linux may accept any receive buffer size and silently reduce it to adhere to system configurations.

The constant `LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE` can be used interchangeably with the string "`LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE`".

Remarks

`LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE` does not affect currently running or already initialized streams.

Relevant Functions

To read `LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE`, use `LJM_ReadLibraryConfigS`.

To write `LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE`, use `LJM_WriteLibraryConfigS` or `LJM_LoadConfigurationFile`.

`LJM_STREAM_TCP_RECEIVE_BUFFER_SIZE` affects the behavior of `LJM_eStreamStart`.

For more LJM configurations, see [Library Configuration Functions](#).

LJM_STREAM_TRANSFERS_PER_SECOND

Summary

`LJM_STREAM_TRANSFERS_PER_SECOND` is a numerical readable and writable LJM library configuration which sets how many times per second LJM's stream thread attempts to receive stream packet(s) from the device.

The constant `LJM_STREAM_TRANSFERS_PER_SECOND` can be used interchangeably with the string "`LJM_STREAM_TRANSFERS_PER_SECOND`".

Details

`LJM_STREAM_TRANSFERS_PER_SECOND` is an advanced configuration. Modifying it is not recommended for most users.

Increasing `LJM_STREAM_TRANSFERS_PER_SECOND` to a higher value will allow for less latency in receiving stream data, since LJM will receive data from the device more often (in smaller quantities). Increasing the `LJM_STREAM_TRANSFERS_PER_SECOND` too much will cause the data contained in each packet to decrease, which will cause unnecessary overhead and potentially cause scans to be missed.

You can enable [debug logging](#) with `LJM_TRACE` as the `LJM_DEBUG_LOG_LEVEL` to see how stream is set up. After running a test stream, search the log file for the term "streaming initialized" to see what effects `LJM_STREAM_TRANSFERS_PER_SECOND` has on stream setup. Make sure to disable debug logging for your real stream sessions after finding settings for stream setup that looks reasonable, since debug logging consumes a significant amount of processing. LJM's stream setup is deterministic, so given the same parameters to `LJM_eStreamStart` and the same stream configurations, stream will be set up the same way every time.

Decreasing `LJM_STREAM_TRANSFERS_PER_SECOND` to a lower value will theoretically cause less overhead in stream, since LJM will receive data from the device less often (in larger quantities), but may have little to no effect. When decreasing `LJM_STREAM_TRANSFERS_PER_SECOND`, it is recommended to increase the size of the stream buffer on the device. See the [Modbus](#) register `STREAM_BUFFER_SIZE_BYTES` (address: 4012).

Setting a new `LJM_STREAM_TRANSFERS_PER_SECOND` value will not affect stream threads that are already in progress.

Relevant Functions

To read `LJM_STREAM_TRANSFERS_PER_SECOND`, use `LJM_ReadLibraryConfigS`.

To write `LJM_STREAM_TRANSFERS_PER_SECOND`, use `LJM_WriteLibraryConfigS` or `LJM_LoadConfigurationFile`.

Example

[C/C++] Read `LJM_STREAM_TRANSFERS_PER_SECOND` then set it to 100.

```
char ErrorString[LJM_MAX_NAME_SIZE];
double value;
```

```
int LJMErr = LJM_ReadLibraryConfigS(LJM_STREAM_TRANSFERS_PER_SECOND, &value);
if (LJMErr != 0) {
```

```

LJM_ErrorToString(LJMEError, ErrorString);
printf("LJM_ReadLibraryConfigS error: %s\n", ErrorString);
}
printf("The default for LJM_STREAM_TRANSFERS_PER_SECOND is %.00f\n", value);

value = 100;
printf("Setting LJM_STREAM_TRANSFERS_PER_SECOND to %.00f\n", value);
LJMEError = LJM_WriteLibraryConfigS(LJM_STREAM_TRANSFERS_PER_SECOND, value);
if (LJMEError != 0) {
    LJM_ErrorToString(LJMEError, ErrorString);
    printf("LJM_WriteLibraryConfigS error: %s\n", ErrorString);
}

```

Possible output:

```

The default for LJM_STREAM_TRANSFERS_PER_SECOND is 25
Setting LJM_STREAM_TRANSFERS_PER_SECOND to 100

```

For more LJM configurations, see [Library Configuration Functions](#).

LJM_USE_TCP_INIT_FOR_T7_WIFI_TCP

Summary

LJM_USE_TCP_INIT_FOR_T7_WIFI_TCP is a numerical readable-writable LJM library configuration that sets whether LJM will use UDP or TCP for T7 WiFi connection initialization when ConnectionType is TCP.

The constant LJM_USE_TCP_INIT_FOR_T7_WIFI_TCP can be used interchangeably with the string "LJM_USE_TCP_INIT_FOR_T7_WIFI_TCP".

LJM_USE_TCP_INIT_FOR_T7_WIFI_TCP was added in LJM 1.1500.

Details

0 - false/disabled - Use UDP

1 - true/enabled (Default) - Use TCP

Relevant Functions

To read LJM_USE_TCP_INIT_FOR_T7_WIFI_TCP, use [LJM_ReadLibraryConfigS](#).

To write LJM_USE_TCP_INIT_FOR_T7_WIFI_TCP, use [LJM_WriteLibraryConfigS](#) or [LJM_LoadConfigurationFile](#).

For more LJM configurations, see [Library Configuration Functions](#).

4 - Error Codes

[Add new comment](#)

Error Codes Overview

Error codes from the ljm_constants.json file. For more information about error codes look at the [installed error codes](#) page.

Error Code	Error Name	Description
0	LJ_SUCCESS	
200	LJME_WARNINGS_BEGIN	This indicates where the warning codes start is not a real warning code.
399	LJME_WARNINGS_END	This indicates where the warning codes end is not a real warning code.
201	LJME_FRAMES_OMITTED_DUE_TO_PACKET_SIZE	Some read/write operation(s) were not sent to the device because the Feedback command being created was too large for the device, given the current connection type.
202	LJME_DEBUG_LOG_FAILURE	The debug log file could not be written to. Check the LJM_DEBUG_LOG_FILE and related LJM configurations.
203	LJME_USING_DEFAULT_CALIBRATION	The device's calibration could not be read, so the nominal calibration values were used. Possible Cause: Incompatible microSD card installed. See SD Card section of T7 Datasheet.
		The message could not be logged because the debug log file has not been

204	LJME_DEBUG_LOG_FILE_NOT_OPEN	opened.
254	FEATURE_NOT_IMPLEMENTED	
1200	LJME_MODBUS_ERRORS_BEGIN	This indicates where the Modbus error codes start is not a real Modbus error code.
1216	LJME_MODBUS_ERRORS_END	This indicates where the Modbus error codes end is not a real Modbus error code.
1201	LJME_MBE1_ILLEGAL_FUNCTION	The device received an invalid function code.
1202	LJME_MBE2_ILLEGAL_DATA_ADDRESS	The device received an invalid address.
1203	LJME_MBE3_ILLEGAL_DATA_VALUE	The device received a value that could not be written to the specified address.
1204	LJME_MBE4_SLAVE_DEVICE_FAILURE	The device encountered an error.
1205	LJME_MBE5_ACKNOWLEDGE	The device acknowledges the request, but will take some time to process.
1206	LJME_MBE6_SLAVE_DEVICE_BUSY	The device is busy and cannot respond currently.
1208	LJME_MBE8_MEMORY_PARITY_ERROR	The device detected a memory parity error.
1210	LJME_MBE10_GATEWAY_PATH_UNAVAILABLE	The requested route was not available.
1211	LJME_MBE11_GATEWAY_TARGET_NO_RESPONSE	The requested route was available but the device failed to respond.
1220	LJME_LIBRARY_ERRORS_BEGIN	This indicates where the LJM error codes start is not a real LJM error code.
1399	LJME_LIBRARY_ERRORS_END	This indicates where the LJM error codes end is not a real LJM error code.
1221	LJME_UNKNOWN_ERROR	LJM reached an unexpected state. Please contact LabJack support if you've received this error.
1222	LJME_INVALID_DEVICE_TYPE	LJM received an unexpected device type.
1223	LJME_INVALID_HANDLE	LJM reached an invalid state.
1224	LJME_DEVICE_NOT_OPEN	The requested handle did not refer to an open device. Perhaps the Open call failed but you still tried to use the returned handle. Perhaps your code called Close somewhere and then after tried to use the handle.
1225	LJME_STREAM_NOT_INITIALIZED	The requested device does not have a stream initialized in LJM.
1226	LJME_DEVICE_DISCONNECTED	The device could not be contacted and LJM is not configured to heal device connections (see LJM config LJME_HEAL_CONNECTION_MODE).
1227	LJME_DEVICE_NOT_FOUND	A device matching the requested device parameters could not be found and therefore could not be opened.
1229	LJME_DEVICE_ALREADY_OPEN	The device being registered was already registered. Please contact LabJack support if you've received this error.
1230	LJME_DEVICE_CURRENTLY_CLAIMED_BY_ANOTHER_PROCESS	At least one matching device was found, but cannot be claimed by this process because a different process has already claimed it. Either close the device handle in your other process (i.e. program or application) or close the other process. Then try opening the device again.
1231	LJME_CANNOT_CONNECT	At least one device matching the requested device parameters was found, but a connection could not be established.
1233	LJME_SOCKET_LEVEL_ERROR	A TCP socket-level error occurred.
1236	LJME_CANNOT_OPEN_DEVICE	The device could not be opened.
1237	LJME_CANNOT_DISCONNECT	An error occurred while attempting to disconnect.
1238	LJME_WINSOCK_FAILURE	A Windows WINSOCK error occurred.
1239	LJME_RECONNECT_FAILED	The device could not be contacted. LJM is configured to heal device connections and the reconnection attempt failed.
1243	LJME_U3_NOT_SUPPORTED_BY_LJM	LJM does not support UD-series LabJack devices; use the UD driver (↗) https://labjack.com/support/ud

1246	LJME_U6_NOT_SUPPORTED_BY_LJM	LJM does not support UD-series LabJack devices; use the UD driver (🔗) https://labjack.com/support/ud
1249	LJME_UE9_NOT_SUPPORTED_BY_LJM	LJM does not support UD-series LabJack devices; use the UD driver (🔗) https://labjack.com/support/ud
1250	LJME_INVALID_ADDRESS	The requested address was not found.
1251	LJME_INVALID_CONNECTION_TYPE	LJM received an unexpected connection type.
1252	LJME_INVALID_DIRECTION	LJM received an unexpected read/write direction.
1253	LJME_INVALID_FUNCTION	LJM received an unexpected Modbus function.
1254	LJME_INVALID_NUM_REGISTERS	LJM received an unexpected number of registers.
1255	LJME_INVALID_PARAMETER	LJM received an unexpected parameter. Please check the LabJackM.h description of the function that returned this error for an explanation.
1256	LJME_INVALID_PROTOCOL_ID	LJM received an unexpected Modbus protocol ID.
1257	LJME_INVALID_TRANSACTION_ID	LJM received an unexpected Modbus transaction ID.
1259	LJME_UNKNOWN_VALUE_TYPE	LJM received an unexpected data type.
1260	LJME_MEMORY_ALLOCATION_FAILURE	LJM was unable to allocate needed memory.
1261	LJME_NO_COMMAND_BYTES_SENT	No Modbus command bytes could be sent to the device.
1262	LJME_INCORRECT_NUM_COMMAND_BYTES_SENT	An incorrect number of Modbus command bytes were sent to the device, as reported by the underlying driver.
1263	LJME_NO_RESPONSE_BYTES_RECEIVED	No Modbus response bytes could be received from the device.
1264	LJME_INCORRECT_NUM_RESPONSE_BYTES_RECEIVED	An incorrect/unexpected number of Modbus response bytes were received from the device.
1265	LJME_MIXED_FORMAT_IP_ADDRESS	LJM received an IP identifier that seemed to be formatted in both hexadecimal and decimal.
1266	LJME_UNKNOWN_IDENTIFIER	LJM received an unknown identifier type.
1267	LJME_NOT_IMPLEMENTED	An unimplemented functionality was attempted. Please contact LabJack support if you've received this error so we can implement it for you!
1268	LJME_INVALID_INDEX	LJM received an index/offset that was out-of-bounds. Please contact LabJack support if you've received this error.
1269	LJME_INVALID_LENGTH	LJM received a length that did not make sense. Please contact LabJack support if you've received this error.
1270	LJME_ERROR_BIT_SET	A Modbus response had the error bit set. Please contact LabJack support if you've received this error.
1271	LJME_INVALID_MAXBYTESPERMBFB	LJM_AddressesToMBFB received a value of MaxBytesPerMBFB that was too small.
1272	LJME_NULL_POINTER	LJM received a null pointer for a pointer parameter that is required to not be null.
1273	LJME_NULL_OBJ	LJM found a null object. Please contact LabJack support if you've received this error.
1274	LJME_RESERVED_NAME	The requested name was invalid/reserved.
1275	LJME_UNPARSABLE_DEVICE_TYPE	LJM received a device type string that was unparseable or unexpected.
1276	LJME_UNPARSABLE_CONNECTION_TYPE	LJM received a connection type string that was unparseable or unexpected.
1277	LJME_UNPARSABLE_IDENTIFIER	LJM received a identifier string that was unparseable or unexpected.
1278	LJME_PACKET_SIZE_TOO_LARGE	The Modbus packet to be sent or the expected Modbus packet to be received exceeded the maximum packet size for the device connection.
1279	LJME_TRANSACTION_ID_ERR	LJM received a mismatched Modbus transaction ID from the device.
1280	LJME_PROTOCOL_ID_ERR	LJM received a mismatched Modbus protocol ID from the device.

1281	LJME_LENGTH_ERR	LJM received a mismatched Modbus packet length from the device.
1282	LJME_UNIT_ID_ERR	LJM received a mismatched Modbus unit ID from the device.
1283	LJME_FUNCTION_ERR	LJM received a mismatched Modbus function ID from the device.
1284	LJME_STARTING_REG_ERR	LJM received a mismatched Modbus register address from the device.
1285	LJME_NUM_REGS_ERR	LJM received a mismatched Modbus number of registers from the device.
1286	LJME_NUM_BYTES_ERR	LJM received a mismatched Modbus number of bytes from the device.
1289	LJME_CONFIG_FILE_NOT_FOUND	The LJM configuration file could not be opened for reading.
1290	LJME_CONFIG_PARSING_ERROR	An error occurred while parsing the LJM configuration file.
1291	LJME_INVALID_NUM_VALUES	LJM received an invalid number of values.
1292	LJME_CONSTANTS_FILE_NOT_FOUND	The LJM error constants file and/or Modbus constants file could not be opened for reading.
1293	LJME_INVALID_CONSTANTS_FILE	The LJM error constants file and/or Modbus constants file could not be parsed.
1294	LJME_INVALID_NAME	The requested name was not found in the register loaded from the constants file (if the constants file is loaded).
1296	LJME_OVERSPECIFIED_PORT	The requested connection type is ANY and port is specified.
1297	LJME_INTENT_NOT_READY	The internal operation object was not prepared correctly. Please contact LabJack support if you've received this error.
1298	LJME_ATTR_LOAD_COMM_FAILURE	A device matching the requested device parameters was found, claimed, and connected, but a communication error prevented device attributes from being loaded to LJM. The device was not opened.
1299	LJME_INVALID_CONFIG_NAME	LJM received an unknown configuration name.
1300	LJME_ERROR_RETRIEVAL_FAILURE	An error occurred on the device and LJM was unable to retrieve more information about that error. Please contact LabJack support if you've received this error.
1301	LJME_LJM_BUFFER_FULL	The LJM stream buffer was filled with stream data and stream was stopped.
1302	LJME_COULD_NOT_START_STREAM	LJM could not start stream. Input may have incorrect or a different error may have occurred.
1303	LJME_STREAM_NOT_RUNNING	LJM is not streaming data from the device.
1304	LJME_UNABLE_TO_STOP_STREAM	The stream could not be stopped.
1305	LJME_INVALID_VALUE	LJM received a value that was not expected.
1306	LJME_SYNCHRONIZATION_TIMEOUT	LJM did not receive data from the device for longer than the timeout specified by LJME_STREAM_RECEIVE_TIMEOUT_MS. Externally clocked stream setup is described here: http://labjack.com/support/software/api/ljm/function-reference/ljmeStreamStart#externally-clocked
1307	LJME_OLD_FIRMWARE	The current firmware version of the device was not sufficient to read/write a requested register, as according to the loaded LJM constants file.
1308	LJME_CANNOT_READ_OUT_ONLY_STREAM	The stream running is out-only and does not produce data values.
1309	LJME_NO_SCANS_RETURNED	The LJM configuration LJME_STREAM_SCANS_RETURN is set to LJME_STREAM_SCANS_RETURN_ALL_OR_NONE and the full ScansPerRead scans have not be received.
1310	LJME_TEMPERATURE_OUT_OF_RANGE	LJM_TCVoltsToTemp received a temperature that was out of range for the given thermocouple type.
1311	LJME_VOLTAGE_OUT_OF_RANGE	LJM_TCVoltsToTemp received a voltage that was out of range for the given thermocouple type.
1312	LJME_FUNCTION_DOES_NOT_SUPPORT_THIS_TYPE	The function does not support the given data type. For example, LJM_eReadName and LJM_eReadAddress do not support reading

		LJM_STRING values, which are too large.
1313	LJME_INVALID_INFO_HANDLE	The given info handle did not exist. (It may have been passed to LJME_CleanInfo already.)
1314	LJME_NO_DEVICES_FOUND	An Open/OpenS call was called - with any device type, any connection type, and any identifier - but no devices were found.
1316	LJME_AUTO_IPS_FILE_NOT_FOUND	The LJM auto IPs file could not be found or read. LJM will attempt to create the auto IPs file, as needed.
1317	LJME_AUTO_IPS_FILE_INVALID	The LJM auto IPs file contents were not valid.
1318	LJME_INVALID_INTERVAL_HANDLE	The given interval handle did not exist.
1319	LJME_NAMED_MUTEX_PERMISSION_DENIED	There was a permission error while accessing a system-level named mutex. Try restarting your computer.
2300	MODBUS_RSP_OVERFLOW	Response packet greater than max packet size.
2301	MODBUS_CMD_OVERFLOW	Command packet greater than max packet size
2310	MODBUS_STRING_CMD_TOO_BIG	
2311	MODBUS_STRING_PARAM_TOO_BIG	
2312	MODBUS_STRING_BAD_NUM_PARAMS	
2313	MODBUS_INVALID_NUM_REGISTERS	Register data types does not match the number of registers in the request.
2314	MODBUS_READ_TOO_LARGE	
2315	MODBUS_NUM_REGS_MUST_BE_EVEN	Register or group of registers requires the access be in an even number of registers.
2316	MODBUS_STRING_MISSING_NULL	Strings must be terminated with a null (\0, 0x00).
2330	STARTUP_CONFIG_INVALID_CODE	
2331	STARTUP_CONFIG_INVALID_READ	An attempt was made to read beyond the configuration structure.
2340	USER_RAM_FIFO_MUST_BE_EMPTY	The FIFO can not contain any data when data size is being changed.
2343	USER_RAM_FIFO_INSUFFICIENT_VALUES	The FIFO contains fewer values than requested.
2344	USER_RAM_FIFO_INSUFFICIENT_SPACE	FIFO does not have enough free space to hold the requested write. No data was added to the FIFO.
2345	USER_RAM_FIFO_SIZE_MUST_BE_EVEN	The number of bytes allocated to the FIFO must be even..
2350	INTFLASH_ADD_INVALID	
2351	INTFLASH_CODE_INVALID	
2352	INTFLASH_OP_PROHIBITED	Attempted to read or write a section that is not allowed.
2353	INTFLASH_SECTION_OVERWRITE	Attempted to write or read beyond the currently selected section.
2354	INTFLASH_KEY_INVALID	Specified Key and Address mismatch.
2355	FLASH_VERIFICATION_FAILED	A write to flash failed to set one or more bits to the desired values.
2356	FLASH_ERASE_FAILED	One or more bits failed to set during a flash erase operation.
2358	INTFLASH_UNAVAILABLE	Flash can not be accessed due to the WiFi module booting up.
2359	FILEIO_UNAVAILABLE	The file system can not be accessed due to the WiFi module booting up.
2370	AIN_RANGE_INVALID	Specified range not available on this device.
2371	AIN_SETTLING_INVALID	Specified settling is greater than device max.
2372	AIN_SET_TO_BINARY	Analog input system currently set to binary. Some operations, such as AIN_EF, will fail.
2373	AIN_NEGATIVE_CHANNEL_INVALID	For channel range 0-13: Negative channel must be even channel number + 1. For extended channel range 48-127: Negative channel must be channel number + 8.

2374	AIN_ALL_ZERO_ONLY	Only a value of zero may be written to this address.
2375	AIN_RESOLUTION_INVALID	Selected resolution is invalid. Valid range is 0-5 for T4 and 0-12 for T7.
2380	LUA_VM_STATE_NO_CHANGE	
2381	LUA_INITIALIZATION_ERROR	
2382	LUA_INVALID_NUM_IO_FLOATS	Requested more than the max possible number of IO floats.
2383	LUA_IO_FLOATMEM_OVERFLOW	Attempt to read/write beyond the currently allocated IO space.
2384	LUA_INVALID_MODE	
2385	LUA_IS_RUNNING	A running script is preventing the requested operation.
2386	LUA_CODE_BUFFER_EMPTY	Attempted to run a program that is not present.
2387	LUA_DEBUG_IS_DISABLED	Attempted to read from debug buffer while debug is disabled.
2388	LUA_TABLE_SMALLER_THAN_SPECIFIED_SIZE	The Lua table provided is too small to process the request.
2389	LUA_IS_CLOSING	The Lua VM is being closed, usually takes less than 100 ms.
2400	SYSTEM_MEMORY_BEREFT	Insufficient memory to perform the requested action.
2401	SYSTEM_MEMORY_OVERWRITE	Attempted to overwrite a buffer.
2402	SYSTEM_REBOOT_CODE_INVALID	Invalid code supplied when issuing a reboot.
2403	SYSTEM_READ_OVERRUN	
2404	SYSTEM_INVALID_PIN	
2405	SYSTEM_NVRAM_UNAVAILABLE	NVRAM is not available on this device.
2406	SYSTEM_NVRAM_INVALID_ADDRESS	The specified NVRAM location is not available on this device.
2407	SYSTEM_WAIT_TOO_LONG	The requested wait time is beyond the max allowed.
2408	SYSTEM_INCOMPATIBLE_FIRMWARE_VERSION	The firmware image is not compatible with this device.
2420	DEVICE_NAME_MUST_BE_ALPHANUM	Attempted to write a device name with invalid characters.
2450	POWER_INVALID_SETTING	Unknown value specified.
2451	POWER_USB_NEEDS_20MHZ_OR_MORE	Core must be running at 20MHz minimum for USB to operate.
2452	POWER_NO_CHANGE	
2456	POWER_CAN_NOT_CHANGE_USED_CONNECTION	Can not change the power level of the connected medium.
2460	POWER_ANALOG_OFF	Analog input system is powered down.
2490	WIFI_NOT_ASSOCIATED	WiFi needs to be connected to a network before the requested action can be performed.
2500	HW_DIO_NOT_AVAILABLE	
2501	DIO_SET_TO_ANALOG	The digital line addressed is set to analog. Digital operations can not be performed.
2508	HW_CNTRA_NOT_AVAILABLE	Counter A is being used by another system.
2509	HW_CNTRB_NOT_AVAILABLE	Counter B is being used by another system.
2510	HW_CNTRC_NOT_AVAILABLE	Counter C is being used by another system.
2511	HW_CNTRD_NOT_AVAILABLE	Counter D is being used by another system.
2520	HW_CIO0_NOT_AVAILABLE	
2521	HW_CIO1_NOT_AVAILABLE	
2523	HW_DAC1_NOT_AVAILABLE	
2550	EF_DIO_HAS_NO_TNC_FEATURES	
2551	EF_INVALID_TYPE	The selected type is not recognized.

2552	EF_TYPE_NOT_SUPPORTED	The selected type is not recognized.
2553	EF_PIN_TYPE_MISMATCH	The requested type is not supported on this DIO pin.
2554	EF_CLOCK_SOURCE_NOT_ENABLED	Attempted to disable a clock source that is not running.
2555	EF_32BIT_DATA_INTO_16BIT_REG	A number greater than 16-bits was written to a clock source configured for 16-bits.
2556	EF_SET_TO_32BIT	
2557	EF_SMOOTH_VALUE_OUT_OF_RANGE	
2558	EF_32BIT_REQUIRES_BOTH_CNT0and1	Both counter 1 and counter 2 must be disabled before enabling counter 0.
2559	EF_PRESCALE_VALUE_INVALID	Specified prescale value is not supported. Supported values are 1,2,4,8,16,32,64, and 256
2560	EF_PIN_RESERVED	Pin is already used by another system.
2561	EF_INVALID_DIO_NUMBER	DIO addresses is not supported on this device.
2563	EF_LINE_MUST_BE_LOW_BEFORE_STARTING	The DIO line must be set to output low to ensure proper signal generation.
2564	EF_INVALID_DIVISOR	
2565	EF_VALUE_GREATER_THAN_PERIOD	Value specified is greater than clock source roll value.
2566	EF_CAN_NOT_CHANGE_INDEX_WHILE_ENABLED	The index of a DIO_EF can not be changed while that DIO_EF is enabled.
2580	AIN_EF_INVALID_TYPE	The type index specified is not supported.
2581	AIN_EF_INVALID_NUM_SAMPLES	Too many samples specified.
2582	AIN_EF_CALCULATION_ERROR	
2583	AIN_EF_CHANNEL_INACTIVE	AIN_EF channel has not been initialized. To initialize, set the index to a non-zero value.
2584	AIN_EF_CALCULATION_OUT_OF_RANGE	
2585	AIN_EF_INVALID_CHANNEL	
2586	AIN_EF_INVALID_CJC_REGISTER	
2587	AIN_EF_STREAM_START_FAILURE	Could not start the data collection stream.
2588	AIN_EF_COULD_NOT_FIND_PERIOD	Failed to detect a period to perform calculations over.
2589	AIN_EF_MUST_BE_DIFFERENTIAL	The selected AIN must be set to differential.
2590	AIN_EF_SCAN_TIME_TOO_LONG	The data collection time (number of samples / scan_rate) is too big. Limit is set to 180 ms as of firmware 1.0170
2600	STREAM_NEED_AT_LEAST_ONE_CHN	The list of channels to stream is empty.
2601	STREAM_CLOCK_BASE_NOT_WRITABLE	Stream clock base is read only.
2602	STREAM_EXTCLK_AND_GATE_MX	
2603	STREAM_IN_SPONTANEOUS_MODE	Stream data can not be read with commands while in spontaneous mode.
2604	STREAM_USB_PKT_OVERFLOW	
2605	STREAM_IS_ACTIVE	The requested operation can not be performed while stream is running.
2606	STREAM_CONFIG_INVALID	Stream resolution can not be greater than 8.
2607	STREAM_CHN_LIST_INVALID	The channel list contains an unstreamable address.
2608	STREAM_SCAN_RATE_INVALID	The scan rate times the number of channels per scan is too great for this device.
2609	STREAM_OUT_BUFF_TOO_BIG	
2610	STREAM_OUT_NUM_INVALID	An invalid stream out number was specified.
2611	STREAM_DATA_TYPE_INVALID	An unsupported data types was specified.

2612	STREAM_TARGET_CONFIG_INVALID	Stream must be set to either spontaneous or command-response.
2613	STREAM_OUT_BUFF_FULL	Attempted to write more data than the buffer can hold. Extra data was discarded.
2614	STREAM_OUT_TARGET_INVALID	Specified address can not be a stream out target.
2615	STREAM_BUFF_SIZE_INVALID	Specified buffer was either too large or not a power of 2.
2616	STREAM_OUT_BUFF_LOOP_OVERWRITE	
2617	STREAM_OUT_BUFF_DNE	The buffer size must be set before data can be written to it.
2618	STREAM_SAMPLES_PER_PKT_INVALID	The specified number of samples per packet is too large.
2619	STREAM_BUFFER_DNE	
2620	STREAM_NOT_RUNNING	
2621	STREAM_SETTLING_INVALID	Specified settling time is greater than the max possible.
2622	STREAM_OUT_LOOP_TOO_BIG	Loop size too big for the current buffer size.
2623	STREAM_OUT_DATA_TRGT_MISMATCH	
2624	STREAM_INVALID_DIVISOR	Selected divisor can not be used.
2625	STREAM_CHN_CAN_NOT_BE_STREAMED	The requested channel can not be streamed.
2626	STREAM_OUT_DAC_IN_USE	The high resolution converter can not be used while stream out is used to update a DAC.
2670	SWDT_ROLLT_INVALID	
2671	SWDT_ENABLED	The watchdog must be disabled before the requested operation can be performed.
2672	SWDT_DIO_SETTINGS_INVALID	
2673	SWDT_DAC0_SETTINGS_INVALID	
2674	SWDT_DAC1_SETTINGS_INVALID	
2690	RTC_TIME_INVALID	
2691	RTC_SNTP_TIME_INVALID	
2692	RTC_NOT_PRESENT	The requested operation can not be performed on units without a real-time-clock.
2700	SPI_MODE_INVALID	Valid modes are 0-3.
2701	SPI_NO_DATA_AVAILABLE	
2702	SPI_CS_PIN_INVALID	Attempted to set an invalid pin.
2703	SPI_CLK_PIN_INVALID	Attempted to set an invalid pin.
2704	SPI_MISO_PIN_INVALID	Attempted to set an invalid pin.
2705	SPI_MOSI_PIN_INVALID	Attempted to set an invalid pin.
2706	SPI_CS_PIN_RESERVED	Selected pin is not available.
2707	SPI_CLK_PIN_RESERVED	Selected pin is not available.
2708	SPI_MISO_PIN_RESERVED	Selected pin is not available.
2709	SPI_MOSI_PIN_RESERVED	Selected pin is not available.
2710	SPI_TRANSFER_SIZE_TOO_LARGE	
2720	I2C_BUS_BUSY	One or both of the I2C lines are held low. Check hardware and reset the bus.
2721	I2C_NO_DATA_AVAILABLE	Attempted to read from an empty buffer.
2722	I2C_SDA_PIN_INVALID	Attempted to set an invalid pin.
2723	I2C_SCL_PIN_INVALID	Attempted to set an invalid pin.

2724	I2C_SDA_PIN_RESERVED	Selected pin is not available.
2725	I2C_SCL_PIN_RESERVED	Selected pin is not available.
2726	I2C_TX_SIZE_TOO_LARGE	
2727	I2C_RX_SIZE_TOO_LARGE	
2728	I2C_BUFFER_OVERRUN	
2729	I2C_SPEED_TOO_LOW	The throttle setting is too low, watchdog may fire. Minimum value = 46000
2740	SBUS_COMM_TIME_OUT	Slave device did not respond.
2741	SBUS_NO_ACK	Slave device did not acknowledge the data transfer.
2742	SBUS_CUSTOM_MODE_INVALID	
2743	SBUS_INVALID_DIO_NUM	Attempted to set an invalid pin.
2744	SBUS_BACKGROUND_SERVICE_ON	Command-response reads can not be used while the background service is running.
2745	SBUS_CHECKSUM_ERROR	SHT communication checksum failed.
2760	TDAC_SDA_SCL_INVALID	SCL must be even and SDA must be SCL+1.
2761	TDAC_SCL_INVALID	Attempted to set an invalid pin.
2762	TDAC_INVALID_CHANNEL	Specified channel not supported on this device.
2763	TDAC_CAL_READ_FAILURE	Failed to read TDAC calibration.
2764	TDAC_NOT_FOUND	The TDAC did not respond to communication attempts.
2770	ONEWIRE_UNSUPPORETD_FUNCTION	Unknown function specified.
2771	ONEWIRE_NO_PRESENCE_PULSE	Unable to detect any devices on the bus.
2780	ASYNCH_NUM_DATA_BITS_INVALID	The specified number of data bits is not supported.
2781	ASYNCH_NUM_TO_WRITE_INVALID	The number of bytes to send is invalid.
2782	ASYNCH_READ_BUFF_SIZE_INVALID	The specified buffer size is invalid. Max is 2048.
2783	ASYNCH_BAUD_TOO_HIGH	Baud rate too high for this device.
2784	ASYNCH_IS_ENABLED	Specified operation can not be performed while enabled.
2785	ASYNCH_IS_NOT_ENABLED	Specified operation can not be performed while disabled.
2786	ASYNCH_TX_BUFFER_FULL	Transmit buffer is full.
2787	ASYNCH_TX_TIMEOUT	Transmission timed out. Do not write more than 100 ms at a time.
2788	ASYNCH_BAUD_ZERO	Baud rate is zero. Please specify a baud rate.
2801	FILE_IO_DISK_ERROR	A hard error occurred in the low level disk I/O layer.
2802	FILE_IO_INTERNAL_ERROR	Assertion failed.
2803	FILE_IO_NOT_READY	The physical drive cannot work.
2804	FILE_IO_NO_FILE	Could not find the file.
2805	FILE_IO_NO_PATH	Could not find the path.
2806	FILE_IO_INVALID_NAME	The path name format is invalid.
2807	FILE_IO_DENIED	Access denied due to prohibited access or directory full.
2808	FILE_IO_EXIST	Access denied due to prohibited access.
2809	FILE_IO_INVALID_OBJECT	The file/directory object is invalid. Under the context of getting performing a "ls" command 2809 indicates that there are no more files.
2810	FILE_IO_WRITE_PROTECTED	The physical drive is write protected.
2811	FILE_IO_INVALID_DRIVE	The logical drive number is invalid.

2812	FILE_IO_NOT_ENABLED	The volume has no work area.
2813	FILE_IO_NO_FILESYSTEM	There is no valid FAT12, FAT16, or FAT32 volume.
2814	FILE_IO_MKFS_ABORTED	The f_mkfs() aborted due to any parameter error.
2815	FILE_IO_TIMEOUT	Could not get a grant to access the volume within defined period.
2816	FILE_IO_LOCKED	The operation is rejected according to the file sharing policy.
2817	FILE_IO_NOT_ENOUGH_CORE	LFN working buffer could not be allocated.
2818	FILE_IO_TOO_MANY_OPEN_FILES	Number of open files greater than allowable limit (files > _FS_SHARE).
2819	FILE_IO_INVALID_PARAMETER	Given parameter is invalid.
2900	WIFI_ASSOCIATED	
2901	WIFI_ASSOCIATING	
2902	WIFI_ASSOCIATION_FAILED	
2903	WIFI_UNPOWERED	
2904	WIFI_BOOTING_UP	
2905	WIFI_COULD_NOT_START	
2906	WIFI_APPLYING_SETTINGS	
2907	WIFI_DHCP_STARTED	
2909	WIFI_OTHER	
2920	WIFI_UPDATE_CONFIG	
2921	WIFI_UPDATE_IN_PROG	
2923	WIFI_UPDATE_REBOOT	
2924	WIFI_UPDATE_SUCCESS	
2925	WIFI_UPDATE_FAILED	
2940	STREAM_AUTO_RECOVER_ACTIVE	
2941	STREAM_AUTO_RECOVER_END	
2942	STREAM_SCAN_OVERLAP	A new scan started before the previous scan finished. Generally occurs because ScanRate > MaxSampleRate/NumChannels. Note that MaxSampleRate is impacted by Range, ResolutionIndex, and Settling. Try adding commands right before StreamStart to set AIN_ALL_RANGE=10, STREAM_RESOLUTION_INDEX=0, and STREAM_SETTLING_US=0.
2943	STREAM_AUTO_RECOVER_END_OVERFLOW	
2944	STREAM_BURST_COMPLETE	
2945	STREAM_BUFFER_FULL	Stream buffer filled up while autorecover was disabled. Stream stopped.
2950	SELFDIAG_MAIN_OSC_FAIL	
2960	FILE_IO_NOT_FOUND	The requested file was not found.
2961	FILE_IO_NO_DISK	No SD card present or SC card could not be initialized.
2962	FILE_IO_INVALID_NAME	The file name is invalid.
2963	FILE_IO_FILE_NOT_OPEN	An open file is required to perform the requested operation.
2964	FILE_IO_TOO_MANY_OPEN	There are too many files open.
2965	FILE_IO_SD_CARD_NOT_FOUND	Failed to mount the SD card. Card may be bad or incompatible.
2966	FILE_IO_END_OF_CWD	There are no more files in the current working directory.

Support - Long Table Styling

Installed Error Codes

Error codes can easily be interpreted into near-English error strings using the [LJM_ErrorToString](#) function.

Check your version of LJM's LabJackM.h for LJM error codes, or check `ljm_constants.json` for LJM and device errors codes.

LabJackM.h locations:

- Windows
 - C:\Program Files\LabJack\Drivers\LabJackM.h
 - - or -
 - C:\Program Files (x86)\LabJack\Drivers\LabJackM.h
- Mac OS X
 - /usr/local/include/LabJackM.h
- Linux
 - /usr/local/include/LabJackM.h

ljm_constants.json locations:

- Windows Vista and later
 - C:\ProgramData\LabJack\LJM\ljm_constants.json
- Windows XP
 - C:\Documents and Settings\All Users\Application Data\LabJack\LJM\ljm_constants.json
- Mac OS X
 - /usr/local/share/LabJack/LJM/ljm_constants.json
- Linux
 - /usr/local/share/LabJack/LJM/ljm_constants.json

5 - Troubleshooting / FAQ

[Add new comment](#)

For other questions, issues, or comments, please [contact us](#).

5.1 - Does LJM handle signals?

Does LJM handle signals?

LJM catches all terminal signals, by default, in order to clean up LabJack device connections. LJM's signal handlers do the following:

- Ends all device streaming and close all device connections
- Optionally log which signal was received at LJM_FATAL priority
- Essentially exit the processes by setting the signal handler to default and re-raising the signal

LJM attempts to set the signal handlers upon the first call to LJM that attempts communication. (LJM_CloseAll is a quick way to force LJM to initialize the signal handlers.)

On **Windows**, signals are handled using [signal](#). The signals that are handled are:

- SIGINT
- SIGILL
- SIGABRT
- SIGFPE
- SIGSEGV
- SIGTERM

On **Linux/macOS**, LJM uses the oldact parameter of [sigaction](#) to determine if there was previously a signal handler; if oldact is not NULL for a given signal, LJM will reset oldact as the signal handler and not handle that signal. The default signals that are handled are:

- SIGHUP
- SIGINT
- SIGQUIT
- SIGILL
- SIGABRT

- SIGFPE
- SIGBUS
- SIGSEGV
- SIGTERM
- SIGTSTP
- SIGPIPE

Custom Cleanup on Linux/macOS: If your application needs to clean up resources, you can call LJM's signal handler from your own signal handler. To do this, force LJM to initialize signal handlers by calling `LJM_CloseAll`, then call `sigaction` to set your handler, and finally set the previous signal handler to be called in your handler. A Linux/macOS C++ example to catch Ctrl-C follows:

```
...
#include <signal.h>
#include <errno.h>

static volatile void (*ljm_handler)(int) = NULL;

void my_handler(int sig)
{
    // printf may not be re-entrant: because of this, it typically should not be
    // used in a signal handler. It is used here for ease of exposition.
    printf("my_handler is handling signal: %d\n", sig);

    // Perform custom cleanup behavior here
    ...

    if (ljm_handler != NULL) {
        printf("Calling ljm_handler from my_handler...\n");
        ljm_handler(sig);
    }
    exit(-1);
}

int main()
{
    // Initialize the LJM signal handler by "attempting" device communication
    // using LJM_CloseAll - we can ignore the return value. Other functions,
    // such as LJM_Open or LJM_ListAll work as well.
    LJM_CloseAll();

    struct sigaction act, oldact;
    act.sa_handler = my_handler;
    act.sa_flags = 0;
    int error = sigaction(SIGINT, &act, &oldact);
    if (error) {
        printf("There was an error during sigaction: %d\n", errno);
        exit(errno);
    }

    printf("oldact.sa_handler: %p\n", oldact.sa_handler);
    ljm_handler = (volatile void (*)(int))oldact.sa_handler;

    ...
}
```

Note that `sigprocmask` can be used to block signals while you are calling `sigaction`. Also note that other signal behaviors can be implemented using `sigaction`. See the man-pages for `sigaction` and `sigprocmask` for more details.

Custom Cleanup for ctrl+c on Windows: If your application needs to clean up resources, you can use `SetConsoleCtrlHandler` to add a handler routine. Do this after LJM's signal handlers are initialized. When ctrl+c is pressed, your handler routine is called, then LJM's signal handler is called, which exits the program.

A minimal Python script to set up a handler routine by importing `Kernel32.dll` is as follows:

```
import ctypes
import sys

from labjack import ljm
ljm.closeAll() # Initializes LJM's signal handlers

if not sys.platform.startswith("win32"):
    raise ValueError("Unsupported platform: " + sys.platform)
kernel32 = ctypes.CDLL("Kernel32.dll")

def ctrlc_handler(dwCtrlType):
    print("Python ctrlc_handler called with dwCtrlType " + str(dwCtrlType))
    # This is where you would close a file, etc.

ctrlc_callback_type = ctypes.WINFUNCTYPE(None, ctypes.c_int)
cc_h = ctrlc_callback_type(ctrlc_handler)
success = kernel32.SetConsoleCtrlHandler(cc_h, True)
if not success:
    ValueError("SetConsoleCtrlHandler failed")

print "please press ctrl+c..."
while True:
```

5.2 - Streaming: LJM_eStreamRead gives error 1301 (LJME_LJM_BUFFER_FULL) or many -9999 values in aData. What can I try?

Streaming: LJM_eStreamRead gives error 1301 (LJME_LJM_BUFFER_FULL) or many -9999 values in aData. What can I try?

When the program running LJM code is unable to keep up with transferring stream data from the device to the computer, `LJM_eStreamRead` will return error 1301 (LJME_LJM_BUFFER_FULL) and/or stream data will contain many -9999 (LJM_DUMMY_VALUE) values. When device sample collection greatly outpaces sample transfer from device, `STREAM_AUTO_RECOVER_END_OVERFLOW` can occur as well. Here's how to address these issues.

Briefly:

- Use a ScansPerRead that is relatively large (one half of the ScanRate is a good place to start)
- Try an Ethernet connection if possible
- Call `LJM_eStreamRead` as quickly as possible
- Use an additional thread or process
- Increase the buffer sizes
- Remove slow USB hubs
- Remove / disable unused connection types
- Free up processing power by ending unneeded processes
- Use plain C/C++
- Increase your program's priority level
- Disable the LJM debug log

In more detail:

ScansPerRead: The larger the number of ScansPerRead you set up in `LJM_eStreamStart`, the less overhead there will be. Increasing ScansPerRead will see larger improvements up until the maximum samples per packet is reached, and smaller improvements after that. For a T-series USB connection, the maximum samples per packet is 24. Over Ethernet, the maximum samples per packet is 512. For example, if you have 2 channels, a ScansPerRead of 12 will utilize the maximum samples per packet over USB. (24 samples per packet ÷ 2 samples per scan = 12 scans per packet)

Ethernet: Ethernet streaming usually ends up being more efficient for this reason, so if you are having problems with USB streaming, try switching to Ethernet streaming if you can. Please note that WiFi streaming is not as fast as Ethernet streaming.

Call LJM_eStreamRead quickly: Sometimes there can be unexpectedly long delays between `LJM_eStreamRead` calls, such as during intense calculations. Make sure your program is calling `LJM_eStreamRead` as quickly as possible. It may be worth putting some timing code in your program to calculate whether `LJM_eStreamRead` is called at the right frequency. For example, if your scan rate is 100,000 Hz and your ScansPerRead is 50,000, you'll want `LJM_eStreamRead` to be called at 2 Hz. You could also try removing all parts of the stream read loop except for the call to `LJM_eStreamRead`, then work your way back towards full functionality.

Use an additional thread or process: Related to the need to the previous point to call `LJM_eStreamRead` as quickly possible, lengthy operations should be performed in a separate thread or process from the thread that calls `LJM_eStreamRead`. If work being done in the loop that calls `LJM_eStreamRead` is slow, such as in the case of displaying graphics, then `LJM_eStreamRead` will fall behind reading from stream. This may cause `LJMScanBacklog` to increase and eventually cause a `LJME_LJM_BUFFER_FULL` error. To prevent this, the data from `aData` may be passed to an alternate thread or process which will perform the lengthy operations.

Increase the buffer sizes: Both LJM and the streaming device have buffers that can run out of space.

- To increase the device buffer: write a larger value to the device's `STREAM_BUFFER_SIZE_BYTES` register. See the [T-series stream documentation](#) for details about `STREAM_BUFFER_SIZE_BYTES`.
- To increase LJM's stream buffer: write a larger value to "`LJM_STREAM_BUFFER_MAX_NUM_SECONDS`". LJM uses this config to calculate the buffer size according to the sample rate. As of this writing, the default is 20 seconds and the maximum is determined by the amount of memory your system has available.

Remove slow USB hubs: Some USB hubs can reduce the maximum throughput of stream:

- Try bypassing any USB hubs and plugging the LabJack's connection directly into the host computer's USB highest-speed port
- Try a high-speed USB hub

Remove / disable unused connection types: Communication can incur non-trivial overhead in microprocessor-based devices. For example, when streaming at maximum rates via USB on the T7, plugging in an Ethernet cable can suddenly cause skipped scans to occur.

- If you're streaming via USB, unplug the LabJack's Ethernet cable, then power-cycle the device before re-starting stream.

Plain C/C++: Using C or C++ is the fastest way to use LJM. You may want to try running the [stream_basic C example](#) to see how it performs and adjusting stream parameters as needed.

Priority level:

- Windows program priority can be set by using the `start command` (/realtime is the highest priority)
- Linux and Mac OS X program priority can be set by using the `nice command` (-20 is the highest priority)

Disable the LJM debug log: Though the LJM debug log is disabled by default, you can ensure it's disabled by setting LJM_DEBUG_LOG_MODE to 1 (LJM_DEBUG_LOG_MODE_NEVER) by using `LJM_WriteLibraryConfigS`—i.e. call `LJM_WriteLibraryConfigS(LJM_DEBUG_LOG_MODE, LJM_DEBUG_LOG_MODE_NEVER)`.

5.3 - What LJM files are installed on my machine?

Description of Files

The following files may be installed, depending on if the installation is a full install or a minimal install. See below for the difference.

- LabJackM.h - a header file that defines LJM functions and contains function usage information
- LabJackM library / dll file(s), which contains the actual LJM code.
- On Windows, the .NET LJM bindings, LabJack.LJM.dll.
- The LabJack LJM folder, which contains:

ljm.log

the default debug log file (Debug logging in LJM is not enabled by default.)

ljm_auto_ips.json

a file to help open connections to LabJack devices during relevant Open calls and ListAll calls.

ljm_constants.json file

a constants file that defines registers and error codes

ljm_deep_search.config

contains IP address ranges for the LJMDeep Search feature

ljm_specific_ips.config

contains IP addresses that will be connected to during every network-based Open or ListAll

ljm_startup_configs.json file

defines the start-up configuration parameters of LJM

readme.md

gives a brief explanation of the files installed to the LabJack LJM folder

Full Install

A full installation of LJM includes library files, device rules files, and graphical applications. It consists of the following files:

Windows 32-bit:

- C:\Program Files\LabJack\Drivers\LabJackM.h
- C:\Program Files\LabJack\Drivers\LabJackM.lib
- C:\Program Files\LabJack\Drivers\LabJack.LJM.dll
- C:\Windows\SysWOW64\LabJackM.dll or C:\Windows\System32\LabJackM.dll
- The LabJack LJM folder, which contains configuration, debug, and information files:
 - Windows Vista and later: C:\ProgramData\LabJack\LJM\
 - Windows XP: C:\Documents and Settings\All Users\Application Data\LabJack\LJM

Windows 64-bit:

- C:\Program Files (x86)\LabJack\Drivers\LabJackM.h
- C:\Program Files (x86)\LabJack\Drivers\64bit\LabJackM.lib
- C:\Program Files (x86)\LabJack\Drivers\LabJack.LJM.dll
- C:\Windows\System32\LabJackM.dll
- The LabJack LJM folder, which contains configuration, debug, and information files:
 - Windows Vista and later: C:\ProgramData\LabJack\LJM\
 - Windows XP: C:\Documents and Settings\All Users\Application Data\LabJack\LJM

Linux:

- /usr/local/include/LabJackM.h
- /usr/local/lib/libLabJackM.so (which is a symlink to the versioned libLabJackM.so.<version> in the same directory)
- /usr/local/share/LabJack/LJM (the LabJack LJM folder), which contains configuration, debug, and information files.
- 90-labjack.rules - usually installed in /lib/udev/rules.d, but can be installed in /etc/udev/rules.d
- /opt/labjack_kipling
- /usr/local/bin/labjack_kipling
- /usr/share/applications/labjack_kipling.desktop
- /usr/share/icons/hicolor/48x48/apps/labjack_kipling.png
- Old versions of LJM: /usr/local/lib/liblabjackusb.so (which is a symlink to the versioned liblabjackusb.so.<version> in the same directory)

macOS:

- /usr/local/include/LabJackM.h
- /usr/local/lib/libLabJackM.dylib (which is a symlink to a versioned libLabJackM-<version>.dylib in the same directory)

- /usr/local/share/LabJack/LJM (the LabJack LJM folder), which contains configuration, debug, and information files.

Minimal Install

A [minimal installation](#) of LJM does not include graphical applications. It consists of the following files.

Windows 32-bit:

- C:\Program Files\LabJack\Drivers\LabJackM.h
- C:\Program Files\LabJack\Drivers\LabJackM.lib
- C:\Program Files\LabJack\Drivers\LabJack.LJM.dll
- C:\Windows\SysWOW64\LabJackM.dll or C:\Windows\System32\LabJackM.dll
- The LabJack LJM folder (see above)

Windows 64-bit:

- C:\Program Files (x86)\LabJack\Drivers\LabJackM.h
- C:\Program Files (x86)\LabJack\Drivers\64bit\LabJackM.lib
- C:\Program Files (x86)\LabJack\Drivers\LabJack.LJM.dll
- C:\Windows\System32\LabJackM.dll
- The LabJack LJM folder (see above)

Linux:

- /usr/local/include/LabJackM.h
- /usr/local/lib/libLabJackM.so (see above)
- /usr/local/share/LabJack/LJM (see above)
- 90-labjack.rules (see above)

macOS:

Please [contact us](#) if you'd like to try a macOS minimal installation.

5.4 - What are the system requirements?

[LJM](#) is supported on Windows, macOS, and Linux.

Windows

- Supported: Windows XP or later
- Supported: 32-bit or 64-bit
- Required: A C or C++ compiler. Microsoft's [Visual Studio](#) works well

Windows UWP

- See: [LJM Windows UWP Support](#)

Linux

- Supported: 32-bit, 64-bit, ARMv6, ARMv7l (armhf), aarch64
- Supported: kernel 2.6.28 or later (older versions may work as well)
- Required: [libusb-1.0](#) library or greater version
- Required: A C or C++ compiler. [gcc](#) works well
- Required: libstdc++ version 4.4.7 or later
- Required: glibc version 2.12 or later
- Most distributions are supported, including:
 - CentOS 6.7 and later
 - Most versions of Ubuntu
 - Raspberry Pi's Raspbian
 - [Let us know](#) if there's a distribution you'd like to use that doesn't work

If you have equal or greater versions of the above libraries, LJM should work.

macOS

- Supported: macOS 10.9 and later
- Supported: 64-bit (x86_64)
 - Legacy 32-bit (i386) and PowerPC (ppc) builds are available
- Required: The [Xcode](#) developer tools (or other C or C++ compiler)

5.5 - What network adapters will LJM use?

If your host machine has multiple network adapters, LJM will try to discover Ethernet/WiFi LabJack connections on all of them.

On Windows, LJM uses `GetAdaptersInfo` to find determine what network adapters are available. On Linux and Mac, LJM uses `sgetifaddrs`.

It is important that each network adapter is configured to have the correct subnet mask, since each broadcast address is determined from a bitwise-OR of the network IP with the inverse of the subnet mask.

5.6 - While writing to internal flash, I'm getting "INTFLASH_KEY_INVALID - Error code: 2354". How do I write to internal flash?

While writing to internal flash, I'm getting "INTFLASH_KEY_INVALID - Error code: 2354". How do I write to internal flash?

When writing to [internal flash](#), the key is only valid for the duration of a single packet, so the operation to write the key and to write or erase the data must be grouped into a single packet. LJM has [multiple value functions](#), such as [LJM_eWriteAddresses](#), which allow you to write the key while writing other values. To ensure packets are within the allowable packet size, you can read the `MaxBytesPerMB` parameter of [LJM_GetHandleInfo](#) to get the maximum packet size for the connection type in use. Then, use the Modbus [protocol description](#) to figure out how many values you can send. LJM uses Feedback (MBFB).

Also, note that having [LJM_ALLOWS_AUTO_CONDENSE_ADDRESSES](#) enabled (as it is by default) will help the efficiency of writing to internal flash.

5.7 - Why won't LJM open devices or find devices via LJM_ListAll through the network?

Why won't LJM open devices or find devices via LJM_ListAll through the network?

In general, you may want to consult the [Basic Network & Troubleshooting](#) guide.

You may want to try [LJM Specific IPs](#) for devices that can be ping'd but for which LJM cannot seem to find.

Linux

Check your iptables configuration. If disabling iptables fixes the issue, you may want to alter your iptables rules. In the general case, running a command such as:

```
sudo iptables -I INPUT 2 -p udp --sport 52362 -j ACCEPT
```

will add an iptables rule in the second spot (INPUT 2) to allow UDP packets on the port 52362, which is the UDP port LJM uses to discover devices.

Don't forget to save your iptables after altering the configurations. For example, on CentOS:

```
sudo /sbin/service iptables save
```

You may want to test your iptables is correct after a reboot.

5.8 - LJME_RECONNECT_FAILED

When LJM detects that a device has not responded to a communication attempt within the expected timeframe, it attempts to reconnect to the device. If the reconnection is successful, the operation is retried. If the reconnection fails, the `LJME_RECONNECT_FAILED` error is returned.

Make sure the LJM timeout is adequate

Reading multiple high resolution AINs can be split up - Analog input readings with high resolution indices result in longer sample times, so if you are using a function like [eReadNames](#) to read multiple AINs high resolution indices, you need to split those reads up or increase LJM's given [SEND_RECEIVE_TIMEOUT_MS](#) config. It is generally safe to read a single AIN at a time, no matter what resolution index is set. The appropriate timeout can be calculated according to the length of time for each AIN sample as given by a Datasheet [appendix](#), for example.

Use the most reliable connection type possible - USB is most reliable, while Ethernet and WiFi connections are less reliable due to inherent network complications. WiFi is less reliable than Ethernet due to possible interference.

Use an appropriate send/receive timeout - LJM's default [SEND_RECEIVE_TIMEOUT_MS](#) configs balance reliability and speed of error detection. For networks with high latency or for failure-intolerant operations, use a longer timeout to increase reliability.

Retry upon failure

Real-world device communication is prone to errors. Though LJM internally retries failed operations once, you may need to retry by calling the same function that returned `LJME_RECONNECT_FAILED` with the same parameters. Doing so is usually appropriate for programs that only loop on reading a given set of registers but is generally not appropriate for writes to or reads from [buffer registers](#).

Determine which device to reconnect to, in case of failure

Reconnection will, by default, reconnect a handle according to the same connection type and the same serial number that it was initially opened with, and always to the same device type. For more details, see the following LJM configurations:

- [LJM_AUTO_RECONNECT_STICKY_CONNECTION](#)

- [LJM_AUTO_RECONNECT_STICKY_SERIAL](#)

Reconnection notification

Use [LJM_RegisterDeviceReconnectCallback](#) to get notified when reconnection occurs.

5.9 - Is LJM thread-safe?

LJM is thread-safe. It synchronizes devices between threads so that multiple threads can use the same [device handle](#).

Applications using the same device handle in multiple threads still must coordinate the closing of device handles—if one application thread call [LJM_Close](#) for a given device handle, that device handle will be closed for all threads.

For applications that need to [stream](#) data, [LJM_SetStreamCallback](#) is an easy way start a worker thread.

Other Remarks

For using LJM and the Python libraries `multiprocessing` and `Tkinter` on macOS: It matters when you import `Tkinter`—you must import it after the `multiprocessing` process(es) have started. For more details, see the forum topic [Multiprocessing with Tkinter and LabJackPython](#)

5.10 - gdb breaks with "signal SIG40, Real-time event 40."

LJM uses gRPC, which internally uses SIG40 to wake up threads. By default, gdb breaks when the process receives a signal. This causes gdb to output something like the following:

```
Thread 7 "eNames" received signal SIG40, Real-time event 40.
[Switching to Thread 0x7ffff7fe700 (LWP 90205)]
0x00007ffff7364627 in __GI_epoll_pwait (epfd=17, events=0x7ffff7fd600, maxevents=100, timeout=10000, set=0x7ffff7fe650)
    at ../sysdeps/unix/sysv/linux/epoll_pwait.c:42
42  ../sysdeps/unix/sysv/linux/epoll_pwait.c: No such file or directory.
(gdb)
```

This looks problematic, but it's actually fine. More info on gdb and signals can be found in section [5.4 Signals](#) of the gdb documentation.

Solution: Suppress gdb's default break behavior for SIG40

To circumvent this, you can use the gdb command:

```
handle SIG40 noprint nostop
```

(Related gRPC issue: [SIG36 while debugging with gdb #7906](#))

Solution: Disable LJM_RPC_ENABLE

To disable LJM's use of gRPC, see [How do I disable gRPC?](#)

5.11 - Can I write an LJM program without a device present?

Overview

LJM 1.1901 and later

If you need to write an LJM program but don't have a device available, you can use `LJM_DEMO_MODE` to prevent LJM from trying to open a real device. Demo mode is good for when your device is in shipping or being used for another project.

Demo mode allows you to:

- Test your software setup to make sure LJM and language wrappers are working.
- Write and learn the structure of a simple LJM program.

Try it out now by [installing LJM](#).

Usage

To use demo mode, pass `LJM_DEMO_MODE` (or the string "-2") as the [identifier](#) for `LJM_Open` or `LJM_OpenS`. That will give you a handle that you can use for [command-response](#) communication—however, demo mode does not support stream mode.

Example of demo mode using C (including the `ErrorCheck` function include in the [C/C++ examples](#)):

```
err = LJM_Open(LJM_dtANY, LJM_ctANY, LJM_DEMO_MODE, &handle);
ErrorCheck(err, "LJM_Open");

err = LJM_eWriteName(handle, "DAC0", 2.5);
```

```

ErrorCheck(err, "LJM_eWriteName");

double value;
err = LJM_eReadName(handle, "AIN0", &value);
ErrorCheck(err, "LJM_eReadName");
printf("read: %f\n", value);

err = LJM_Close(handle);
ErrorCheck(err, "LJM_Close");

```

This may output something like:

```
read: 0.000000
```

With demo mode, functions like `LJM_eReadNames` return meaningless values. Future versions of LJM might return different values.

Not Supported

Demo mode does not support [stream functions](#).

Demo mode does not support [low-level functions](#).

5.12 - What permissions does LJM need?

By default, LJM tries to access the following:

- Local file system
- USB
- Network
- Named mutexes

Some programming environments constrain permissions by default. If LJM is not working, it's worth checking if permissions are set correctly.

Set LJM Configurations Before Accessing Devices

The LJM configurations mentioned on this page should essentially be set before any other LJM calls, like `LJM_Open`, since they alter the behavior of functions `LJM_Open`.

Use [LJM_WriteLibraryConfigS](#) to set the configurations that take numbers as values.

Use [LJM_WriteLibraryConfigStringS](#) to set the configurations that take strings as values.

Local file system

Necessary:

LJM needs to access a `ljm_constants.json` file so that it can look up register information. By default, LJM tries to read the `ljm_constants.json` file from the location where `ljm_constants.json` is installed. However, the location of `ljm_constants.json` can be specified via the `LJM_CONSTANTS_FILE` configuration. For more information, see [LJM_CONSTANTS_FILE](#).

The following shows an example of error output when the `ljm_constants.json` file could not be accessed:

```
The error constants file '/usr/local/share/LabJack/LJM/ljm_constants.json' could not be opened.
```

Optional:

LJM may access [other files](#) on the file system for the purposes of improving network device detection, configuration, and debug logging.

USB

LJM needs access to USB in order to communicate via USB with LabJack devices. Alternately, you can use network connections [odemo mode](#).

Network

LJM needs access to the network in order to communicate via TCP or UDP with LabJack devices. Alternately, you can use USB connections [odemo mode](#).

For [LJM on Windows UWP](#), TCP communications require the `internetClientServer` capability.

Optional:

Each LJM instance (each process that loads LJM, including [LabJack Kipling](#)) tries to communicate with other LJM instances for the purpose of identifying and sharing USB connections to LabJack devices. Since this functionality is not necessarily important, you can [disable gRPC](#).

The following shows an example of error output that may result from LJM's usage of gRPC failing due to permissions on a Unix-based machine:

```
E0205 16:05:30.305708000 4295751104 server_chnp2.c:53] [{"created":"@1549404330.305647000","description":"No address added out of total 1 resolved","file":"src/core/ext/transport/chnp2/server/chnp2_server.c","file_line":260,"referenced_errors":[{"created":"@1549404330.305643000","description":"Failed to add any wildcard
```

```
listeners", "file": "src/core/lib/iomgr/tcp_server_posix.c", "file_line": 353, "referenced_errors": [{"created": "@1549404330.305592000", "description": "OS Error", "errno": 47, "file": "src/core/lib/iomgr/socket_utils_common_posix.c", "file_line": 271, "os_error": "Address family not supported by protocol family", "syscall": "socket", "target_address": "[::]0"}, {"created": "@1549404330.305643000", "description": "Unable to configure socket", "fd": 4, "file": "src/core/lib/iomgr/tcp_server_utils_posix_common.c", "file_line": 215, "referenced_errors": [{"created": "@1549404330.305631000", "description": "OS Error", "errno": 1, "file": "src/core/lib/iomgr/tcp_server_utils_posix_common.c", "file_line": 188, "os_error": "Operation not permitted", "syscall": "bind"}]}]}
```

Named Mutexes

LJM uses named mutexes to synchronize usage gRPC. To circumvent this, you can [disable gRPC](#) by setting LJM_RPC_ENABLE to 0.

LJM uses named mutexes to synchronize usage of the [LJM Auto IPs](#) file. To circumvent this, you can disable LJM Auto IPs by setting LJM_AUTO_IPS to 0.

5.13 - How do I disable gRPC?

To disable LJM's use of gRPC, set the LJM library configuration [LJM_RPC_ENABLE](#) to 0 (false). Do this before calling any Open or ListAll calls. This can be done in the [startup configs](#) or by writing using WriteLibraryConfigS.

For example, in C:

```
int error = LJM_WriteLibraryConfigS("LJM_RPC_ENABLE", 0);
if (error != LJME_NOERROR) {
    // handle error
    printf("LJM_RPC_ENABLE fail\n");
}
```

or, with [LJM_Uutilities.h](#):

```
SetConfigValue("LJM_RPC_ENABLE", 0);
```

5.14 - Does LJM support Windows UWP?

We've heard from customers of LJM working on Windows UWP, but there's a couple of steps you need to take.

Please see the [permissions page](#) for details, but here's the basic steps:

Move ljm_constants.json to a local directory that your app can access and change LJM_CONSTANTS_FILE to point to that file path.

Disable gRPC by setting LJM_RPC_ENABLE to 0.

For TCP connections:

Disable LJM Auto IPs by setting LJM_AUTO_IPS to 0.

You'll also need to add the internetClientServer capability:

```
<Capabilities>
<Capability Name="internetClientServer" />
</Capabilities>
```

If you have any trouble with LJM working on Windows UWP, please [contact us](#).

5.15 - Sharing a particular device among multiple processes or computers

Sometimes it's beneficial to access a particular device from multiple processes or computers.

The most lightweight way to share a device is using [UDP reads](#). The [leader strategy](#) is the most robust option.

Device considerations

Connection types

USB: T-series devices support a single USB connection.

Ethernet: T-series devices support 2 simultaneous Ethernet TCP connections. They also support [essentially unlimited Ethernet UDP connections](#).

WiFi: T7-Pros support a single WiFi TCP connection. They also support WiFi UDP connections, but Ethernet UDP should be preferred.

All together: T-series device connections are not mutually exclusive. For example, a T7-Pro could simultaneously handle 1 USB connection, 2 Ethernet TCP connections, and 1 WiFi TCP connection, plus Ethernet UDP connections.

Modbus

The Modbus module in firmware is single-threaded and synchronous. This means that only one Modbus operation may be serviced at a time. To estimate

whether a single device will be able to be shared according to your requirements, see [T-series data rates](#) for information about how much communication overhead and sampling time is required. For example, reading the AIN of a T7 using the highest resolution index takes 159 milliseconds. No other processes are able to perform Modbus operations for that duration.

Lua

The [on-board Lua scripting](#) engine interfaces with the Modbus module, similar to how external connections do. While this means that Lua scripting has full functionality, it also means Lua scripts can be another source of contention.

Lua can also be used to gather data and [put results into user RAM](#). This is useful for long-running operations, such as AIN reads with high resolution.

Strategy: UDP - For network communication _

Using UDP to communicate with the device is simple to implement. It's generally the recommended way to share a device among multiple processes.

UDP is technically connectionless, which means that sharing a device on the network using UDP doesn't consume any of the T-series devices' connections. The result is that your loop doesn't have to worry about opening or closing. With UDP, you can [open](#) a device handle using UDP as the ConnectionType, then your loop only needs to read. Collisions are less likely because opening incurs an overhead.

Strategy: Open-read-close - For USB or network communication _

The open-read-close strategy is when each process that needs to access the device opens a connection, reads from the device (or writes), then immediately closes the device connection. This strategy is:

- Simple to implement
- A good alternative to UDP when using USB
- Sufficient for slower sampling frequencies, depending on how many processes access the device
- Susceptible to sampling jitter

Pseudocode for doing open-read-close might look loosely like the following:

```
while (reading) {
    device = OpenDevice(openParameters)
    result = ReadFromDevice(device, readParameters)
    CloseDevice(device)
    ProcessResult(result)
    Sleep()
}
```

One downside of open-read-close compared to UDP reads is that opening incurs overhead in two forms: the TCP SYN/FIN commands incur very slight overhead (a couple of packets) and LJM reads from the device every time it opens a device (less than 10 round-trip packets).

One downside of open-read-close compared to the leader strategy is that clock drift between different computers can increase the likeliness that multiple computers will try to open the device at the same time.

With LJM, you may want to experiment with the [open timeouts](#) and [send timeouts](#) to give each process a better chance at re-trying to opens and Modbus operations. For example, you might want to set the open timeout (for whatever connection type you're using) to 200 ms, so that there are 5 attempts to read from the device within 1 second.

Strategy: Use a Leader Process - For USB or network communication _

The leader strategy is when one process acts as the leader. The leader is the only process that reads from the device. Other processes then read from the leader. This works for USB or network connections. This strategy:

- Requires the most custom work
- Is the most robust strategy
- Can have the least amount of sampling jitter
- Is compatible with [stream mode](#)
- Scales to allow the most processes reading from the device

For network connections, the leader process may be on a separate computer from the others and it exposes network connections to the other 3 hosts that let them get readings.

Regardless of whether the connection is via USB or the network, the leader responsibilities include:

- Leader must know what the other processes need to sample and at what interval.
- Leader must expose an interface for the other processes to read from.
 - Files, named pipes, shared memory, remote procedure calls, and REST endpoints are a couple of ways for the leader to expose such an interface.
 - Recommended proof-of-concept for USB: The leader can write a timestamp and the current value to a file for each non-leader process. For example, if the leader knows that Process B needs to read AIN2, it could write the current [Epoch time](#) comma-separated with the current value of AIN2 to a file—the file could be named `proc_b_ain2.csv` and the leader could write `1556211610, 3.2` to indicate that AIN2's current value at that time is 3.2. Process B could then read that file. Every time the timestamp changes, Process B knows that a new value has been written.

This is the approach that is likely have the least amount of sampling jitter because the leader process can regularly schedule readings (the [LJM timing functions](#) are helpful for this). Clock drift between multiple computers does not affect the sampling time because only one computer's clock is triggering the

sample collections.

For hardware-timed sampling using stream mode, this is the only possible type of device sharing strategy because T-series devices may only perform one stream at a time. Also, stream mode takes over the entire analog input system such that analog inputs are not available to any command-response reads.

Since communication with the device can be a performance bottleneck, the leader strategy is the most scalable because it can incur the least amount of communication overhead with the device.
