19 May 2019

Published on *LabJack* (https://labjack.com)

Direct Modbus TCP

Log in or register to post comments

Introduction

Modbus is a simple and common protocol used in industrial environments. We specifically use Modbus TCP, which sends Modbus packets via TCP/IP.

LabJack devices that speak Modbus are Modbus TCP Servers. Software or devices that can act as a Modbus TCP Client, can talk directly to a Modbus Server over a TCP/IP connection such as Ethernet or WiFi. We do not know of any clients that can speak Modbus TCP over USB.

Modbus TCP/UDP is the only protocol used by the T4 and T7. Even the LJM library is using Modbus packets under-the-hood for all communication with these devices. That said, our high-level LJM library uses a couple features that are not "standard" Modbus. These features are an option for anyone using Modbus in their own application, but for standard COTS Modbus clients these features are likely not an option:

- Spontaneous Stream Data - When our LJM library starts a stream, it launches a background thread that is constantly listening for incoming data from the device such that the device can spontaneously send data packets at any time without being asked for each data packet. The data packets are formed as Modbus TCP packets, but sending them spontaneously is not standard and would not be supported by any COTS Modbus clients. Command-response stream is an option, but still would not work with any COTS Modbus clients.
- Feedback Function (MBFB) - This is a custom function we have defined that is very efficient, and in particular allows writes and reads in the same packet. No COTS Modbus client will support this proprietary function, but all the same things can be done with separate standard write and read functions.

The U3, U6, and UE9 have limited Modbus support, and for full functionality use a proprietary low-level protocol described in Section 5 of each datasheet.

Getting Started

The T7 and T4 have five test registers that should be used before trying to read/write to other registers. This will help in trying to debug any getting-started related issues with addressing or bit-wise (byte-flipping) issues.

Modbus Client Applications

Available Modbus TCP/UDP client applications.

Example Code

Available modbus TCP/UDP library examples are on our examples page.

Modbus Map

The Modbus Map defines the address and name of all registers, along with other information. This section has a dynamic map that allows you to filter and search.

Protocol Details

Lengthy (and confusing) detail about Modbus TCP can be found atmodbus.org, but all the information actually needed is covered in this section.

UD Modbus (Old)

The U3, U6, and UE9 implemented a deprecated Modbus system called UD Modbus.

Getting Started

Log in or register to post comments

Modbus - Getting Started [referenceable]

To get started with direct Modbus TCP, first try writing & reading to/from the test registers.

See the Examples section on theModbus Protocol Details page.

If using COTS Modbus Client software, keep the following in mind:

- The LabJack is a Modbus TCP Server. A Modbus TCP Client can send a command to the LabJack and get back a response. Sometimes a Server is called the Slave and a Client is called the Master.
- We use a single register map with addresses from 0 to 65535. Each address points to a 16-bit value that might be readable, writable, or both.
- The meaning of the registers are defined in our Modbus Map.
- Use function 3, 4, 6, or 16. Choose "Holding" if needed.

If you don't have a better way to see the bytes written and read for each packet, you can useWireshark to get a TCP capture.

Test Registers

Name	Start Address	Type	Access	Default
TEST	55100	UINT32	R	1122867
TEST_UINT16	55110	UINT16	R/W	17
TEST_UINT32	55120	UINT32	R/W	1122867
TEST_INT32	55122	INT32	R/W	-2003195205
TEST_FLOAT32	55124	FLOAT32	R/W	-9999.0

TEST

A read of this test register should always return 0x00112233 or d1122867. If your software has the word swap quirk, you will incorrectly read 0x22330011 or 573767697. If your software has the address-1 quirk, a UINT16 (1-register) read from 55101 will incorrectly return 0x0011 (should read 0x2233).

TEST_UINT16

Write a value and read back to test UINT16 operation. Default is 0x0011 or d17.

TEST_UINT32

Write a value and read back to test UINT32 operation. Default is 0x00112233 or d1122867. If your software has the word swap quirk, the default will incorrectly read 0x22330011 or 573767697.

TEST_INT32

Write a value and read back to test INT32 operation. Default is 0x8899AABB or d-2003195205. If your software has the word swap quirk, the default will incorrectly read 0xAABB8899 or -1430550375.

TEST_FLOAT32

Write a value and read back to test FLOAT32 operation. Default is 0xC61C3C00 or -9999.0. If your software has the word swap quirk, the default will incorrectly read 0x3C00C61C or 0.00786.

Modbus Client Applications

Log in or register to post comments

Modbus Client Applications

LabJacks with Modbus support follow the standard, so any Modbus TCP Client should be able to talk to our devices. See theProtocol Details page for everything you need to know about Modbus TCP on a LabJack, but here are a few highlights:

- The LabJack is a Modbus TCP Server. A Modbus TCP Client can send a command to the LabJack and get back a response. Sometimes a Server is called the Slave and a Client is called the Master.
- We use a single register map with addresses from 0 to 65535. Each address points to a 16-bit value that might be readable, writable, or both.
- The meaning of the registers are defined in our Modbus Map.
- Use function 3, 4, 6, or 16. Choose "Holding" if needed.

Common quirks to watch for with Modbus Clients:

1. The client subtracts 1 from all addresses. You tell the client you want to read address 2000, but the client puts 1999 in the actual packet. That means if you want to read address 2000, you have to tell the client 2001. This seems to be an attempt to give the user addresses that go from 1-65536 rather than 0-65535. We use 0-65535 addressing. If you want to read address 2000, then 2000 should be in the packet.
2. The software flips the order of the words within a 32-bit value. For example, a read of TEST should return 0x00112233, but the client returns 0x22330011.
3. The software says it is adding 40000 to the addresses, but if you look at the actual packet it is not. For example, if you ask the software to read from 2000, it will say it is reading from 42000, but in the actual Modbus packet it specifies address 2000. Does not seem to cause problems, but can be confusing.

General Modbus Information

More getting started information and general Modbus resources can be found on theModbus API Documentation page.

Open PLC (Free, Open Source)

Open PLC is an open source PLC software suite that has been built to function on an industrial and home automation level. It has been put together by a PhD student Thiago Rodrigues Alves and according to his website consists of three parts: Runtime, Editor, and HMI builder. The HMI builder (ScadaBR) portion of the project can be used to directly interface with LabJack T-Series devices. The other applications can also be used to utilize LabJack devices.

pvbrowser (Free, Open Source)

pvbrowser is an open source HMI/SCADA/DCS project whose code is available on GitHub. The project started in 2000 and is a fairly mature project that enables users to create client-server architectured open source SCADA systems and HMIs. pvbrowser at its core is a client-server uses QT at its core to enable users to create customized SCADA system servers (install the PV development package) that can connect to multiple Modbus TCP slave devices, collect data, and then report data using their pvbrowser (client) application. Both the client and server portions can be run on multiple operating systems: Windows, Linux, macOS, and others (downloads page). There is a great introductory example on their examples page. Essentially, go to their downloads page, download the appropriate client application, and instruct it to connect to their demo server "pv://pvbrowser.org:5050.

ScadaBR (Free, Open Source)

ScadaBR is a project that started in 2006 and has been slowly working on an open source free software platform that communicates with Modbus TCP devices and allows users to create customizable HMI screens tor Automation, Data Acquisition, and Supervisory Control applications. This application is featured in the OpenPLC project, and has a v2.0 or an -LTS version in the works that looks quite promising.

Node-Red (Free, Open Source)

Node-RED is a powerful high-level GUI based programming language (much like LabVIEW) that can be run on a Raspberry Pi. Here is a good Raspberry Pi + Node-RED + Modbus TCP + MQTT tutorial video on YouTube.

SCADA-LTS (Free and Paid, Open Source)

SCADA-LTS new and potentially up-and-coming SCADA application that is developed in Java, is open source, and features a GUI builder that may be worth checking out.

PyScada (Free, Open Source)

PyScada is an open source application built using Python and features a modern HTML5 based HMI. It has very low hardware requirements for the server and is completely open source. They have great documentation and their code is available on GitHub PyScada.

Infinite Automation (Paid, Contracted SCADA System Development)

Infinite Automation produces a Building Automation, Data Acquisition, IIoT, or SCADA application called Mango Automation that has a free version as well as multiple paid option and advertises many great features from protocol choices to HMI pages and data logging. They also have an open source Modbus library written in Java that can be downloaded free of charge from their website called Modbus4J.

mySCADA (paid, contracting available)

mySCADA produces a device agnostic SCADA system with a lot of options for logging, controlling, and HMI development. If you are looking for a company to help you make a customized solution to automate your lab, factory, etc, take a look at this company and pair their application with powerful T-Series DAQ devices.

IGSS (Free to 50 channels)

IGSS is an application produced by schneider electric that has a free version that supports up to 50 objects/tags and allows you to use a generic Modbus TCP driver (ID:64) to collect data from generic PLC equipment like LabJack T-Series devices (T4,T7).

Advanced HMI (Custom HMI development, paid)

Advanced HMI is a company that produces industrial panel PCs that run windows and produces a free HMI software package that has support for

Modbus TCP devices. There is a simple video tutorial posted on their SourceForge page.

OpenSCADA

OpenSCADA Is project that has been around since 2006 and is maintained by some folks in Ukraine. It looks like they have had some active development in 2018 and their SCADA software supports a large number of Linux distributions and single board computer hardware platforms.

OpenAPC

OpenAPC (Open Advanced Process Control) Is project that has been around for quite a while and offers a platform independent HMI/PLC/SCADA application that runs on macOS, Windows, and Linux.

Simply Modbus TCP (Free and Paid Options, old school look)

Simply Modbus is a very simple Modbus TCP test and log program that has existed for many years. They have a webpage that describes Modbus TCP (the protocol) and they have a free demo Modbus TCP Client application available and offer paid license options for going further.

Modbus Tools (Free and Paid)

Modbus Tools is another simple Modbus TCP test and log program/API suite that has existed for many years. They have a free demo available and offer paid license options for going further. They offer a 3rd party option to our LJM library for a programmatic interface to Modbus TCP devices and they also offer an Active X plugin for Excel.

TCP Modbus Examiner tool (Free, source code available)

The TCP Modbus Examiner tool is a simple tool that communicates with Modbus TCP devices developed usingWPF that illustrates the simplicity and usefulness of using the Modbus TCP protocol for data acquisition applications.

AzeoTech QuickMod (free download)

We got this to work easily. Do the following to start reading FIO0-FIO3:

Start => Programs => DAQFactory => Samples => QuickMod Modbus Scanner:

- Select Modbus TCP over Ethernet.
- Click Configure, set IP Address to the correct value, and set Port = 502.
- Change Tag # = 2000.

Chipkin CAS Modbus Scanner (free download)

We got this to work easily, but noticed all 3 quirks above.

Steps to read FIO0-FIO3:

- Add a TCP connection with proper IP and port 502.
- Add a device with ID 1.
- Add a request with Function #3, Offset=2001, and Length=4.
- Go back to the main screen, select that task/request, and click "Poll".

Rockwell / Allen-Bradley PLC

A customer reported success with an EN2T module specifically, and others have reported success with Allen-Bradley PLCs in general. Reference technote 470365 (from Rockwell Automation or Allen-Bradley).

Maple Systems 5070TH, Weintek MT8070iH, EasyBuilder Pro

A customer told us that the 5070TH and MT8070iH are the same HMI. EasyBuilder Pro software is a free download, and has a simulator that can be used for testing. While helping the customer we did get EasyBuilder Pro to work ourselves, and made the following obscure notes for someone who knows nothing about the software:

- To change settings for the LabJack, go to Edit => System Parameters.
- To change settings for the Numeric Display Object, click on it, then right-click and go to Attributes.
- To make it go, click on On-Line Simulation.

We noticed that this client had quirks #1 and #2 from the list at the top of this page. To handle #2, EasyBuilder provides a function called SWAPW that the customer used successfully.

Wonderware

A customer reported success with Wonderware. They reported that it had all 3 quirks from the list at the top of this page, and in fact they had to request address 40001 to get it to read address 0.

More SCADA software can be found through google searches and these links:

- Source Forge (SCADA)
- A post on 14core.com's website about SCADA programs for your SBC
- A post by Chipkin Automation containing a list of useful Modbus TCP tools.

Modbus Client Applications Table

Modbus Client Applications (T-Series)

T7, T4

Clients...	Windows	Mac	Linux
AzeoTech QuickMod	✔		
Chipkin CAS Modbus Scanner	✔		
IntegraXor	✔		
Maple Systems Web Studio SCADA	✔	✔	✔

More information about Modbus as well as some getting started information can be found on the Modbus API Documentation page.

Log in or register to post comments

Direct Modbus TCP Examples

Log in or register to post comments

Direct Modbus TCP

Direct Modbus TCP Overview

Because the T7 and T4 transmit all of their information using Modbus TCP, they are compatable with dozens of 3rd party libraries. If we don't support a particular language required for an application, try searching for a Modbus TCP library that is compatible with your language of choice. The following is a list of libraries that we have successfully used to communicate with our devices using Modbus TCP.

Getting Started

The T7 and T4 have five test registers that should be used before trying to read/write to other registers. This will help in trying to debug any getting-started related issues with addressing or bit-wise (byte-flipping) issues.

Modbus Map

The Modbus Map defines the address and name of all registers, along with other information. This section has a dynamic map that allows you to filter and search.

Direct Modbus API Documentation

For more information about LabJack's Direct Modbus implementation check out the API documentation section.

Modbus TCP Example Code Table

Modbus TCP Example Code Table (Referencable)

Direct Modbus TCP Example Code

T7, T4

Libraries...	Windows	Mac	Linux
LabVIEW	✔	✔	✔
Node.js	✔	✔	✔
Python	✔	✔	✔
C/C++ (Streaming)	✔	✔	✔

To use a 3rd party Modbus program, such as a Rockwell/Allen Bradley application, there is no download - simply follow the instructions on the Modbus Client Applications page.

More information about Modbus as well as some getting started information can be found on the Modbus API Documentation page. A list of the available Modbus registers that the T7 supports is available on the Modbus Map page.

Log in or register to post comments

Modbus Map

Log in or register to post comments

3.1 Modbus Map [T-Series Datasheet]

Modbus Map Tool

Device: All Devices

Tags: All Tags
AIN
AIN_EF
ASYNCH
CONFIG
CORE
DAC
DIO
DIO_EF
ETHERNET

Expand addresses: ☐

An error has occurred.

The filter and search tool above displays information about the Modbus registers of T-series devices.

- Name: The string name that can be used with the LJM library to access each register.
- Address: The starting address of each register, which can be used through LJM or with direct Modbus.
- Details: A quick description of the register.

- Type: Specifies the datatype, which specifies how many registers each value uses.
- Access: Each register is readable, writable, or both.
- Tags: Used to associate registers with particular functionality. Useful for filtering.

For the U3, U6 and UE9, see the deprecated Modbus system called UD Modbus.

For a printer-friendly version, see the printable Modbus map.

Also On This Page

- 0-Based Addressing
- Single Overlapping Map of Addresses from 0-65535
- Big-Endian
- Data Type Constants
- Sequential Addresses
- ljm_constants.json

Usage

T-series devices are controlled by reading or writing Modbus registers as described on the Communication page.

Protocol

Modbus protocol is described on the Protocol Details page.

0-Based Addressing

The addresses defined in the map above are the same addresses in the actual Modbus packet, and range from 0 to 65535.

Some clients subtract 1 from all addresses. You tell the client you want to read address 2000, but the client puts 1999 in the actual Modbus packet. That means if you want to read Modbus address 2000, you have to tell the client 2001. We use 0-65535 addressing everywhere, so if you want to read an address we document as 2000, then 2000 should be in the Modbus packet.

Single Overlapping Map of Addresses from 0-65535

We have a single map of addresses from 0 to 65535. Any type of register can be located anywhere in that range regardless of data type or whether it is read-only, write-only, or read-write.

Some client software uses addresses written as 4xxxx. In this case, the 4 is a special code that means to use the Modbus read function 0x03 and the xxxx is an address from 0-9999 that might additionally have 1 subtracted before being put in the Modbus packet. The magic number of 40000 (or 40001) has no mention in the Modbus spec that we can find. What this means in terms of how to talk to the LabJack depends on what exactly the client is doing, but if you want to read from an address we have defined as x (0-65535), then x should be the address in the Modbus packet sent out over TCP.

Big-Endian

Modbus is specified as big-endian, which means the most significant value is at the lowest address. With a read of a 16-bit (single register) value, the 1st byte returned is the MSB (most significant byte) and the 2nd byte returned is the LSB (least significant byte). With a read of a 32-bit (2 register) value, the value is returned MSW then LSW, where each word is returned MSB then LSB, so the 4 bytes come in order from most significant to least significant.

We have seen some clients that expect Modbus to be implemented with big-endian bytes but with the least significant word before the most significant word. In other words, the client software flips the order of the words within a 32-bit value. For example, a read of TEST (address 55100) should return 0x00112233, but the client returns 0x22330011.

Data Type Constants

Type	Integer Value
LJM_UINT16	0
LJM_UINT32	1
LJM_INT32	2
LJM_FLOAT32	3
LJM_BYTE	99
LJM_STRING	98

Sequential Addresses

Many registers are sequentially addressed. The Modbus Map gives you the starting address for the first register, and then—depending on whether the data type is 16-bits or 32-bits—you increment the address by 1 or 2 to get the next value:

`Address = StartingAddress + 1*Channel#` (UINT16)

`Address = StartingAddress + 2*Channel#` (UINT32, INT32, FLOAT32)

Note that the term "register" is used 2 different ways throughout documentation:

- A "register" is a location that has a value you might want to read or write (e.g. AIN0 or DAC0).
- The term "Modbus register" generally refers to the Modbus use of the term, which is a 16-bit value pointed to by an address of 0-65535.

Therefore, most "registers" consist of 1 or 2 "Modbus registers".

For example, the first entry in the Modbus Map has the name AIN#(0:254), which is shorthand notation for 255 registers named AIN0, AIN1, AIN2, ..., AIN254. The AIN# data type is FLOAT32, so each value needs 2 Modbus registers, an thus the address for a given analog input is `channel*2`.

ljm_constants.json

LabJack distributes a constants file called ljm_constants.json that defines information about the Modbus register map. The filter and search tool above pulls data from that JSON file.

The ljm_constants GitHub repository contains up-to-date text versions of the Modbus register map:

- **JSON** - ljm_constants.json
- **C/C++ header** - LabJackMModbusMap.h

Log in or register to post comments

Printable Modbus Map

Log in or register to post comments

This page displays information from ljm_constants.json.

Analog Input Registers

Name	Start Address	Type	Access
AIN#(0:13) Returns the voltage of the specified analog input.	0	FLOAT32	R
AIN#(0:13)_RANGE The range/span of each analog input. Write the highest expected input voltage.	40000	FLOAT32	R/W
AIN#(0:13)_NEGATIVE_CH Specifies the negative channel to be used for each positive channel. 199=Default=> Single-Ended.	41000	UINT16	R/W
AIN#(0:13)_RESOLUTION_INDEX The resolution index for command-response and AIN-EF readings. A larger resolution index generally results in lower noise and longer sample times.	41500	UINT16	R/W
AIN#(0:13)_SETTLING_US Settling time for command-response and AIN-EF readings.	42000	FLOAT32	R/W
AIN_ALL_RANGE A write to this global parameter affects all AIN. A read will return the correct setting if all channels are set the same, but otherwise will return -9999.	43900	FLOAT32	R/W
AIN_ALL_NEGATIVE_CH A write to this global parameter affects all AIN. Writing 1 will set all AINs to differential. Writing 199 will set all AINs to single-ended. A read will return 1 if all AINs are set to differential and 199 if all AINs are set to single-ended. If AIN configurations are not consistent 0xFFFF will be returned.	43902	UINT16	R/W
AIN_ALL_RESOLUTION_INDEX The resolution index for command-response and AIN-EF readings. A larger resolution index generally results in lower noise and longer sample times. A write to this global parameter affects all AIN. A read will return the correct setting if all channels are set the same, but otherwise will return 0xFFFF.	43903	UINT16	R/W
AIN_ALL_SETTLING_US Settling time for command-response and AIN-EF readings. A write to this global parameter affects all AIN. A read will return the correct setting if all channels are set the same, but	43904	FLOAT32	R/W

otherwise will return -9999. Max is 50,000 us.

Name	Start Address	Type	Access

&print=true

DAC Registers

Name		Start Address	Type	Access
DAC#(0:1)	Pass a voltage for the specified analog output.	1000	FLOAT32	R/W

&print=true

Digital I/O Registers

Name		Start Address	Type	Access
FIO#(0:7)	Read or set the state of 1 bit of digital I/O. Also configures the direction to input or output. Read 0=Low AND 1=High. Write 0=Low AND 1=High.	2000	UINT16	R/W
EIO#(0:7)	Read or set the state of 1 bit of digital I/O. Also configures the direction to input or output. Read 0=Low AND 1=High. Write 0=Low AND 1=High.	2008	UINT16	R/W
CIO#(0:3)	Read or set the state of 1 bit of digital I/O. Also configures the direction to input or output. Read 0=Low AND 1=High. Write 0=Low AND 1=High.	2016	UINT16	R/W
MIO#(0:2)	Read or set the state of 1 bit of digital I/O. Also configures the direction to input or output. Read 0=Low AND 1=High. Write 0=Low AND 1=High.	2020	UINT16	R/W
FIO_STATE	Read or write the state of the 8 bits of FIO in a single binary-encoded value. 0=Low AND 1=High. Does not configure direction. Reading lines set to output returns the current logic levels on the terminals, not necessarily the output states written. The upper 8-bits of this value are inhibits.	2500	UINT16	R/W
EIO_STATE	Read or write the state of the 8 bits of EIO in a single binary-encoded value. 0=Low AND 1=High. Does not configure direction. Reading lines set to output returns the current logic levels on the terminals, not necessarily the output states written. The upper 8-bits of this value are inhibits.	2501	UINT16	R/W
CIO_STATE	Read or write the state of the 4 bits of CIO in a single binary-encoded value. 0=Low AND 1=High. Does not configure direction. Reading lines set to output returns the current logic levels on the terminals, not necessarily the output states written. The upper 8-bits of this value are inhibits.	2502	UINT16	R/W
MIO_STATE	Read or write the state of the 3 bits of MIO in a single binary-encoded value. 0=Low AND 1=High. Does not configure direction. Reading lines set to output returns the current logic levels on the terminals, not necessarily the output states written. The upper 8-bits of this value are inhibits.	2503	UINT16	R/W
FIO_DIRECTION	Read or write the direction of the 8 bits of FIO in a single binary-encoded value. 0=Input and 1=Output. The upper 8-bits of this value are inhibits.	2600	UINT16	R/W
EIO_DIRECTION	Read or write the direction of the 8 bits of EIO in a single binary-encoded value. 0=Input and 1=Output. The upper 8-bits of this value are inhibits.	2601	UINT16	R/W
CIO_DIRECTION	Read or write the direction of the 4 bits of CIO in a single binary-encoded value. 0=Input and 1=Output. The upper 8-bits of this value are inhibits.	2602	UINT16	R/W
MIO_DIRECTION	Read or write the direction of the 3 bits of MIO in a single binary-encoded value. 0=Input and 1=Output. The upper 8-bits of this value are inhibits.	2603	UINT16	R/W
DIO#(0:22)	Read or write the state of all digital I/O in a single binary-encoded value. 0=Low AND 1=High. Does not configure direction. A read of an output returns the current logic level on the terminal, not necessarily the output state written. Writes are filtered by the value in DIO_INHIBIT.	2800	UINT32	R/W
DIO_STATE	Read or write the direction of all digital I/O in a single binary-encoded value. 0=Input and 1=Output. Writes are filtered by the value in DIO_INHIBIT.	2850	UINT32	R/W

Name	Start Address	Type	Access
DIO_DIRECTION A single binary-encoded value where each bit determines whether _STATE, _DIRECTION or _ANALOG_ENABLE writes affect that bit of digital I/O. 0=Default=Affected, 1=Ignored.	2900	UINT32	R/W

&print=true

Digital Extended Features

Name	Start Address	Type	Access
DIO#(0:22)_EF_ENABLE 1 = enabled. 0 = disabled. Must be disabled during configuration. Note that DIO-EF reads work when disabled and do not return an error.	44000	UINT32	R/W
DIO#(0:22)_EF_INDEX An index to specify the feature you want.	44100	UINT32	R/W
DIO#(0:22)_EF_OPTIONS Function dependent on selected feature index.	44200	UINT32	R/W
DIO#(0:22)_EF_VALUE_A Function dependent on selected feature index.	44300	UINT32	R/W
DIO#(0:22)_EF_VALUE_B Function dependent on selected feature index.	44400	UINT32	R/W
DIO#(0:22)_EF_VALUE_C Function dependent on selected feature index.	44500	UINT32	R/W
DIO#(0:22)_EF_VALUE_D Function dependent on selected feature index.	44600	UINT32	R/W
DIO#(0:22)_EF_READ_A Reads an unsigned integer value. The meaning of the integer is dependent on selected feature index.	3000	UINT32	R
DIO#(0:22)_EF_READ_A_AND_RESET Reads the same value as DIO#(0:22)_EF_READ_A and forces a reset.	3100	UINT32	R
DIO#(0:22)_EF_READ_B Reads an unsigned integer value. The meaning of the integer is dependent on selected feature index.	3200	UINT32	R

&print=true

Digital EF Clock Source

Name	Start Address	Type	Access
DIO_EF_CLOCK0_ENABLE 1 = enabled. 0 = disabled. Must be disabled during configuration.	44900	UINT16	R/W
DIO_EF_CLOCK0_DIVISOR Divides the core clock. Valid options: 1,2,4,8,16,32,64,256.	44901	UINT16	R/W
DIO_EF_CLOCK0_OPTIONS Bitmask: bit0: 1 = use external clock. All other bits reserved.	44902	UINT32	R/W
DIO_EF_CLOCK0_ROLL_VALUE The clock will count to this value and then start over at zero. The clock pulses counted are those after the divisor. 0 results in the max roll value possible. This is a 32-bit value (0-4294967295) if using a 32-bit clock, and a 16-bit value (0-65535) if using a 16-bit clock.	44904	UINT32	R/W
DIO_EF_CLOCK0_COUNT Current tick count of this clock. Will read between 0 and ROLL_VALUE-1.	44908	UINT32	R

&print=true

SPI Registers

Name	Start Address	Type	Access
SPI_CS_DIONUM The DIO line for Chip-Select.	5000	UINT16	R/W

Name	Start Address	Type	Access
SPI_CLK_DIONUM The DIO line for Clock.	5001	UINT16	R/W
SPI_MISO_DIONUM The DIO line for Master-In-Slave-Out.	5002	UINT16	R/W
SPI_MOSI_DIONUM The DIO line for Master-Out-Slave-In.	5003	UINT16	R/W
SPI_MODE The SPI mode controls the clock idle state and which edge clocks the data. Bit 1 is CPOL and Bit 0 is CPHA, so CPOL/CPHA for different decimal values: 0 = 0/0 = b00, 1 = 0/1 = b01, 2 = 1/0 = b10, 3 = 1/1 = b11. For CPOL and CPHA explanations, see Wikipedia article: <a target='_blank' href='https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus'>https://e... style='margin-right: -1;' src='https://ljsimpleregisterlookup.herokuapp.com/static/images/ui-icons-extl... />.	5004	UINT16	R/W
SPI_SPEED_THROTTLE This value controls the SPI clock frequency. Pass 0-65535. Default=0 corresponds to 65536 internally which results in ~800 kHz. 65500 = ~100 kHz, 65100 = ~10 kHz, 61100 = ~1 kHz, 21000 = ~100 Hz, and 1 = ~67 Hz. Avoid setting too low such that the entire transaction lasts longer than the 250 millisecond timeout of the internal watchdog timer.	5005	UINT16	R/W
SPI_OPTIONS Bit 0 is Auto-CS-Disable. When bit 0 is 0, CS is enabled. When bit 0 is 1, CS is disabled. Bit 1: 0 = Set DIO directions before starting the SPI operations, 1 = Do not set DIO directions. Bit 2: 0 = Transmit data MSB first, 1 = LSB first. Bits 4-7: This value sets the number of bits that will be transmitted during the last byte of the SPI operation. Default is 8, valid options are 1-8.	5006	UINT16	R/W
SPI_GO Write 1 to begin the configured SPI transaction.	5007	UINT16	W
SPI_NUM_BYTES The number of bytes to transfer.	5009	UINT16	R/W
SPI_DATA_TX Write data here. This register is a buffer.	5010	BYTE	W
SPI_DATA_RX Read data here. This register is a buffer. Underrun behavior - fill with zeros.	5050	BYTE	R

&print=true

I2C Registers

Name	Start Address	Type	Access
I2C_SDA_DIONUM The number of the DIO line to be used as the I2C data line. Ex: Writing 0 will force FIO0 to become the I2C-SDA line.	5100	UINT16	R/W
I2C_SCL_DIONUM The number of the DIO line to be used as the I2C clock line. Ex: Writing 1 will force FIO1 to become the I2C-SCL line.	5101	UINT16	R/W
I2C_SPEED_THROTTLE This value controls the I2C clock frequency. Pass 0-65535. Default=0 corresponds to 65536 internally which results in ~450 kHz. 1 results in ~40 Hz, 65516 is ~100 kHz.	5102	UINT16	R/W
I2C_SLAVE_ADDRESS The 7-bit address of the slave device. Value is shifted left by firmware to allow room for the I2C R/W bit.	5104	UINT16	R/W
I2C_NUM_BYTES_TX The number of data bytes to transmit. Zero is valid and will result in a read-only I2C operation.	5108	UINT16	R/W
I2C_NUM_BYTES_RX The number of data bytes to read. Zero is valid and will result in a write-only I2C operation.	5109	UINT16	R/W
I2C_OPTIONS Advanced. Controls details of the I2C protocol to improve device compatibility. bit 0: 1 = Reset the I2C bus before attempting communication. bit 1: 0 = Restarts will use a stop and a start, 1 = Restarts will not use a stop. bit 2: 1 = disable clock stretching.	5103	UINT16	R/W
I2C_GO Writing to this register will instruct the LabJack to perform an I2C transaction.	5110	UINT16	R/W

Name	Start Address	Type	Access
I2C_ACKS An array of bits used to observe ACKs from the slave device.	5114	UINT32	R/W
I2C_DATA_TX Data that will be written to the I2C bus. This register is a buffer.	5120	BYTE	W
I2C_DATA_RX Data that has been read from the I2C bus. This register is a buffer. Underrun behavior - fill with zeros.	5160	BYTE	R

&print=true

LJTick-DAC Registers

Name	Start Address	Type	Access
TDAC#(0:22) Update a voltage output on a connected LJTick-DAC accessory. Even TDAC# = DACA, Odd TDAC# = DACB. For instance, if LJTick-DAC accessory is connected to FIO2/FIO3 block on main device, TDAC2 corresponds with DACA, and TDAC3 corresponds with DACB.	30000	FLOAT32	W
TDAC_SERIAL_NUMBER Returns the serial number of an LJTick-DAC, and forces a re-read of the calibration constants. Which LJTDAC is determined by the last write to TDAC# ... whether it was successful or not.	55200	UINT32	R

&print=true

SBUS Registers

Name	Start Address	Type	Access
SBUS#(0:22)_TEMP Reads temperature in Kelvin from an SBUS sensor (EI-1050/SHT1x/SHT7x). SBUS# is the DIO line for the EI-1050 enable. If SBUS# is the same as the value specified for data or clock line, there will be no control of an enable line.	30100	FLOAT32	R
SBUS#(0:22)_RH Reads humidity in % from an external SBUS sensor (EI-1050/SHT1x/SHT7x). # is the DIO line for the EI-1050 enable. If # is the same as the value specified for data or clock line, there will be no control of an enable line.	30150	FLOAT32	R
SBUS#(0:22)_DATA_DIONUM This is the DIO# that the external sensor's data line is connected to.	30200	UINT16	R/W
SBUS#(0:22)_CLOCK_DIONUM This is the DIO# that the external sensor's clock line is connected to.	30225	UINT16	R/W
SBUS_ALL_DATA_DIONUM A write to this global parameter sets all SBUS data line registers to the same value. A read will return the correct setting if all channels are set the same, but otherwise will return 0xFF.	30275	UINT16	R/W
SBUS_ALL_CLOCK_DIONUM A write to this global parameter sets all SBUS clock line registers to the same value. A read will return the correct setting if all channels are set the same, but otherwise will return 0xFF.	30276	UINT16	R/W
SBUS_ALL_POWER_DIONUM Sets the power line. This DIO is set to output-high upon any read of SBUS#_TEMP or SBUS#_RH. Default is FIO6 for the T4 and FIO2 for the T7. An FIO line can power up t… sensors while an EIO/CIO/MIO line or DAC line can power up to 20 sensors. Set to 9999 to disable. To use multiple power lines, use a DAC line for power, or otherwise control power yourself, set this to 9999 and then control power using writes to normal registers such as FIO5, EIO0, or DAC0.	30277	UINT16	R/W

&print=true

1-Wire Registers

Name	Start Address	Type	Access
ONEWIRE_DQ_DIONUM The data-line DIO number.	5300	UINT16	R/W

Name	Start Address	Type	Access
ONEWIRE_DPU_DIONUM The dynamic pullup control DIO number.	5301	UINT16	R/W
ONEWIRE_OPTIONS Controls advanced features. Value is a bitmask. bit 0: reserved, bit 1: reserved, bit 2: 1=DPU Enabled 0=DPU Disabled, bit 3: DPU Polarity 1=Active state is high, 0=Active state is low (Dynamic Pull-Up)	5302	UINT16	R/W
ONEWIRE_FUNCTION Set the ROM function to use. 0xF0=Search, 0xCC=Skip, 0x55=Match, 0x33=Read.	5307	UINT16	R/W
ONEWIRE_NUM_BYTES_TX Number of data bytes to be sent.	5308	UINT16	R/W
ONEWIRE_NUM_BYTES_RX Number of data bytes to be received.	5309	UINT16	R/W
ONEWIRE_GO Instructs the T7 to perform the configured 1-wire transaction.	5310	UINT16	W
ONEWIRE_ROM_MATCH_H Upper 32-bits of the ROM to match.	5320	UINT32	R/W
ONEWIRE_ROM_MATCH_L Lower 32-bits of the ROM to match.	5322	UINT32	R/W
ONEWIRE_ROM_BRANCHS_FOUND_H Upper 32-bits of the branches detected during a search.	5332	UINT32	R
ONEWIRE_ROM_BRANCHS_FOUND_L Lower 32-bits of the branches detected during a search.	5334	UINT32	R
ONEWIRE_SEARCH_RESULT_H Upper 32-bits of the search result.	5328	UINT32	R
ONEWIRE_SEARCH_RESULT_L Lower 32-bites of the search result.	5330	UINT32	R
ONEWIRE_PATH_H Upper 32-bits of the path to take during a search.	5324	UINT32	R/W
ONEWIRE_PATH_L Lower 32-bits of the path to take during a search.	5326	UINT32	R/W
ONEWIRE_DATA_TX Data to be transmitted over the 1-wire bus. This register is a buffer.	5340	BYTE	W
ONEWIRE_DATA_RX Data received over the 1-wire bus. This register is a buffer. Underrun behavior - buffer is static, old data will fill the extra locations, firmware 1.0225 changes this to read zeros.	5370	BYTE	R

&print=true

Asynchronous Serial

Name	Start Address	Type	Access
ASYNCH_ENABLE 1 = Turn on Asynch. Configures timing hardware, DIO lines and allocates the receiving buffer.	5400	UINT16	R/W
ASYNCH_BAUD The symbol rate that will be used for communication. 9600 is typical. Up to 38400 works, but heavily loads the T7's processor.	5420	UINT32	R/W
ASYNCH_RX_DIONUM The DIO line that will receive data. (RX)	5405	UINT16	R/W
ASYNCH_TX_DIONUM The DIO line that will transmit data. (TX)	5410	UINT16	R/W
ASYNCH_RX_BUFFER_SIZE_BYTES Number of bytes to use for the receiving buffer. Max is 2048. 0 = 200	5430	UINT16	R/W
ASYNCH_NUM_BYTES_RX The number of data bytes that have been received.	5435	UINT16	R

Name	Start Address	Type	Access
ASYNCH_NUM_BYTES_TX The number of bytes to be transmitted after writing to GO. Max is 256.	5440	UINT16	R/W
ASYNCH_TX_GO Write a 1 to this register to initiate a transmission.	5450	UINT16	W
ASYNCH_DATA_TX Write data to be transmitted here. This register is a buffer.	5490	UINT16	W
ASYNCH_DATA_RX Read received data from here. This register is a buffer. Underrun behavior - fill with zeros.	5495	UINT16	R

&print=true Unknown register(s): ASYNCH_NUM_BITS

Stream Configuration

Name	Start Address	Type	Access
STREAM_SCANRATE_HZ Write a value to specify the number of times per second that all channels in the stream scanlist will be read. Max stream speeds are based on Sample Rate which is NumChannels*ScanRate. Has no effect when using and external clock. A read of this register returns the actual scan rate, which can be slightly different due to rounding. For scan rates >152.588, the actual scan interval is multiples of 100 ns. Assuming a core clock of 80 MHz the internal roll value is (80M/(8*DesiredScanRate))-1 and the actual scan rate is then 80M/(8*(RollValue+1). For slower scan rates the scan interval resolution is changed to 1 us, 10 us, 100 us, or 1 ms as needed to achieve the longer intervals.	4002	FLOAT32	R/W
STREAM_NUM_ADDRESSES The number of entries in the scanlist	4004	UINT32	R/W
STREAM_SAMPLES_PER_PACKET Specifies the number of data points to be sent in the data packet. Only applies to spontaneous mode.	4006	UINT32	R/W
STREAM_SETTLING_US Time in microseconds to allow signals to settle after switching the mux. Does not apply to the 1st channel in the scan list, as that settling is controlled by scan rate (the time from the last channel until the start of the next scan). Default=0. When set to less than 1, automatic settling will be used. The automatic settling behavior varies by device.	4008	FLOAT32	R/W
STREAM_RESOLUTION_INDEX The resolution index for stream readings. A larger resolution index generally results in lower noise and longer sample times.	4010	UINT32	R/W
STREAM_BUFFER_SIZE_BYTES Size of the stream data buffer in bytes. A value of 0 equates to the default value. Must be a power of 2. Size in samples is STREAM_BUFFER_SIZE_BYTES/2. Size in scans is (STREAM_BUFFER_SIZE_BYTES/2)/STREAM_NUM_ADDRESSES. Changes while stream is running do not affect the currently running stream.	4012	UINT32	R/W
STREAM_AUTO_TARGET Controls where data will be sent. Value is a bitmask. bit 0: 1 = Send to Ethernet 702 sockets, bit 1: 1 = Send to USB, bit 4: 1 = Command-Response mode. All other bits are reserv…	4016	UINT32	R/W
STREAM_NUM_SCANS The number of scans to run before automatically stopping (stream-burst). 0 = run continuously. Limit for STREAM_NUM_SCANS is 2^32-1, but if the host is not reading data as fast as it is acquired you also need to consider STREAM_BUFFER_SIZE_BYTES.	4020	UINT32	R/W
STREAM_ENABLE Write 1 to start stream. Write 0 to stop stream. Reading this register returns 1 when stream is enabled. When using a triggered stream the stream is considered enabled while waiting for the trigger.	4990	UINT32	R/W
STREAM_SCANLIST_ADDRESS#(0:127) A list of addresses to read each scan. In the case of Stream-Out enabled, the list may also include something to write each scan.	4100	UINT32	R/W

&print=true

Constant Current Sources

Name	Start Address	Type	Access

Name	Start Address	Type	Access
CURRENT_SOURCE_200UA_CAL_VALUE Fixed current source value in Amps for the 200UA terminal. This value is stored during factory calibration, it is not a current reading. Using the equation V=IR, with a known current and voltage, it is possible to calculate resistance of RTDs.	1902	FLOAT32	R
CURRENT_SOURCE_10UA_CAL_VALUE Fixed current source value in Amps for the 10UA terminal. This value is stored during factory calibration, it is not a current reading. Using the equation V=IR, with a known current and voltage, it is possible to calculate resistance of RTDs.	1900	FLOAT32	R

&print=true

Internal Temp Sensor

Name	Start Address	Type	Access
TEMPERATURE_AIR_K Returns the estimated ambient air temperature just outside of the device in its red plastic enclosure. This register is equal to TEMPERATURE_DEVICE_K - 4.3. If Ethernet and/or WiFi is enabled, subtract an extra 0.6 for each.	60050	FLOAT32	R
TEMPERATURE_DEVICE_K Takes a reading from the internal temperature sensor using range=+/-10V and resolution=8, and applies the formula Volts*-92.6+467.6 to return kelvins.	60052	FLOAT32	R

&print=true

Protocol Details

Log in or register to post comments

Modbus - Protocol Details [referenceable]

Lengthy (and confusing) detail about Modbus TCP can be found atmodbus.org, but all the information actually needed is covered in the following.

The LabJack is a Modbus TCP Server. A Modbus TCP Client can send a command to the LabJack and get back a response. Sometimes a Server is called the Slave and a Client is called the Master.

Our Modbus TCP interface is quite simple. It consists of a register map with addresses from 0 to 65535. Each address points to a 16-bit value that might be readable, writable, or both. Any function we support can be used to read or write values from any address. The meaning of the registers are defined in the Modbus Map.

Note that in the Modbus spec, and the function documentation below, a register is specifically a 16-bit value. In our Modbus Map we define some 32-bit values, and these are often referred to as a register, but when looking at the details of Modbus protocol realize that these are actually 2 registers. For example, AIN0 is defined as a 32-bit value that is read starting at address 0. AIN0 is actually stored in 2 registers: the MSW (most significant word) is at address 0 and the LSW (least significant word) is at address 1.

Modbus is big-endian, which means the most significant value is at the lowest address. With a read of a 16-bit (single register) value, the 1st byte returned is the MSB (most significant byte) and the 2nd byte returned is the LSB (least significant byte). With a read of a 32-bit (2 register) value, the value is returned MSW then LSW, where each word is returned MSB then LSB, so the 4 bytes come in order from most significant to least significant.

Packet size limits for the T7 are USB=64, Ethernet=1040, and WiFi=500 bytes. Packet size limits for the T4 are USB=64 and Ethernet=1040 bytes. Modbus packets on the U3 and U6 are limited to 64 bytes, including the 2 zeros appended to the command.

Modbus Functions

We support standard functions 3 (Read Multiple), 4 (Read One), 6 (Write One), and 16 (Write Multiple). We also support a custom function called Modbus Feedback (MBFB) that can handle multiple reads & writes in a single packet.

Some Modbus clients will ask you to specify "Coil", "Holding", "Discrete", or "Input". Choose "Holding", which should tell the client to use function 3, 4, 6, or 16.

Functions 4 and 6 are seldom used, so we will focus our discussion on functions 3, 16, and MBFB:

Command	Read Regs	Write Regs	Feedback (MBFB)
	#3	#16	#76
Byte[0:1]	Trans ID	Trans ID	Trans ID
Byte[2:3]	Protocol ID	Protocol ID	Protocol ID
Byte[4:5]	Length	Length	Length
Byte 6	Unit ID	Unit ID	Unit ID
Byte 7	3	16	76
Byte[8:9]	Address	Address	Frames
Byte[10:11]	Num Regs (n)	Num Regs (n)	
Byte 12		Byte Count (2*n)	
Byte 13...		Data	

Response	Read Regs	Write Regs	Feedback (MBFB)
	#3	#16	#76
Byte[0:1]	Trans ID	Trans ID	Trans ID
Byte[2:3]	Protocol ID	Protocol ID	Protocol ID
Byte[4:5]	Length	Length	Length
Byte 6	Unit ID	Unit ID	Unit ID
Byte 7	3	16	76
Byte 8	Byte Count (2*n)	Address MSB	Data
Byte 9	Data	Address LSB	....
Byte 10	....	Num Regs (n)	
Byte 11			

Read Multiple Registers (Function #3)
Standard Modbus function that reads 1 or more sequential registers from the specified starting address.

Command [# Bytes = 12]
Bytes 0-1: 0-65535 (Transaction ID, echoed by device)
Bytes 2-3: 0 (Protocol ID)
Byte 4: 0 (MSB of length)
Byte 5: 6 (LSB of length)
Byte 6: 1 (Unit ID)
Byte 7: 3 (Function #)
Bytes 8-9: 0-65535 (Starting register address, MSB-LSB)
Bytes 10-11: 1-127 (Number of registers to read, MSB-LSB)

Response [# Bytes = 9 + 2*#Registers, limit depends on device]
Bytes 0-3: Echo of command bytes 0-3 (Transaction ID and Protocol ID)
Bytes 4-5: 3 + 2*#Registers (Length, MSB-LSB)
Byte 6: 1 (Unit ID)
Byte 7: 3 (Function #)
Byte 8: 2*#Registers
Bytes 9+: Data

Write Multiple Registers (function #16)
Standard Modbus function that writes 1 or more sequential registers from the specified starting address.

Command [# Bytes = 13 + 2*#Registers, limit depends on device]
Bytes 0-1: 0-65535 (Transaction ID, echoed by device)
Bytes 2-3: 0 (Protocol ID)
Bytes 4-5: 7 + 2*#Registers (Length, MSB-LSB)
Byte 6: 1 (Unit ID)
Byte 7: 16 (Function #)
Bytes 8-9: 0-65535 (Starting register address, MSB-LSB)
Bytes 10-11: 0-65535 (Number of registers to write, MSB-LSB)
Byte 12: 2*#Registers
Bytes 13+: Data

Response [# Bytes = 12]
Bytes 0-3: Echo of command bytes 0-3 (Transaction ID and Protocol ID)
Byte 4: 0 (MSB of length)
Byte 5: 6 (LSB of length)
Byte 6: 1 (Unit ID)
Byte 7: 16 (Function #)

Bytes 8-9: 0-65535 (Starting register address, MSB-LSB)
Bytes 10-11: 0-65535 (Number of registers to write, MSB-LSB)

Modbus Feedback (MBFB, function #76)
Custom function that supports multiple frames, where each frame reads or writes 1 or more sequential registers. Frames are executed in order.

Command
Bytes 0-1: 0-65535 (Transaction ID, echoed by device)
Bytes 2-3: 0 (Protocol ID)
Bytes 4-5: 0-65535 (Length, MSB-LSB)
Byte 6: 1 (Unit ID)
Byte 7: 76 (Function #)
Bytes 8+: Frames

Read Multiple Frames
Frame Byte 0: 0 (Frame Type)
Frame Byte 1-2: 0-65535 (Starting register address)
Frame Byte 3: 1-255 (Number of registers to read)

Write Multiple Frames
Frame Byte 0: 1 (Frame Type)
Frame Byte 1-2: 0-65535 (Starting register address)
Frame Byte 3: 1-255 (Number of registers to write)
Frame Byte 4+: Data

Response
Bytes 0-3: Echo of command bytes 0-3 (Transaction ID and Protocol ID)
Bytes 4-5: 0-65535 (Length, MSB-LSB)
Byte 6: 1 (Unit ID)
Byte 7: 76 (Function #)
Bytes 8+: Data (Response for read frames)

- Transaction ID: The device echos this value. Use to match responses with commands.
- Protocol ID: Not used. Just pass 0.
- Length: The number of bytes after the "Length" parameter, per the Modbus spec.
- Unit ID: Not used. Pass 1 to fit with convention.
- Function #: 3, 4, 6, or 16 are standard Modbus "holding" functions. 76 is our custom MBFB function.
- Address: A 16-bit address that points to a 16-bit register.
- Register: A 16-bit value pointed to by an address. Registers are defined in ourModbus Map.
- Data: Data to write to registers are data read from registers.

In the event of an error, all functions return the standard Modbus error response, which means you get back the 8-byte header above but bit 7 of byte 7 (function #) will be set, so the value is 131/132/134/144/204 rather than 3/4/6/16/76. You then get a 9th byte, which is an official Modbus spec errorcode:

```
#define ILLEGAL_FUNCTION 0x01
#define ILLEGAL_DATA_ADDRESS 0x02
#define ILLEGAL_DATA_VALUE 0x03
#define SLAVE_DEVICE_FAILURE 0x04
#define ACKNOWLEDGE 0x05
#define SLAVE_DEVICE_BUSY 0x06
#define MEMORY_PARITY_ERROR 0x08
#define GATEWAY_PATH_UNAVAILABLE 0x0A
#define GATEWAY_TARGET_NO_RESPONSE 0x0B
```

Anytime there is an error, further information is available by reading from a group of 4 registers that start at 55000. There are 8 of these groups with starting addresses from 55000 to 55028. For example, the first group is:

55000: LabJack Error Code #0
55001: Error Frame #0
55002: Modbus Error #0
55003: Transaction ID #0

The standard Modbus functions (3/4/6/16) only use the first error information group shown above.

MBFB only uses the other 7 groups with starting addresses from 55004 to 55028. If you get an error response from an MBFB command, look at the upper nibble of the 9th byte to get an offset that tells you which error information group to look at. So with MBFB the starting address for the error information group of 4 registers, is 55000 + (4 * UpperNibble9thByte).

With MBFB, if a frame generates an error, no further frames are executed. Frames before the ErrorFrame are executed (e.g. outputs are set), but no data is returned for those frames (since you just get the standard Modbus error response).

On the U3/U6, when a Modbus command is sent by USB the low-level packet must have 2 zeros appended to the front. This is how the U3/U6 knows that the packet (after the 2 zeros) is Modbus, and not the normal low-level protocol used by the U3/U6. The response does not have anything added and is pure Modbus.

Examples

Read FIO0

FIO0 is a UINT16 (single register) at address 2000. It should read 1 if floating, and 0 if you jumper it to GND. Here is a packet captured with Wireshark:

```
Command:  0x00 0x00 0x00 0x00 0x00 0x06 0x01 0x03 0x07 0xD0 0x00 0x01
Response:  0x00 0x00 0x00 0x00 0x00 0x05 0x01 0x03 0x02 0x00 0x01
```

The first 8 bytes in both packets are Transaction ID (0x0000), ProtocolID (0x0000), Length (0x0006 or 0x0005), UnitID (0x01), and Function# (0x03).

In the command, the last 4 bytes are Address (0x07D0) and #Registers (0x0001). 0x07D0 is decimal 2000.

In the response the last 3 bytes are 2*#Registers (0x02) and Data (0x0001). We get 2*#Registers = 2 as expected, and the data value of our read is 1 meaning that FIO0 is reading high.

Read TEST

TEST starts at address 55100. If you read a UINT32 (2 registers) from here, you should get 0x00112233 (d1122867), or if you read just a UIN16 (1 register) from here you should get 0x0011 (d17).

UINT32 read:

```
Command:  0x00 0x00 0x00 0x00 0x00 0x06 0x01 0x03 0xD7 0x3C 0x00 0x02
Response:  0x00 0x00 0x00 0x00 0x00 0x07 0x01 0x03 0x04 0x00 0x11 0x22 0x33
```

UINT16 read:

```
Command:  0x00 0x00 0x00 0x00 0x00 0x06 0x01 0x03 0xD7 0x3C 0x00 0x01
Response:  0x00 0x00 0x00 0x00 0x00 0x05 0x01 0x03 0x02 0x00 0x11
```

Write TEST_UINT32

Write the value 0xC0BCCCCD to TEST_UINT32 which starts at address 55120.

Write using function d16 (0x10):

```
Command:  0x00 0x00 0x00 0x00 0x00 0x0B 0x01 0x10 0xD7 0x50 0x00 0x02 0x04 0xC0 0xBC 0xCC 0xCD
Response:  0x00 0x00 0x00 0x00 0x00 0x06 0x01 0x10 0xD7 0x50 0x00 0x02
```

Read back using function d3 (0x03):

```
Command:  0x00 0x00 0x00 0x00 0x00 0x06 0x01 0x03 0xD7 0x50 0x00 0x02
Response:  0x00 0x00 0x00 0x00 0x00 0x07 0x01 0x03 0x04 0xC0 0xBC 0xCC 0xCD
```

If your Modbus client supports 32-bit integers and/or floating point values, this is a good register to test that. First write 0xC0BCCCCD as described above.

0xC0BCCCCD is 3233598669 as a decimal 32-bit unsigned integer. 0xC0BCCCCD is -1061368627 as a decimal 32-bit signed integer. Have your client read 55120 as a UINT32 and INT32 and confirm you get those values. If instead you get 3436036284 and -858931012 your client has swapped the word order.

0xC0BCCCCD is equal to -5.90 as a float, so if you read 55120 as a 32-bit float you should get -5.90. If you instead get -107873760.0, your client is swapping the words and interpreting the data as 0xCCCDC0BC.

Write DAC0

DAC0 is a FLOAT32 value starting at address d1000 (0x03E8). We will set DAC0 to 3.3 volts, which is 0x40533333 in hex.

```
Command:  0x00 0x00 0x00 0x00 0x00 0x0B 0x01 0x10 0x03 0xE8 0x00 0x02 0x04 0x40 0x53 0x33 0x33
Response:  0x00 0x00 0x00 0x00 0x00 0x06 0x01 0x10 0x03 0xE8 0x00 0x02
```

Useful FLOAT32 values in hex:
0.0 = 0x00000000
3.3 = 0x40533333
5.0 = 0x40A00000

Read T7 or T4 Product ID (Search network for a device)

Searching for a device is typically handled by the LJM_ListAll() function, but this is how to do it yourself. Broadcast a UDP Modbus feedback packet asking for the product ID (address 60000).

Read product ID:

Command: 0x00 0x00 0x00 0x00 0x00 0x06 0x01 0x4C 0x00 0xEA 0x60 0x02
T7 Response: 0x00 0x00 0x00 0x00 0x00 0x06 0x01 0x4C 0x40 0xE0 0x00 0x00
T4 Response: 0x00 0x00 0x00 0x00 0x00 0x06 0x01 0x4C 0x40 0x80 0x00 0x00

UD Modbus (Old, Deprecated)

Log in or register to post comments

This page is for the deprecated Modbus map called UD Modbus, which applies to:

U3: Hardware 1.30 (U3C), Firmware 1.05+
U6: Firmware 1.00+
UE9: Comm 1.48+, Control 1.97+

The registers listed below are supported to work as specified, but no new development is planned for UD Modbus. The T-series devices use our current Modbus map which is active and fully-supported.

UD Modbus Software

The LJSocket page catalogs our experience connecting to Modbus software. We list of some of the quirks we've seen, e.g., a zero-based offset vs. one-based offset.

Examples of Modbus communication in LabVIEW are attached to the bottom of this page. These work for direct Modbus communication to the UE9, or Modbus communication over USB using LJSocket.

Modbus Support:		
Device	***Hardware Version***	***First Firmware***
U3	1.2	Not Supported
U3	1.21	Not Supported
U3	1.3	Modbus Added in FW 1.05
U6	2	Modbus Added in FW 1.00
UE9	1.1	Modbus Added in Control 1.70, reworked in 1.97. Ethernet support in Comm 1.43 and USB support in Comm 1.48

UD Modbus Protocol

LabJack devices implement the Modbus TCP protocol. For USB devices such as the U3 and U6, use LJSocket to provide a TCP socket interface. Without LJSocket, there is no COTS software we know of that can communicate using Modbus TCP over the LabJack's USB interface.

Only Modbus functions 3 (Read Multiple), 4 (Read One), 6 (Write One), and 16 (Write Multiple) are supported. In COTS software this often means specifying "Holding" registers.

On the U3/U6/UE9, when a Modbus command is sent by USB the low-level packet must have 2 zeros appended to the front. This is how the U3/U6/UE9 knows that the packet (after the 2 zeros) is Modbus, and not the normal low-level protocol. The response does not have anything added and is pure Modbus. This is generally handled automatically at higher levels (e.g. LJSocket adds the zeros). SkyMotes only speak Modbus, so commands should always be pure Modbus with no added zeros.

Ethernet Modbus packets on the UE9 are limited to 256 bytes. USB Modbus packets on the UE9 are limited to 256 bytes, including the 2 zeros appended to the command. USB Modbus packets on the U3 and U6 are limited to 64 bytes, including the 2 zeros appended to the command. SkyMote Modbus packets over USB or Ethernet are limited to 64 bytes.

While the Modbus website goes into more detail, here are the bytes the LabJack devices expect:

Requests					
Function Name	**Read Reg.**	**Read Regs.**	**Write Reg**	**Write Regs.**	
Function Number	**4**	**3**	**6**	**16**	**Description**
Byte[0:1]	Trans ID	Trans ID	Trans ID	Trans ID	Send anything
Byte[2:3]	Protocol ID	Protocol ID	Protocol	Protocol ID	Always pass zero

Byte[4:5]	Length	Length	ID Length	Length	Number of bytes starting at byte 6
Byte 6	Unit ID	Unit ID	Unit ID	Unit ID	For U3, U6, and UE9 pass 0xff
Byte 7	4	3	6	16	Function #
Byte[8:9]	Address	Address	Address	Address	As it appears in the Modbus map
Byte[10:11]	Num Regs (n)	Num Regs (n)	Reg Value	Num Regs (n)	
Byte 12				Byte Count (2*n)	
Byte 13...				Reg Values...	
	Read Reg.	**Read Regs.**	**Write Reg**	**Write Regs.**	
Responses	**4**	**3**	**6**	**16**	
Byte[0:1]	Trans ID	Trans ID	Trans ID	Trans ID	Echo of sent value
Byte[2:3]	Protocol ID	Protocol ID	Protocol ID	Protocol ID	
Byte[4:5]	Length	Length	Length	Length	Number of bytes starting at byte 6
Byte 6	Unit ID	Unit ID	Unit ID	Unit ID	
Byte 7	4	3	6	16	
Byte 8	Byte Count (2*n)	Byte Count (2*n)	Address	Address	
Byte 9	Reg Data	Reg Data			
Byte 10	....	....	Reg Value	Num Regs (n)	
Byte 11					

If developing your own software, functions 3 (Read Multiple Registers) and 16 (Write Multiple Registers) are all that is needed, so here is more detail about those:

Read Multiple Registers (function #3)

Command [# Bytes = 12]
Bytes 0-1: 0-65535 (Transaction ID, echoed by device)
Bytes 2-3: 0 (Protocol ID)
Byte 4: 0 (MSB of Length)
Byte 5: 6 (LSB of Length)
Byte 6: 0-254 (Unit ID, echoed on U3/U6/UE9, 0 for SkyMote bridge, 1-254 for mote)
Byte 7: 3 (function #)
Bytes 8 & 9: 0-65535 (MSB and LSB, respectively, of starting register address)
Byte 10: 0 (MSB of #Registers)
Byte 11: 1-27 or 1-123 (LSB of #Registers, # of registers to read)

Response [# Bytes = 9 + 2*#Registers, max is 64 or 256]
Bytes 0-3: Echo of command bytes 0-3 (Transaction ID and Protocol ID)
Byte 4: 0 (MSB of Length)
Byte 5: 3 + 2*#Registers (LSB of Length)
Byte 6: Unit ID
Byte 7: 3 (function #)
Bytes 8: 2-54 or 2-246 (2*#Registers)
Bytes 9+: Data

Write Multiple Registers (function #16)

Command [# Bytes = 13 + 2*#Registers, max is 62/64 or 254/256]
Bytes 0-1: 0-65535 (Transaction ID, echoed by device)
Bytes 2-3: 0 (Protocol ID)
Byte 4: 0 (MSB of Length)
Byte 5: 7 + 2*#Registers (LSB of Length)
Byte 6: 0-254 (Unit ID, echoed on U3/U6/UE9, 0 for SkyMote bridge, 1-254 for mote)
Byte 7: 16 (function #)
Bytes 8 & 9: 0-65535 (MSB and LSB, respectively, of starting register address)
Byte 10: 0 (MSB of #Registers)
Byte 11: 1-24/25 or 1-120/121 (LSB of #Registers, # of registers to write)

Bytes 12: 2-48/50 or 2-240/242 (2*#Registers)
Bytes 13+: Data

Response [# Bytes = 12]
Bytes 0-3: Echo of command bytes 0-3 (Transaction ID and Protocol ID)
Byte 4: 0 (MSB of Length)
Byte 5: 6 (LSB of Length)
Byte 6: Unit ID
Byte 7: 16 (function #)
Bytes 8 & 9: 0-65535 (MSB and LSB, respectively, of starting register address)
Byte 10: 0 (MSB of #Registers)
Byte 11: 1-24/25 or 1-120/121 (LSB of #Registers, # of registers to write)

Note: "Length" is number of bytes after the "Length" parameter. From the Modbus spec.

Examples

Here is an example LabJackPython session that communicates with a U6 connected over USB. Debugging is turned on to show the bytes sent and received. These are done directly over USB, not through LJSocket, so the outgoing packets have 2 extra zeros in front that tell the USB device this is a Modbus packet.

```
>>> import u6
>>> d = u6.U6()
>>> d.debug = True
>>> d.readRegister(0)
Sent:  [0x0, 0x0, 0xa6, 0x3f, 0x0, 0x0, 0x0, 0x6, 0x0, 0x3, 0x0, 0x0, 0x0, 0x2]
Response:  [0xa6, 0x3f, 0x0, 0x0, 0x0, 0x7, 0x0, 0x3, 0x4, 0xb8, 0xf5, 0x70, 0x0]
-0.00011703372001647949
```

The Python code reads from register 0, which the map below states is AIN0. Here’s how to read from AIN1:

```
>>> d.readRegister(2)
Sent:  [0x0, 0x0, 0xa6, 0x40, 0x0, 0x0, 0x0, 0x6, 0x0, 0x3, 0x0, 0x2, 0x0, 0x2]
Response:  [0xa6, 0x40, 0x0, 0x0, 0x0, 0x7, 0x0, 0x3, 0x4, 0x40, 0x9d, 0x94, 0xfc]
4.9244365692138672
```

AIN1 is at register 2 because each analog input takes 2 registers (32-bits). The Modbus map below lists how many registers each address requires in the “Min Regs” column. Here’s how to read AIN0, AIN1, AIN2, and AIN3 at the same time:

```
>>> d.readRegister(0, numReg = 8)
Sent:  [0x0, 0x0, 0xa6, 0x41, 0x0, 0x0, 0x0, 0x6, 0x0, 0x3, 0x0, 0x0, 0x0, 0x8]
Response:  [0xa6, 0x41, 0x0, 0x0, 0x0, 0x13, 0x0, 0x3, 0x10, 0xb8, 0xee, 0xe0, 0x0, 0x40, 0x9d, 0xa7, 0xbe, 0x3f, 0x3, 0x84, 0x62, 0x3f, 0x16, 0x24, 0xe8]
[-0.00011390447616577148, 4.9267263412475586, 0.51373875141143799, 0.58650064468383789]
```

Because the addresses (0, 2, 6, and 8) are all consecutive, we can request 8 registers starting at address 0. The four floating point values are returned as a sequence of 16 bytes, and LabJackPython (and other Modbus software) knows how to recombine them.

Here’s how to set DAC0 to 3.7 V.

```
>>> d.writeRegister(5000, 3.7)
Sent:  [0x0, 0x0, 0xa6, 0x42, 0x0, 0x0, 0x0, 0xb, 0x0, 0x10, 0x13, 0x88, 0x0, 0x2, 0x4, 0x40, 0x6c, 0xcc, 0xcd]
Response:  [0xa6, 0x42, 0x0, 0x0, 0x0, 0x6, 0x0, 0x10, 0x13, 0x88, 0x0, 0x2]
3.7000000000000002
```

We’ve wired DAC0 to AIN1 so that we can read it back:

```
>>> d.readRegister(2)
Sent:  [0x0, 0x0, 0xa6, 0x43, 0x0, 0x0, 0x0, 0x6, 0x0, 0x3, 0x0, 0x2, 0x0, 0x2]
Response:  [0xa6, 0x43, 0x0, 0x0, 0x0, 0x7, 0x0, 0x3, 0x4, 0x40, 0x6c, 0x5d, 0x37]
3.6931893825531006
```

UD Modbus Map

We've made the UD Modbus map available as a spreadsheet (opens in new window). The spreadsheet may refer to alpha, beta, or unreleased firmware. Contact us for assistance.

In the columns marked “Read” and “Write”, we indicate the progress we’ve made with the following legend:

- **x** means that the register is fully implemented and tested to the satisfaction of our team.
- **n** means that the register is implemented, but not fully tested to the satisfaction of our team.
- **f** means that the register won’t throw an error, but isn’t implemented. This is most commonly used for features that devices don’t offer. For example, the UE9/U6 doesn’t have FIOs that can become Analog, so the FIO Analog registers don’t do anything on the UE9/U6.

- If the space is blank, then it means it isn’t implemented, and will throw a Modbus error if you try to access it.

  The “Min Regs” column indicates if you need to read more than one register. Each register is 16 bits. For example, AIN readings are 32-bit floating point numbers, and so they must be read in increments of 2 16-bit registers.

The online jsBeautifier program can organize your JavaScript code, even if it is unreadable.

					Supported									
					U3 FW 1.28		**UE9 Ctrl 2.04**		**U6 Ctrl 1.06**		**UEW B 0.06**		**SM 0.06**	
Address	**Description**	**Min Regs**	**Data Type**	**More Info**	**Read**	**Write**	**Read**	**Write**	**Read**	**Write**	**Read**	**Write**	**Read**	**Write**
0	Analog Inputs	2	single		x		x		x					
1000	AIN Res	1	u16		x	f	x	x	x	x				
1500	AIN Range	1	u16		x	f	x	x	x	x				
2000	AIN Settling	1	u16		x	f	x	x	x	x				
2500	AIN Options	1	u16		x	f	x		x	x				
3000	AIN Neg Chn	1	u16		x	x	x		x	x				
5000	DAC Values	2	single		x		x	x	x	x				
6000	DIO State	1	u16		x	x	x	x	x	x				
6100	DIO Direction	1	u16		x	n	x	x	x	n				
6200	Power Switch State	1	u16											
6700	FIO States (Upper Byte is Mask)	1	u16	Masks are only	n	n	x	x	x	n				
6701	EIO States (Upper Byte is Mask)	1	u16	meaningful	n	n	x	x	x	n				
6702	CIO States (Upper Byte is Mask)	1	u16	when writing	n	n	x	x	x	n				
6703	MIO States (Upper Byte is Mask)	1	u16											
6750	FIO Directions (Upper Byte is Mask)	1	u16		n	n	x	x	x	n				
6751	EIO Directions (Upper Byte is Mask)	1	u16		n	n	x	x	x	n				
6752	CIO Directions (Upper Byte is Mask)	1	u16		n	n	x	x	x	n				
6753	MIO Directions (Upper Byte is Mask)	1	u16											
6800	Raw Single IO State	1	u16		n	n	x	x	n	n				
6900	Raw Single IO Direction	1	u16		n	n	x	x	n	n				
7000	Timer Clock (Long)	2	u32		x	x	x	x	x	x				
7002	Timer Divisor (Long)	2	u32		x	x	x	x	x	x				
7100	Timer Config	2	u32		x	x	x	x	x	x				
7200	Timers (Read/Reset)	2	u32/i32		n	n	x	x	x	x				
7300	Counters (Read/Reset)	2	u32		n	n	x	x	n	n				
8000	Stream Config													
8200	Stream Start													
8210	Stream Stop													
8500	Stream Data													
9000	SHT													
9100	I2C Options	8		R / W			x	x						
9101	I2C Speed						x	x						
9102	I2C SDA						x	x						
9103	I2C SCL						x	x						
9104	I2C Address						x	x						
9105	I2C Num To Tx						x	x						
9106	I2C Num To Rx						x	x						
9107-9122	I2C Tx Data						x	x						
9200	SPI													
9300	UART													
10000	VBatt												x	
10002	Temperature (C? K? F?)										x		x	
10004	RH%												x	

10006	Light (Lux? Cd?)												x	
10008	Pressure (Unit?)												x	
10800	Motion Detected?												x	
10802	Motion Settings												x	x
10012	Sound (# Addresses?)													
10800	Switch A													
10810	Switch B													
10900	Relay A													
12000	RX LQI		single										x	
12002	TX LQI		single										x	
12004	Get Battery		single										x	
12006	Get Temp		single										x	
12008	Get Light		single										x	
12010	Get Motion		single										x	
12012	Get Sound		single										x	
12014	Get RH%		single										x	
12016	Get Pressure		single											
14000	TLB Check-In Report 1	24												
14000	Status/Reserved	1	u16	bit 0 - Flashing									x	
14001	RX LQI (Byte0), TX LQI (Byte1)	1	2 bytes										x	
14002	Battery Voltage (V)	2	single										x	
14004	Bump Count / Motion	1	u16										x	
14005	# MMA Sample	1	u16										x	
14006	Temperature (C)	2	single										x	
14008	Light Current (Units?)	2	single										x	
14010	Light Max	2	single										x	
14012	Light Min	2	single										x	
14014	Sound Current (Units?)	2	single										x	
14016	Sound Max	2	single										x	
14018	Sound Min	2	single										x	
14020	RH %	2	single										x	
14022	Reserved	2	reserved											
14030	TLB Check-In Report 2	23											x	
14030	Status/Reserved	1	u16										x	
14031	Temperature Max	2	single										x	
14033	Temperature Min	2	single										x	
14035	Temperature Average	2	single										x	
14037	Light Average	2	single										x	
14039	Sound Average	2	single										x	
14041	RH % Max	2	single										x	
14043	RH % Min	2	single										x	
14045	RH % Average	2	single										x	
14047	Reserved	6	reserved										x	
10700	Burst Settings													
10701	Frequency Index													
10702	Burst Num Samples													
10703	Burst Channel 0													
10704	Burst Channel 1													
35100	Buzz Duration													
35101	Buzz 1													
35102	Buzz 2													
35103	Power													
35104	Inten 1													
35105	Inten 2													
40000	Script Settings													
40100	Script Debug													
40200	Script Stop													

40300	Script GO													
41004-41999	Script RAM													
42004-42999	Script App													
50000	General Config													
50000	PANID		u16											x
50001	Channel Mask	1		Bridge only, bit 0 = chan 11, bit 16 = chan 26										x
50100	Sleeping Mote Config													
50100	Process Interval	2	u16										f	f
50102	Sleep Time / Check In Time (ms)	2	u16										x	x
50104	Sleep Options	2	u16										x	x
50106	numChildCommFailures	2	u16										x	x
													x	x
50120	Network Password	8		byte 0 = Options, bytes 1-15 = Password									x	x
50400	Spontaneous config	1		First bit = 0(off), 1(on)										
50500	Pin Offset	1			x	x	x	f	x	x				
50501	Num Timers Enabled	1			x	x	x	x	x	x				
50502	Counter Mask	1			x	x	x	x	x	x				
50590	FIO Analog	1		Bitmask (1=Analog, 0=digital)	x	x			f	f				
50591	EIO Analog	1		Bitmask (1=Analog, 0=digital)	x	x			f	f				
50800	SWDT Settings	6					x	x						
50801	SWDT Time						x	x						
50802	DIO Response A						x	x						
50803	DIO Response B						x	x						
50804	DAC0 Response						x	x						
50805	DAC1 Response						x	x						
55000	Detailed Error	1					x						x	
55900	Reserved													
Ethernet Processor (56000-56999)														
56000	Firmware Version	1	u16								x			
56002	Buffer Status	12	u32											
56004	NumEthRX		u32											
56006	NumEthTX		u32											
56008	NumPIBRX		u32											
56010	NumPIBTX		u32											
56012	Num Mapped		u32											
	Eth Processor Last Error		u16											
	DM Overflows		u16											
	DM OVF During		u16											
	DM OVF During		u16											
	Num iPIB TOs		u16											
	Nun oPIB TOs		u16											
56100	Settings (Read only)	12	u32								x			
56102	Current IP		u32								x			

56104	Current Subnet		u32								x			
56106	Current Gateway		u32								x			
56108	Current DNS		u32								x			
56110	Current Alt DNS		u32								x			
56150	Settings	12	u32								x	x		
56152	Default IP		u32								x	x		
56154	Default Subnet		u32								x	x		
56156	Default Gateway		u32								x	x		
56158	Reserved		u32								x	Filler		
56160	Reserved		u32								x	Filler		
56200	MAC (802.3, 6-byte)	3												
56900	Flash Lock Key													
56902	Flash Erase	2												
56904	Flash Pointer													
56906	Flash Write	18												
56908	Flash Read													
56998	W = Enter flash mode / R = BL Version													
56999	Restart (4C4A)													
USB Processor (57000-57999)														
57000	USB Processor Firmware Version	1	2 bytes	[MSB, LSB]							x			
57001	USB Proc. Buffer Status	1	u16								x			
57002	NumUSBRX	2	u32								x			
57004	NumUSBTX	2	u32								x			
57006	NumPIBRX	2	u32								x			
57008	NumPIBTX	2	u32								x			
57010	USB Processor Last Error	1	u16								x			
57011	DM Overflows	1	u16								x			
57012	DM OVF During PIB TRNX	1	u16								x			
57013	DM OVF During USB TRNX	1	u16								x			
57014	Num PIB TOs	1	u16								x			
57015	Nun USB TOs	1	u16								x			
57016	Num Spontaneous Sent	2	u32											
57018	Num Spontaneous Dropped	2	u32											
57020	Error History (last 16 errors)		u16								x			
57050	VUSB	2	single								P			
57052	VJack	2	single								x			
57054	VST	2	single								x			
57998	W = Enter flash mode / R = BL Version		u16											
57999	Restart (4C4A)		u16											
58000	Device Name (USB String Format)			Writing Byte 0 erases			x	x						
59000	Flash Lock Key	2										x	x	x
59020	0xAA55	1												x
59021	0xC33C	1												x

59022	Address to Write	2												x
59024	Start of 32 bytes of data	1-40		(16 registers long)										x
59074	0xAA55													x
59075	0xC33C													x
59076	Block to Erase			2 registers										x
59080	Start of Image													x
59082	Length of Image (Bytes)													x
59084	Start of 20-byte checksum													x
59100	Read Address	2										x	x	x
59120	Data from flash										x		x	
59198	Change mode (0=norm, 1=flash)													
59200	Num Known Devices	2	u32								x			
59202	Device 0 ID	1	u16								x			
59329	Device 127 ID	1	u16								x			
59400	Network Channel	1	u16										x	
59410	Energy Scan	8		16 bytes, byte0 = chan11, takes 2212 ms										
59940	Wireless to UART ping test													
59950	AIN binary / Num Samples													
59952	AIN Max / Range													
59953	AIN Min / JN_Chn (0-3)													
59954	AIN Avg /													
59955	AIN Standard Deviation /													
59980	Set Hardware Config												x	x
59988	Range Test Mode. Num Minutes												x	p
59989	Hi Power Mode. Num Minutes													
59990	Rapid mode timeout (Minutes)	1												
59999	Reset (0x4C4A)													x
60000-60999	User Mem	16					x							
61000-61999	Calibration Data	16					x							
64000-64007	AES Key				n		x	x						
64008-64009	IP Address for CloudDot	2	string		n		x	x						
64010	Port for CloudDot	1	int		n		x	x						
64016	Heartbeat				n		x	x						
65000	Product ID	1	int		x		x		x					
65001	Serial Number	2	int		x		x		x					
65003	Local ID	1	int		x		x		x					
65004	Hardware Version	1	int		x		x		x					
65005	Hardware Options	1	int		x		x		x					
65006	Firmware Version	1	spf		x		x		x				x	
65007	Bootloader Version	1	spf		x		x		x					
65008	Other Versions	1	spf				x							

65100	Product ID	2											x	
65102	Firmware Version	1											x	
65103	Unit ID	1											x	x
65104	MAC (805.15.4, 8-byte)	4											x	
65108	Serial Number	2											x	
65110	Device Name	16									x	x	x	x