README.md

```
# Modbus DSLink

## License

GNU GPL

## Connection Guide



Use the action `add ip connection` or `add serial connection` to set up a
connection

### Add IP Connection
 - `name` - a name for the connection
 - `transport type` - transport protocol to use - will almost always be
TCP
 - `host` - IP address that your Modbus devices are at
 - `port` - port used by the devices for Modbus communication - 502 is the
protocol standard
 - `Timeout` - timeout for Modbus requests, in milliseconds
 - `retries` - how many attempts to make to open the connection
 - `max read bit count` - maximum number of bits that can be read (from
coils or discrete inputs) in one request
 - `max read register count` - maximum number of (holding or input)
registers that can be read in one request
 - `max write register count` - maximum number of holding registers that
can be written in one request
 - `discard data delay` - If set to a value greater than 0, will cause
data to be discarded after that many milliseconds have passed since it
arrived
 - `use multiple write commands` - When to use "write multiple
coils/registers" commands (function codes 15 and 16) vs "write single
coil/register" commands (function codes 5 and 6). Options are:
   - `Always` - will use function code 15 or 16 for all writes. Use this
if your device only supports multiple-write commands
   - `Never` - will use function code 5 or 6 for all writes, sending
multiple such commands if the value spans multiple registers. Use this if
your device doesn't support multiple-write commands.
   - `As Appropriate` - will use function code 5 or 6 for 1-register
values and function code 15 or 16 for multi-register values.

Most of the time, you can use the default values for all of the parameters
except `name` and `host`.

### Add Serial Connection
```

```
 - `name` - a name for the connection
 - `transport type` - transport protocol to use - will usually be RTU
 - `comm port id` - serial port to connect to
     - The DSLink should automatically detect any available serial ports, allowing you to choose one from a drop-down menu
     - If you don't see your serial port in the drop-down, try invoking the `scan for serial ports` action
     - If no serial ports are found, you can enter the name of your serial port manually here
 - `comm port id (manual entry)` - If the serial port you want to use is not in the drop-down, enter it here. Otherwise, leave this field blank.
     - If this field is not blank, then its value will be used instead of the selection from the drop-down
 - `baud rate` - the baud rate of the serial connection
 - `data bits` - the data bits of the serial connection
 - `stop bits` - the stop bits of the serial connection
 - `parity` - the parity of the serial connection
 - `Timeout` - timeout for Modbus requests, in milliseconds
 - `retries` - how many attempts to make to open the connection
 - `max read bit count` - maximum number of bits that can be read (from coils or discrete inputs) in one request
 - `max read register count` - maximum number of (holding or input) registers that can be read in one request
 - `max write register count` - maximum number of holding registers that can be written in one request
 - `discard data delay` - If set to a value greater than 0, will cause data to be discarded after that many milliseconds have passed since it arrived
  - `use multiple write commands` - When to use "write multiple coils/registers" commands (function codes 15 and 16) vs "write single coil/register" commands (function codes 5 and 6). Options are:
   - `Always` - will use function code 15 or 16 for all writes. Use this if your device only supports multiple-write commands
   - `Never` - will use function code 5 or 6 for all writes, sending multiple such commands if the value spans multiple registers. Use this if your device doesn't support multiple-write commands.
   - `As Appropriate` - will use function code 5 or 6 for 1-register values and function code 15 or 16 for multi-register values.

Most of the time, you can use the default values for all of the parameters except `name`, `comm port id`, `baud rate`, `data bits`, `stop bits`, and `parity`.

### Add device
Now that you have a connection set up, its node should have a child called `Connection Status`. If the connection was successful, this should have a value of `Connected`. Now you can use the connection's `add ip device` or `add serial device` action to connect to a device.
```

```
 - `name` - a name for the device
 - `slave id` - the device's ID number
 - `polling interval` - in seconds, how often the DSLink should poll the
device for the values of points that you're subscribed to
 - `zero on failed poll` - whether the DSLink should display a value of
`0` (or `false` for booleans) when it fails to get a point's value
 - `use batch polling` - whether the DSLink should use batch read requests
to get values from the device
 - `contiguous batch requests only` - affects how read requests are
batched, if `use batch polling` is set to true.
     - Some Modbus devices have non-contiguous sets of values within a
single register range, potentially causing error responses to batch read
requests
     - Setting this parameter to `true` will ensure that this doesn't
happen, by partitioning requests into only contiguous sets
     - This is generally not very efficient, so only set this to `true` if
you are seeing errors when trying to read points

### Add point
Now that you've added a device, you can begin adding its points into the
DSA tree. You can use the `add folder` action of the device node (or a
folder node) to add a folder child node. This is purely for organizational
purposes. You can use the `add point` action of the device node or a
folder node to add a point whose value you want to track.
You will likely need to consult the device's register map in order to know
the parameters of your device's points. Register maps vary significantly
in format, so some trial and error may be necessary to figure it out.

 - `name` - a name for the data point
 - `type` - the type of Modbus entity (coil, discrete input, input
register, or holding register) represented by this point
     - If you don't know which to use, see the _Point Addressing_ section
below
 - `offset` - the address of this point (See _Point Addressing_ section
below)
 - `number of registers` - the number of registers that this point takes
up
     - only applies to string data types (`CHARSTRING` and
`VARCHARSTRING`)
     - for non-string data types, number of registers is inferred from the
type
 - `data type` - the data type of this point (See _Data Types_ section
below for more information)
 - `bit` - which bit of the register contains the value
     - only applies to packed booleans (`BOOLEAN` data type and either
`INPUT` or `HOLDING` type), where 16 boolean values are stored in a single
16-bit register
```

```
 - `scaling` - number to divide the value by before displaying it
 - `scaling offset` - number to add to the value before displaying it
 - `writable` - whether or not the point is writable
     - controls whether or not the DSLink will allow you to try to write
to this point

#### Point Addressing
(The information in this section was taken from [the Wikipedia article on
Modbus](https://en.wikipedia.org/wiki/Modbus#Coil.2C_discrete_input.2C_inp
ut_register.2C_holding_register_numbers_and_addresses))

The way in which register maps display the "address" of a Modbus entity
can vary. There is a distinction between an entity's "number" and its
"address", and many register maps display the _number_. The _number_ uses
its first digit to indicate the type of entity, and the rest to indicate
the _address_ (or "offset"). In the traditional standard, the numbering is
as follows:

 - coils _numbers_ start with **0** and span from 00001 to 09999,
 - discrete input _numbers_ start with **1** and span from 10001 to 19999,
 - input register _numbers_ start with **3** and span from 30001 to 39999,
 - holding register _numbers_ start with **4** and span from 40001 to
49999.

The _address_ (_offset_) of a point is the offset of its number from the
lowest number of its type. So, for example, if a point has _number_
`40100`, then its type is `HOLDING` and its _offset_ is `99`.

This allows for addresses to be anywhere between 0 and 9998. Since the
_offset_ field of a Modbus request frame is 16 bits long, some devices
forego the traditional numbering in favor of an extended referencing, in
which each number has 6 digits, as follows:

 - coil _numbers_ span from 000001 to 065536,
 - discrete input _numbers_ span from 100001 to 165536,
 - input register _numbers_ span from 300001 to 365536,
 - holding register _numbers_ span from 400001 to 465536.

#### Data Types
Some information on the available data types:

 - Modbus registers are 16-bit, so 16-bit values take up one register, 32-
bit values take up two, and so on.
 - INTX types represent signed X-bit integers
 - UINTX types represent unsigned X-bit integers
 - FLOATX types represent X-bit floating-point numbers
 - BCDX types represent X-bit [binary-coded
decimal](https://en.wikipedia.org/wiki/Binary-coded_decimal) numbers
```

```
 - INT32M10K and UINT32M10K types represent "modulo 10000" 32-bit
integers,  where the more significant register contains `value/10000` and
the less significant register contains `value%10000`
 - There is some ambiguity in numeric types as to which bytes are more or
less significant. Normally, the first register is most significant, and
within each register, the first byte is most significant. When this is not
the case, you should use one of the data types that end in "SWAP".
     - The ordering of bytes from most to least signficant is as follows
(`rX[Y]` represents the Yth byte of the Xth register):
         - INT16, UINT16, and BCD16: `r0[0], r0[1]`
         - INT16SWAP and UINT16SWAP: `r0[1], r0[0]`
         - INT32, UINT32, FLOAT32, BCD32, and INT32M10K: `r0[0], r0[1],
r1[0], r1[1]`
         - INT32SWAP, UINT32SWAP, FLOAT32SWAP, BCD32SWAP, and
INT32M10KSWAP: `r1[0], r1[1], r0[0], r0[1]`
         - INT32SWAPSWAP and UINT32SWAPSWAP: `r1[1], r1[0], r0[1], r0[0]`
         - INT64, UINT64, and FLOAT64: `r0[0], r0[1], r1[0], r1[1], r2[0],
r2[1], r3[0], r3[1]`
         - INT64SWAP, UINT64SWAP, and FLOAT64SWAP: `r3[0], r3[1], r2[0],
r2[1], r1[0], r1[1], r0[0], r0[1]`
```