**Power Decoder**

Written By: Westin Breger, Andy Holdener, and Stephen Jarnagin POWER Engineering Contact: Keith Gray

[Type the abstract of the document here. The abstract is typically a short summary of the contents of the document. Type the abstract of the document here. The abstract is typically a short summary of the contents of the document.]

Contents

[Overview 2](#_Toc354407370)

[Problem Executive Summary 2](#_Toc354407371)

[A. High-Level Parser 5](#_Toc354407372)

[B. DNP3 Parser/Translator 5](#_Toc354407373)

[C. Modbus Parser/Translator 6](#_Toc354407374)

[D. User Input 7](#_Toc354407375)

[E. Interface Design 9](#_Toc354407376)

[F. Tree-view Messages 9](#_Toc354407377)

[**Non-functional requirements** 10](#_Toc354407378)

[Scheduling 12](#_Toc354407379)

[ Milestones 12](#_Toc354407380)

[1st Semester Milestones 12](#_Toc354407381)

[Date Completed 12](#_Toc354407382)

[Completion Status 12](#_Toc354407383)

[ Gantt 14](#_Toc354407384)

[Glossary 18](#_Toc354407385)

# Overview

In the course of servicing power transmission equipment, a POWER engineer may have to interpret raw information that the machine in question is broadcasting. It is our job to create a program that will make this task much easier by quickly and accurately translating this information into organized and readable points of data. The input for our application will come from a log file captured from another program, which our program can then process. Once our application finishes translating the user’s inputted data, it will display a tree of data points, arranged by transmission times, that tells the user detailed information about each message in the input file.



Power Engineers at work, courtesy of Power Engineers, www.powereng.com



# Problem Executive Summary

POWER Engineers, among other development projects, develops and integrates various electrical devices into buildings and equipment for clients. The integration process uses network protocols as a standard to communicate between the devices. POWER Engineers uses two network protocols to do this: Distributed Network Protocol version 3 (DNP3), and Modbus. These network protocols act as a baseline between the devices to communicate successes, tests, and in other cases failures between the devices.

During the integration among the electrical devices, the devices use network protocols to transmit the error messages that can occur, between or among the devices being integrated, to the engineer. Once the POWER Engineer receives the message, he must manually decode this it using a set of predefined, but not universally standardized, tables. These tables contain the meanings for all messages that can be generated from the current set of devices. Another method of translating the messages requires the use of a hardware interpreter that is available to the engineer. The problem is that this device is limited in its quantity and is not readily available to the majority of the engineers in the field. This leaves manual translation as the main method for decoding the messages. For the POWER Engineer, this takes approximately ten minutes due to the size of the table and obscurity of the format it is in. This is the core of the problem which our product, the decoder, is being created to solve for POWER Engineers. The time it takes to translate the error message may be minimal by itself, but when this issue is compounded by multiple errors being communicated by not just one device but multiple devices, the time it takes to solve a problem can escalate into a hour or more rather than a few minutes. Another issue that is coupled with time is the rate of complexity that can be generated. If the error message being translated is one that points to an error in another device, then this can create a moving target problem where the engineer will be going back and forth between devices trying to figure out the systemic issue or issues that may be plaguing the integration process as a whole.

Our product will solve this problem by automatically translating these messages for the POWER Engineer and displaying detailed information about each one to him. This means that, when using this product to translate his captured messages, all he has to do is take the given index from the message he’s interested in and look it up in one of his tables. After he does this, he’ll know what the message was intended to do and he won’t have wasted much time figuring it out.

**Functional Requirements**

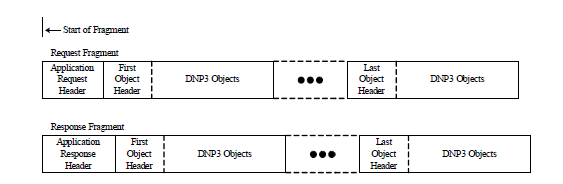
## A. High-Level Parser

In order for our application to accept a wide variety of user-inputted text and to assist the specific protocol parsers as well as the user, we are sending the user-inputted text to the High-Level Parser before having either of the other parsers look at it. This is used when the user does not know the origin of the protocol the message is written in and or is uncertain to the messages’ accuracy. The High-Level Parser initially goes through all of the user-inputted text and makes a list containing all of the DNP3 and Modbus messages while ignoring the rest of the text. This allows for users to have extra data in their input without being penalized and it also gives users the option to use message-capturing software if they choose to do so.

Once the High-Level Parser finishes its initial pass through the user-inputted data, it will then go through the list of messages that it found, and it pass each message to its corresponding protocol parser to be decoded. If the user has selected the default option of “Automatic Parsing”, then the High-Level Parser will determine which protocol parser will decode which messages; otherwise, each message will be sent through the protocol parser that the user has selected. We will check the users’ input to see if each message follows the DNP3 or Modbus standard before decoding them in order to significantly reduce the number of times that the DNP3 Parser and Modbus Parser fail to decode messages and also to ignore all excess input, which allows for the user to use message-capturing software on a network to automatically generate text files to use for the input.

## B. DNP3 Parser/Translator

This product will be able to parse and translate Hexadecimal words that were generated by devices using the DNP3 networking protocol. This product will eventually need to recognize the full range of DNP3 words and structures, and be able to differentiate between ModBus and DNP3.

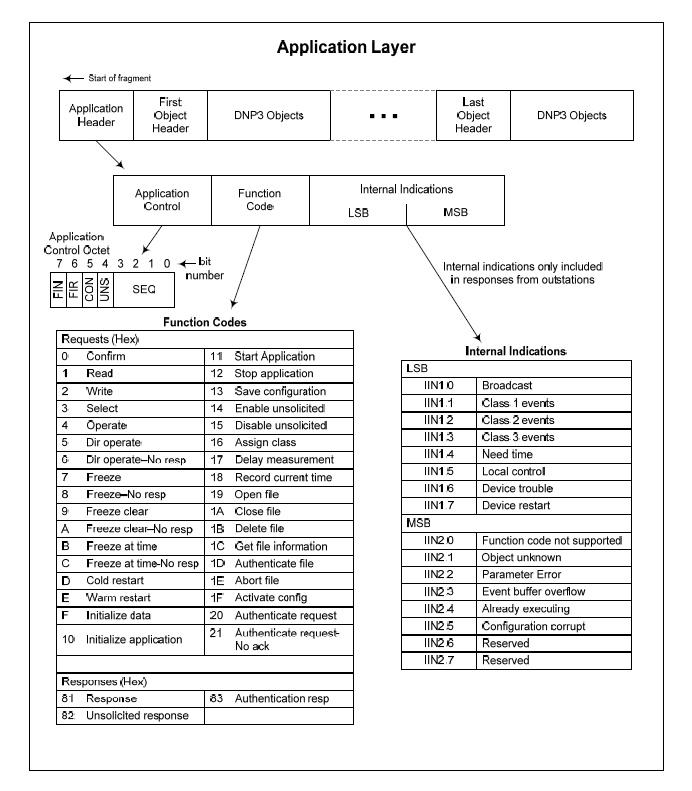


*Figure 1A: DNP3 fragment bit layout. Note that the three dots represent additional DNP3 objects*

A “fragment” is a variable-length hexadecimal word generated by the DNP3 networking protocol that can contain data commands (such as read or write), and data. A fragment can contain any amount of data, and fragments can be chained together in cases where devices can only handle a limited length of words.

Our program will need to correctly determine what sort of command DNP is issuing (such as read, write, or wait), the circumstances of the message (first message, part of a chain, response message), and the data contained within. In the case of chained data, our program will also need to combine information as logically as possible, and present this data to the GUI in a clear fashion.

Data is contained in DNP3 transmission objects. These objects are standard objects for DNP3 that contain and describe the data within. For example: DNP3 has a set of objects specifically for describing time, device information, and GPS data. It is important to note, however, that there are a couple “standard-nonstandard” objects that DNP3 allows for, which gives programmers the option of sending data that wasn’t originally foreseen. As such, it is impossible for a translator to make sense of this data without context, so we will allow the user to filter this data in various ways, such as binary, hexadecimal, and decimal.



*Fig 1B: Detail view of DNP3 fragment and function codes.*

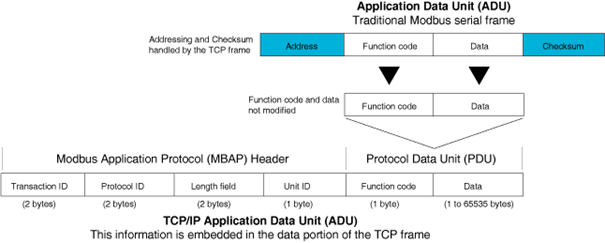
## C. Modbus Parser/Translator

This product should be able to detect and decode input that is in the form of the network protocol Modbus. In order to detect the input, a standard algorithm must be used to detect any prefixes that would identify the input to be that of Modbus origins.



*Figure 1B: Modbus example message format*

The reason this identification process is necessary is because the decoder can receive the input and can translate it automatically. Once the input is considered to fall under the Modbus network protocol, the decoder will then begin read each section of the input received. The way the decoder would do this is by checking for separators while following the rules of Modbus.

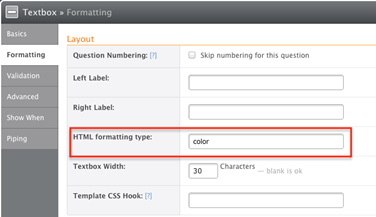


*Figure 2B: Modbus Protocol Breakdown*

Each separator would act as an identifier to which section and which rule must be checked and then parsed out. The reason for this is that in order to effectively parse the correct outputs, the rules for decoding Modbus must be followed in a particular order from left to right with separators acting as a closure or end of a rule. Once the input is successfully read in and effectively parsed following the protocol of Modbus, an output is generated in Modbus format to the user. The purpose of this is so that the user can then take that output and check a predefined table of solutions for the translation of that output.

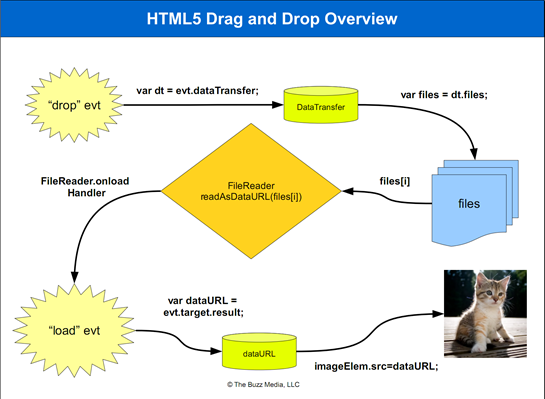
## D. User Input

In order to generate the appropriate error messages for the translation tables, the user must input their error messages. This can be done in two ways. The first way is manual input with the user inputting his/her error message.



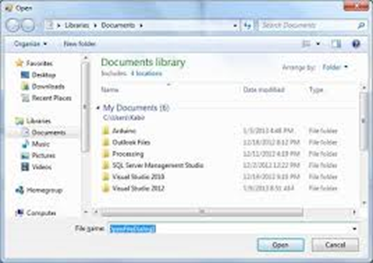
*Figure 1C: Textbox Design Example*

This is accomplished through allowing the user to input their information into a text box and then allowing that user to click a button to notify the decoder that the translation is ready to proceed. An addition to this allowing the user to ‘drag and drop’ his/her set of data into the textbox and then clicking the button to proceed. The ‘drag and drop’ feature is implemented using html 5.



*Figure 2C: Drag and Drop Explanation*

The other way the user can input his/her data is to open a file and have that file automatically read into the textbox and decoded. This is accomplished through creating an open button and having the decoder search for files in a .txt format within a dialog box. The user can then select a file and click open in the dialog box.



*Figure 3C: Open dialog box example*

Once this is done, the decoder will close the dialog box and then proceed to transfer that file’s information into the text box. The user can then select a button to notify the decoder to begin translations.

## E. Interface Design

A display is necessary to perform three tasks for the user: to let the user know when their task was completed, to present any information in a readable format, and to allow the user ample ability to interact with the decoder. The first step in developing the environment to fit the users’ needs is to develop an interface. This is accomplished by creating a background, a title, a textbox for input and one for output, radio buttons for selecting variations of a task, an open button to open other files, a dialog box following the open button, and a main button to initiate the decoding. Once these objects are created, the functionality of these objects are then tied to the back-end coding of the decoder itself. If and when the user interacts with the interface, the interface would then change to reflect those decisions, the most common change being the results of decoding the message.



Figure 2E: Phase 1 Interface

## F. Tree-view Messages

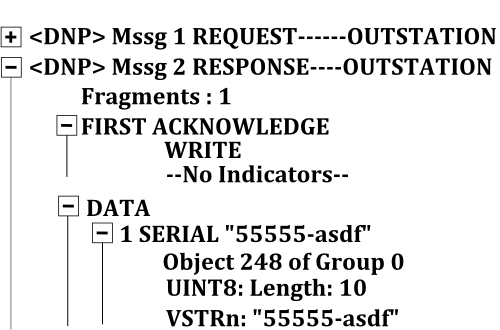
 Since we’re allowing users to enter multiple messages to be decoded at the same time, our application will display each message as an expandable object in order to save space on the users’ screen. The messages will look like the example shown to the right. All the messages will be shown at once, without taking up too much room, and when a user wants to see more information about a specific message, he/she will click the plus symbol next to the messages he/she wants to know more about, and it’ll bring up detailed information below the message. When the user is done examining a specific message, he/she can click the minus symbol and it’ll collapse the message to save space.

Figure 1F: Tree View Example

# **Non-functional requirements**



* Our application will be distributed via an installer
  + We are going to use Pip to install our application onto our users’ computers.
  + We will create a shortcut on our users’ desktop and whenever they want to run our application, they can double-click the shortcut and it’ll open the default web browser to our application running on their local host.
* Python 3.3 is the programming platform
  + We’re coding our application in Python because we all know Python fairly well and so does our client, so he should be able to read through our code if any problems arise in the future, after we’ve delivered our final product.
* Django 1.5 will serve the web pages locally, and tentatively over the internet in the future.
  + Django is a web server that also deals in web scripting. Django runs in and with Python 3.x, which affords the portability and power of Python. Running on a web server also allows the website to detect mobile platforms and modify content as needed.
* Our application will run through web pages on local host that will be tested for Internet Explorer 8 and 9, Firefox (20), and Chrome (25)
  + Output website will run javascript 1.8.5 and HTML5
  + Explorer 10 will be developed by the client
* Upon our client’s request, our application will be released under the GPL v.3 license
  + The GPL licensing is a form of license that allows users to modify the code at will on legally obtained versions of this program. POWER Engineers retains the right to sell and distribute the code as they see fit. If this program is used in future versions or other software, that program(s) also inherit GPL v.3.
* To ensure that our product runs properly with most cases, we are going to test all of our Python code with PyUnit.

High Fidelity Prototype

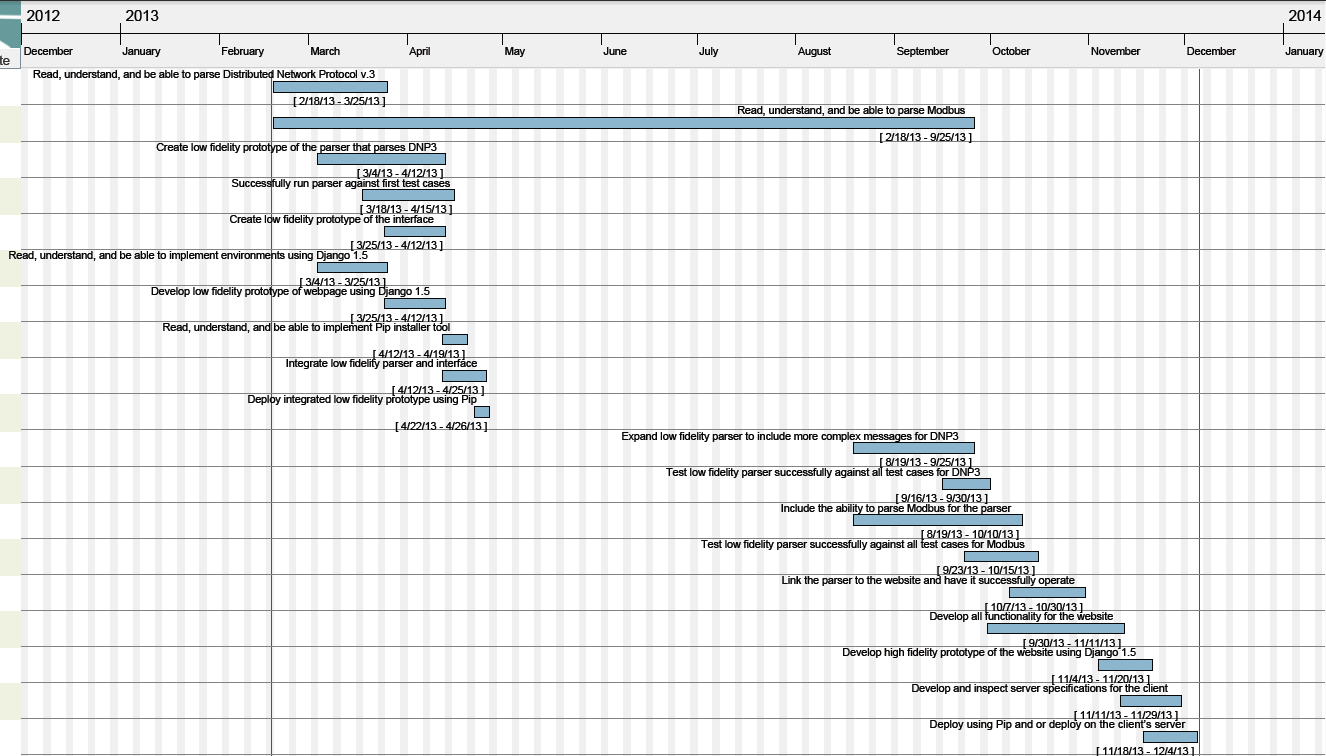
# Scheduling

# Milestones

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1st Semester Milestones | Date Completed | Completion Status |
| **1** | **Read, understand, and be able to parse Distributed Network Protocol v.3** | **3/25/2013** | **pending** |
| **2** | **Read, understand, and be able to parse Modbus** | **9/25/2013** | **pending** |
| **3** | **Create low fidelity prototype of the parser that parses DNP3** | **4/12/2013** | **complete** |
| **4** | **Successfully run parser against first test cases** | **4/15/2013** | **complete** |
| **5** | **Create low fidelity prototype of the interface** | **4/12/2013** | **complete** |
| **6** | **Read, understand, and be able to implement environments using Django 1.5** | **3/25/2013** | **pending** |
| **7** | **Develop low fidelity prototype of webpage using Django 1.5** | **4/12/2013** | **complete** |
| **8** | **Read, understand, and be able to implement Pip installer tool** | **4/20/2013** | **pending** |
| **9** | **Integrate low fidelity parser and interface** | **4/25/2013** | **pending** |
| **10** | **Deploy integrated low fidelity prototype using Pip** | **4/26/2013** | **pending** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 2nd Semester Milestones | Date Completed | Completion Status |
| **11** | **Expand low fidelity parser to include more complex messages for DNP3** | **9/25/2013** | **pending** |
| **12** | **Test low fidelity parser successfully against all test cases for DNP3** | **9/30/2013** | **pending** |
| **13** | **Include the ability to parse Modbus for the parser** | **10/10/2013** | **pending** |
| **14** | **Test low fidelity parser successfully against all test cases for Modbus** | **10/15/2013** | **pending** |
| **15** | **Link the parser to the website and have it successfully operate** | **10/30/2013** | **pending** |
| **16** | **Develop all functionality for the website** | **11/10/2013** | **pending** |
| **17** | **Develop high fidelity prototype of the website using Django 1.5** | **11/20/2013** | **pending** |
| **18** | **Develop and inspect server specifications for the client** | **11/30/2013** | **pending** |
| **19** | **Deploy using Pip and or deploy on the client’s server** | **12/4/2013** | **pending** |

# Gantt



* Scheduled Meetings

|  |  |  |
| --- | --- | --- |
| Meeting With | Time | Day |
| Client | 3:30 - 4:30: PM | Tuesday (bi-weekly) |
| Team | 3:30 - 5:30: PM | Thursday |

Team Process

Peer Review

* Effort Estimation per Deliverable
  + Individual Effort
  + Meeting Minutes

//Just include the greensheet minutes a few of the saved notes

* + Individual Responsibilities
    - Stephen-wrote problem executive summary, Modbus parser feature, Interface features, partial functional requirements, scheduling section, bibliography.
    - Westin wrote: summaries, DNP and Modbus descriptions, prototype DNP parser, beginning of Django website.
  + Version Control Summary
    - Our git commits can be viewed in detail at:
      * <https://github.com/xrabohrok/dnp-modbus-decoder>
* Aggregate Estimated Effort
  + Individual Effort
  + Individual Responsibilities
  + Version Control Summary

# Glossary

* Intelligent Electronic Device (IED): Generic Electrical Engineering term for devices that have a microprocessor. Power transformers, circuit breakers, and similar equipment are often IEDs.
* Remote Terminal Unit (RTU): Device whose primary function is to gather and transmit data regarding the operational status of another device.
* Distributed Network Protocol (DNP or DNP3): DNP is a network protocol used primarily in the Utilities industry. It is designed for transmitting and receiving data on irregular intervals of unknown size, on a large scale. It is more robust that MODBUS, but is not open source.
* MODBUS (not an acronym): MODBUS is an open-source network protocol standard used primarily in the Utilities industry. MODBUS allows for large-scale transmission and gathering of data. MODBUS is simpler to use than DNP3, but transmissions have a fixed size, so it is not as flexible.
* Accumulator: A device that gathers and saves data about something is a accumulator. The data is usually passed off to a RTU to be transmitted to a master station.
* Master Station: Computer system that receives data from RTU’s to be interpreted.

Bibliography