InfraModel: An Interactive Electrical Modeling System Using Common, Commercially Available Software

Walt Akers*, Paul Powers, Jason Willoughby
Thomas Jefferson National Accelerator Facility, 12000 Jefferson Avenue, Newport News, VA 23696

April 23, 2018

I. INTRODUCTION

Jefferson Lab has an electrical infrastructure that distributes power from two utility feeds with a rated capacity of 56 MVA each. Power is distributed through dozens of large transformers and into hundreds of distribution panels. In addition to the nearly 100 buildings on campus, this electrical system also supports the Continuous Electron Beam Accelerator Facility (CEBAF), a Low Energy Recirculating Facility (LERF), and four experimental halls where users conduct the Lab's experimental nuclear physics program. Each of these areas has unique requirements and is regularly reconfigured to support new activities.

This continuously changing configuration makes it difficult to maintain and update electrical documentation to accurately reflect the current state of the system. Further, while some metering has been introduced in panels at the top of the hierarchy, it is not economically feasible to extend metering throughout the entire system. As a result, system documentation is often incomplete or inaccurate, making it difficult to determine the amount of power consumed in various regions, or to plan for future installations which could exceed the capacity of the system.

Proprietary software exists that addresses some of the issues involved in managing a large electrical installation. However, these packages are generally targeted at highly specific problems, such as arc-flash calculations, harmonic analysis, and protective device coordination. The specificity, cost and complexity of these programs inhibits their adoption to address the fundamental configuration management issues which occur in most large electrical installations. This leaves many facilities in the unfortunate position of managing their electrical configuration using a disjointed collection of panel schedules, spreadsheets, and static one-line diagrams which are supplemented with meters.

Despite the scale of the infrastructure being managed, the configuration management issues most organizations face are quite fundamental. In the simplest terms, these are issues of capacity and distribution. To address these challenges, an institution has the choice of a) investing in proprietary systems which require a highly trained workforce to implement and use, b) developing local software solutions which require a highly skilled workforce to construct and maintain, or c) managing their configuration using schematics, documentation and electronic records which are unlikely to ever leave the file room. In this paper we discuss a fourth, hybrid option which leverages the emerging capabilities of common office applications to produce *smart-documents* targeted at specific configuration management issues.

II. OBJECTIVES

In approaching this problem, the first step is to identify the specific problems that must be resolved, and then to identify a software platform that can be used to address these issues with a minimal development effort. While the preliminary objectives discussed here are geared to address electrical distribution, the idea of extending this product to broader infrastructure issues (HVAC, process cooling water, cryogenics) remains prominent. Still, this document will focus on the requirements that the initial product must satisfy and will use those requirements to identify and justify the selected software platform.

General Requirements

a. **Appropriateness of Scope: Specificity**

In systems theory, there is a balance that exists between concrete specificity and abstraction. For instance, a model that is developed with the goal of exactly representing reality must, necessarily, contain all known information and detail. The reader can infer, therefore, that even for the simplest system, the vastness of information that is known (or knowable) can quickly overwhelm the senses. This avalanche of information immediately creates the conditions for confusion because, as described in the Law of Requisite Saliency (Warfield, 1993), *not all information*

within a system is of equal importance. Worse yet, Boulding's (1966) concept of "spurious saliency" dictates that when faced with an overwhelming volume of information, *individuals will often assign importance to the wrong things*. A good model, therefore, must choose which information is required to address the problem at hand, and which information is superfluous. Having resolved this, the critical information must be presented with specificity, while peripheral information must be abstracted, or better yet, eliminated.

b. **Appropriateness of Scope: Abstraction**

Even once all peripheral information has been eliminated, a complicated infrastructure system may still have an overwhelming volume of data. In this case, it becomes the responsibility of the model (and the modeler) to develop mechanisms of abstraction that allow the data to be viewed through different *lenses*. These *lenses* are simply abstractions that allow data that is momentarily non-essential to be removed from the current perspective. This creates an environment where the most salient issues are brought to the fore.

It should be noted, that good abstraction does not eliminate pertinent data. On the contrary, it synthesizes and digests it, to provide a clearer perspective of the problem at hand.

c. **Ease of Use: Representation**

Whatever model is created must have an immediate, clear and meaningful relationship to the system being represented. While metaphor driven models are often useful for introducing new concepts via well-known constructs, it should be stipulated that individuals who are managing a large infrastructure already understand the basic tenets of their system. As an illustration, the use of the tree diagram in *Figure 1* to represent a hierarchy may benefit those with no knowledge of the topic, but it adds little or no value for those who already understand the shape, structure and the nature of relationships within the system.

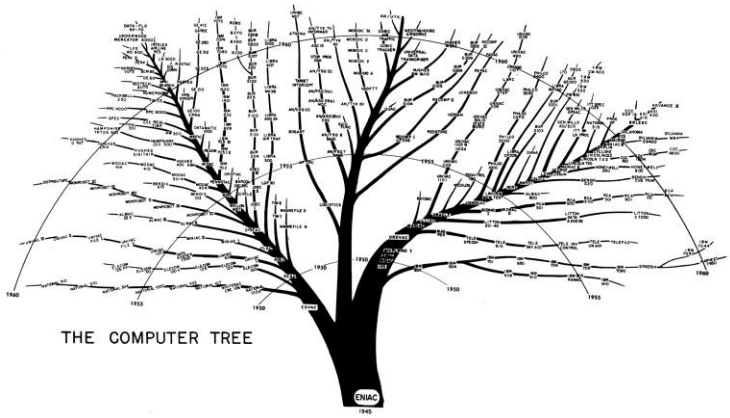

Figure 1. The Computer Tree (Kempf, 1961) employs an unrelated metaphor to describe the developmental history of computing systems.

Infrastructure models should seek to emulate the actual structure of the system being modeled. This approach allows the user to infer the physical topography of the system using the same model that describes the distribution and flow of resources within it.

d. **Ease of Use: Interface**

The interface that is provided for creating, manipulating and viewing infrastructure models should be consistent with the interfaces that are in common use. In a perfect implementation, the user of a model would be able to apply only the techniques that are commonly used in existing applications, with the modeling application introducing no new or novel interfaces.

Non-standard interface operations should either be eliminated or should be concealed within the underlying software. This is not to suggest that common techniques such as *hot-keys* should not be employed. On the contrary, these techniques can be used, but they should conform to the standard key combinations in common use on the platform.

e. **Ease of Maintenance: Code Minimization**

For software developers, there is always a temptation to address emerging issues by developing a new application or library. If done properly, embedded code has the ability to greatly enhance the speed, flexibility and efficiency of a model. Still, each line of code that is added to the modeling application is a line that will need to be maintained by the local organization in the future. By contrast, a system that exclusively uses capabilities that are intrinsic to the underlying application may continue to operate across many software releases and version updates with little or no intervention by local developers.

This being said, it is unlikely that any truly beneficial modeling system can be derived directly from a common application without the need for some software development. Therefore, any software that is introduced into the model should provide a highly justified benefit, a minimum amount of code, and be well documented.

Platform Considerations

A key decision in developing this solution was the selection of an appropriate application to use as the platform for developing the model and the modeling software. The considerations in selecting the software platform included: cost, availability, prevalence and how well it satisfied the general requirements discussed earlier.

The following applications were considered for the initial implementation of this infrastructure model.

a. **AutoDesk AutoCad**

Originally released in 1982 (Weisberg, 2011), AutoCad has immediate appeal when considering platforms to use for infrastructure modeling. As a drafting application, it has a long history and is widely used in both Facilities and Design departments. AutoCad has a programmable backend for creating custom entities and has support for languages such as LISP, VBA, C++ and C#.

There are, however, significant drawbacks to using AutoCad as a foundation for system modeling. First, the time required to learn to use this product ranges from months to years, depending on the motivation and skill of the user. The selection of AutoCad for system modeling would essentially demand that the individuals maintaining the model must be highly skilled in the use of the software, as well as have an in depth understanding of the system being modeled.

Further, at a current cost of $1,500 per year for a software license for AutoCad, the product quickly prices itself out of the market for use as a *simple, affordable* platform for modeling.

b. **Trimble Sketchup**

While not as venerable as AutoCad, Sketchup is an emerging power in the modeling community and is marketed under the banner *"3D modeling for everyone."* (Donley, 2011) The application is simple enough that most users can gain a working understanding by watching a few hours of online video instructions, and can be productive within a few weeks. Additionally, a license for Sketchup is economically priced at just $695 per seat. Still, there are challenges with using Sketchup as a platform for developing infrastructure models.

One major issue with employing Sketchup for this use is the fact that it is 3D all the time. While it is possible to create views of a 3D model using a *parallel projection* that makes them appear to be 2D, the complexity of the 3D universe is always present while drawing, updating or using these models.

Further, Sketchup does not support the concept of pages. Each model is a representation of the entire three dimensional universe, and can be as small or as vast as the user chooses. Limited views of this 3D universe can be created using hidden layers, but this is not an optimal solution for models that may span a variety of independent sub-systems.

Finally, Sketchup's components are programmed using an archaic Ruby interface. While the language is powerful and entities that are programmed in this interface are very useful, they often have idiosyncrasies and unexpected behaviors that are manifested when a programmed object is grouped, reshaped or copied.

None of this should be taken to denigrate Sketchup, which is an excellent modeling platform. None the less, these issues do negatively impact its usability for infrastructure modeling in this environment.

c. **Microsoft Visio**

Originally released in 1992 as Shapeware (Johnson, 2013), Visio was acquired in 2000 by the Microsoft Corporation and has been its premiere diagramming package since then. Currently priced at $500 for a Visio Professional license, the product is more economical than either AutoCad or Sketchup.

The real benefit that Visio provides for this application is it's totally conformance to the standard user interface used in all Microsoft products. This means that most users who are already familiar with the MS Windows platform will be able to immediately use this product. Additionally, this product has integrated support for Visual Basic, and can be easily integrated with C#, C++, C, Fortran or any other language

that can be used to create a Windows Dynamic Link Library.

Finally, the Visio platform is designed to support *Stencils,* which are small, task specific toolboxes of shapes. The pre-programmed objects within a stencil can be distributed and updated independently of the application.

Of course, the Visio platform is not without its drawbacks. The programming interface for shapes is spreadsheet driven and can be clumsy to use when attempting to manipulate several objects at the same time. In terms of documentation, the number of books that specifically discuss Visio programming are few, and most of those were written for older versions. Still, the product has a vibrant online community that tends to be highly responsive to programming questions.

d. **Inkscape (Scalable Vector Graphics)**

Another application that was considered as a platform for developing this modeling system was Inkscape. Inkscape is a freeware package that generates and manipulates Scalable Vector Graphic (SVG) files. SVG images use an XML-like text structure, are programmed with embedded java script and are directly viewable through most web browsers. Models created in this format have the benefit of being immediately accessible and updateable by a large, distributed group of people.

While Inkscape provides mechanisms for almost unlimited programmability, it requires that the developer have an intimate understanding of how the application works, as well as the underlying structure and behavior of the SVG files. Further, although the user interface for drawing is consistent with many common applications, the programming interface is largely focused on mouse events and any sophisticated programming must be performed in an external editor.

Platform Selection: Microsoft Visio

For the initial release of this package, we have chosen to use Microsoft Visio. While each of the other applications has benefits, Visio provides an immediate ease of use and design that allow this product to be developed and deployed quickly. Still, web accessibility remains a highly desirable capability; therefore, it is likely that an SVG interface will be integrated into future designs.

The remainder of this document will discuss the Visio objects that were created for use in electrical modeling, along with a description of their behaviors, capabilities and limitations.

III. OBJECTS AND INTERFACES

Design Perspective

As stated in the objectives, the primary goals of the design process include: ensuring that the user experience is as consistent as possible with commonly used applications; that the completed model has the right balance between specificity and generalization; that the visual representation is meaningful to a user who is versed in the subject matter; and that the modeling system remains tightly focused on the goal at hand, specifically, *modeling the distribution of power*.

The issue of maintaining focus is perhaps the most challenging (and limiting) of these concerns. There is always a temptation to allow any model to expand to encompass more and more functionality, in hopes that it will provide a better representation of reality. In truth though, the noise that is created by the addition of new features often overwhelms the system; preventing it from achieving the goals for which it was designed. To avoid this, the development of this modeling system will remain tightly focused on the distribution of three-phased, alternating current power in either 480Y/277V or 208Y/120V voltages.

Power distribution will be modeled through the use of a collection of entities that represent different elements in the power system. These include sources, distributions, breakers, switches and loads. All of these objects derive their base functionality from a fundamental entity – *the distribution object.* While they may add distinctive behaviors or visual representations, they all essentially distribute power from an incoming source to connected loads. Therefore, a discussion of these objects will begin with a description of the *distribution* object, and then will define all other entities by their variations from that base class.

The Distribution Object

As discussed earlier, the distribution object contains characteristics that are common across all of the model elements. There are two types of data fields *(or attributes)* that are described below: those that are visible, and those that are hidden. The visible attributes are displayed as part of the model's onscreen interface, while the hidden attributes are only accessible through the objects *Shape Sheet* or from Visio's *Shape Data* window. These hidden attributes are typically used for computing the state of a distribution object or for summing the flow of power through the object.

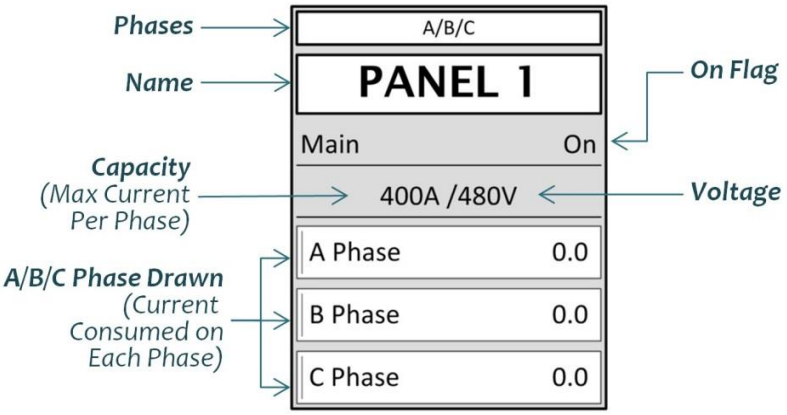

Figure 2. The Distribution Object

The following ***visible*** data attributes can be viewed from the model's onscreen interface.

a. **Name**

Each distribution object should have a unique name *(shown as Panel 1 in the figure)*. The name of a distribution object is typically the same as the name that is stamped on the outside of the physical panel or device.

The name field is included in the object's shape data and is the unique identifier that is used to reference the object when loading external data. One should note that each object also has an application generated identifier which is used by Visio to relate the objects to one another. The object name described here has no relationship to that identifier.

b. **Phases**

A distribution object receives power on any combination of the A, B or C phases. For instance, a single phase distribution object may receive its power from the A, B or C phase of its parent object. Likewise, a two phased distribution may receive power from the A/B, B/C or A/C phases of its parents. This is an important distinction because power is distributed (and accumulates load) along each phase. Retaining phase identity throughout the distribution allows the model to detect and display phase imbalances.

By extension, the user should note that a distribution object can only have the phases that are provided by its parent. Further, it can only distribute one or more of *those* phases to the loads that are connected to it. In the event that a distribution object is connected to a source that does not provide the phases it requires, it will immediately *trip-off,* and will not distribute power downstream.

If a connected load *(a child)* requires a power phase that is not provided by the distribution object, then that child will *trip-off*, but the distribution object will continue to provide power to other connected loads.

c. **Capacity**

Each distribution object has a capacity (in amps) that represents the maximum power that can be delivered along each individual phase. If the power drawn along any phase exceeds that capacity, then the object will *trip-off* and will no longer deliver power downstream to any connected object.

d. **Voltage**

In three-phase/four-wire systems, the voltage value for a distribution object will be either *"480V/277V"* or *"208V/120V"*. The incoming and outgoing voltage for a distribution object is always the same. A distribution object cannot be connected to a source *(a parent)* unless the voltage is the same. If a distribution object is connected to a source object with a different voltage, then the distribution object will *trip-off* and will not deliver power downstream. Likewise, if a child object has a different voltage than the distribution object, then the distribution object will remain functional, but the child will *trip-off*.

e. **On**

This is a simple binary switch that determines if the unit is switched on or off. Note that the "On" flag is not the sole requirement to ensure the

unit is delivering power downstream. In order to deliver power, the distribution object must meet three criteria:

i. The *On* flag must be TRUE
ii. The *Powered* flag must be TRUE
iii. The *Tripped* flag must be FALSE

f. **A Phase Drawn** ***[B,C]***

This is a floating point value that is used to display the current (in amps) that is actively being drawn on the A (or B, or C) phase if this object is on, powered and not tripped. This value is the sum of the hidden attributes: *A Phase Load* and *A Phase Local*.

The following are the hidden attributes of the distribution object. While the user might never directly manipulate these values, they provide the underlying functionality for the object. They are accessible using Visio's Shape Sheet or Shape Data windows.

g. **Distribution**

This is a simple integer field that identifies the object as a member of the family of distribution objects. This flag is present on sources, loads, switches, and any other object that is derived from the distribution object. The attribute is used by applications to confirm that an object being evaluated is a distribution object before performing operations on it.

h. **Powered**

The *Powered* flag is a binary switch that indicates that this distribution object is receiving incoming power from a source. For this flag to be TRUE this object must be fed by another distribution object *(the parent)* and the *Powering* flag on that parent object must be TRUE.

i. **Powering**

Unlike the *Powered* flag, the *Powering* flag indicates that this object is actively delivering (or is capable of delivering) power to objects that are connected downstream. This flag is TRUE if the following conditions are met.

i. The *On* flag is TRUE.
ii. The *Powered* flag is TRUE.
iii. The *Tripped* flag is FALSE.

j. **Phase Mismatch**

The *Phase Mismatch* flag is a binary switch that is TRUE if the *Phases* attribute of this distribution object is incompatible with the *Phases* attribute of its parent. *For instance, if this object is defined to have a "B" phase, but its parent provides only an "A" Phase.*

k. **Tripped**

The *Tripped* flag is a binary switch that will be TRUE if either of the following conditions are met:

i. The *Phase Mismatch* flag is TRUE.
ii. The load on any phase of this distribution object exceeds the object's *Capacity*.

l. **A Phase** ***[B,C]***

This field exists as a mechanism for determining how much power the object would consume if it were powered on. Like the *A Phase Drawn* field, the *A Phase* attribute is also the sum of the *A Phase Load* and *A Phase Local* attributes. The notable difference is that this field will compute the value even if the object is not on, is not powered, or is tripped.

m. **A Phase Local** ***[B.C]***

This is a floating point value that contains the current that is consumed by this object locally. This value is sometimes used to represent transformation costs, inefficiencies or other losses. Most often, though, this field is used in *load* objects to represent the amount of power that a terminal load *(such as a motor)* is consuming.

n. **A Phase Load** ***[B,C]***

This is a floating point value that contains the sum of all currents that are being drawn from the A Phase by objects that are downstream from this distribution object. For instance, if this distribution object has two objects connected to it that are drawing 20 amps and 60 amps respectively *(from the "A" phase)*, then it will have an A Phase Load of 80 amps

o. **A Phase Reading** ***[B,C]***

This is a floating point number that contains an actual reading that was taken from this device when it was in operation. This reading can be used to extrapolate loads throughout the model for computational purposes.

Some special considerations exist for this attribute. Specifically:

i. If the reading is greater than 0, then the model will assume that the current drawn along this phase by this and all connected children is equal to the reading.

ii. If the reading is 0, then the model will assume that no power is drawn along this phase by this object or any of its children.

iii. If the attribute is set to -1, then the model will assume that a reading does not exist for this object and it should be computed based on the characteristics of the attached loads.

p. **BTU/Hour**

This is the number of British thermal units that are generated per hour by this piece of equipment. The field is a floating point value that is computed using the voltage and the ***local*** load on all three phases. If this distribution object does not have any load indicated in the *A/B/C Phase Local* fields, then it will have no BTU/Hour. For the purpose of this model, *heat exists where the load exists.* Therefore, power can pass through any number of distribution objects, but it only generates heat at the terminal load unless a local load is indicated.

The Source Object

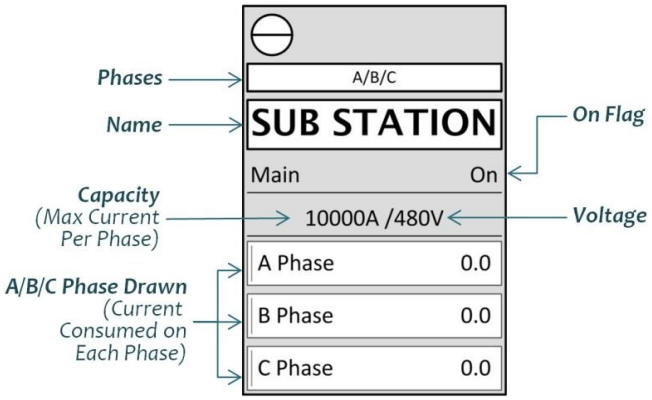

Figure 3. The Source Object

The source object is at the uppermost level in a power distribution model. These objects typically represent generators, sub-stations, or power plants. The source object is functional identical to the distribution object with a few minor exceptions in the following fields.

a. **Capacity**

This value is not just the amount of current that this object can deliver or its breaker capacity – it is the amount of power that is provided by this power source.

b. **Powered** *(Hidden)*

The *Powered* flag on a source object is always TRUE. It does not need to be connected to an energized parent in order to receive power. In fact, it should not be downstream from any other distribution object.

c. **Source** *(Hidden)*

In conjunction with the *Distribution* attribute *(which is inherited from the distribution object)*, the *Source* attribute is an integer flag that specifically identifies this as a source type of a distribution object. This flag is provided to support application development and dynamic type discovery.

The Breaker Object

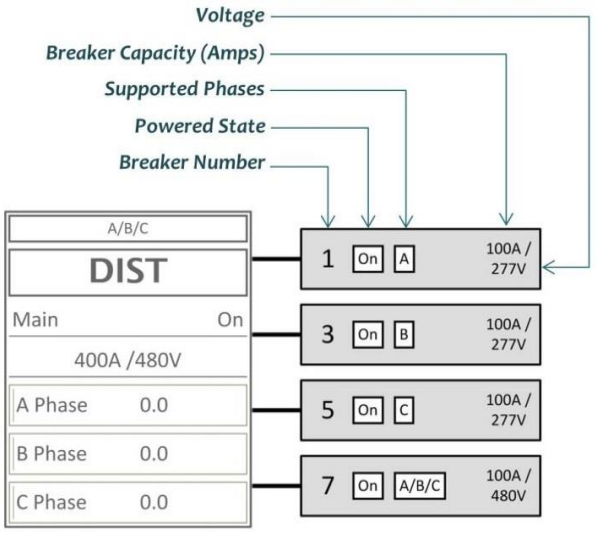

Figure 4. The Breaker Object

Breaker objects are used to represent circuit breakers that are installed between other distribution objects. These objects impose maximum limitations on the amount of power that can be distributed downstream to either, a) protect underlying components, or b) limit the total power that can be consumed by a specific electrical component or branch. The behavior of these objects is essentially identical to that of distribution objects, except for a few parameters.

a. **Name** *(Hidden)*

Like all distribution objects, the breaker object has a *Name* field. However, to minimize the visible footprint of this object in the model, the name is typically not shown. In most cases, the *Breaker Number* attribute is displayed instead.

b. **Breaker Number**

This is an integer value that identifies the breaker number within the panel board. While the application does not dictate what value is used for this field, breakers that consume multiple poles or positions will typically use the lowest number.

The Load Object

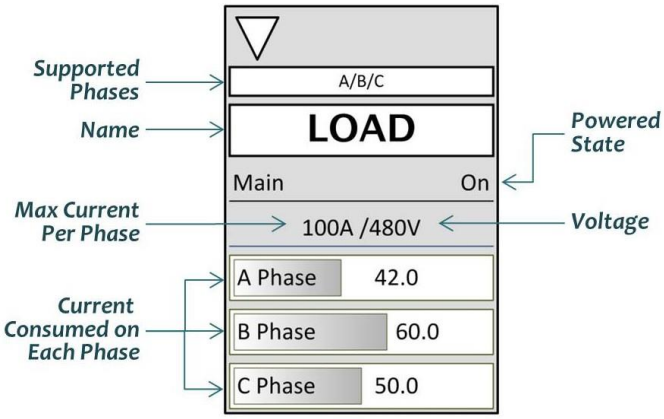

Figure 5. The Load Object

The load object is the terminal element in an electrical distribution model. No objects should be installed downstream from a load.

Power that is consumed by a load object is entered into the *[A/B/C] Phase Local* field of the object. This power is then accumulated as electrical load in all of the distribution objects that are upstream from this entity.

The load object has several distinct properties:

a. **Load** *(Hidden)*

In conjunction with the *Distribution* attribute which is also present, the *Load* attribute is an integer field that identifies this object as a *load* type of a distribution object. The attribute is used for application programming and for dynamic type discovery.

b. **Priority** *(Hidden)*

The *Priority* flag is a unique characteristic of the load object and indicates the load's relative importance as part of the electrical system. Although the entire distribution system that feeds a critical load is equally critical, to minimize distractions, only the terminal load is marked with a *Priority* flag. The following options are available.

i. *Normal* – this is a load that can lose power without jeopardizing the overall operability of the system. Examples might include lights and receptacles.
ii. *Essential* – an essential load is one that will prevent the total system from being functional if it loses power.
iii. *Critical* – a critical load is one that will result in a threat to either life safety, machine protection or will disrupt a legally required standby system if it loses power.

While not shown in the figures, when an essential or critical load loses power, a flag appears next to the object indicating that it is offline. The flag for an essential system is a yellow disc, while the flag for a critical system is a red disc.

The Load-Mini Object

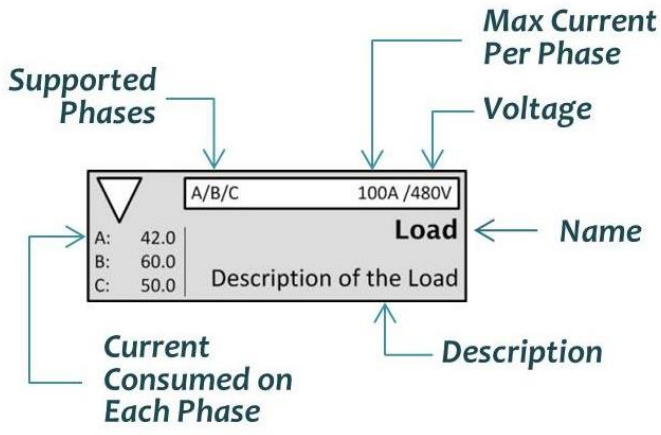

Figure 6. The Load-Mini Object

The load-mini object is a special variation of the load object that is designed to substantially reduce the footprint of the model and still provide adequate data. While this object has all of the underlying data and functionality of a load object, its size is significantly reduced.

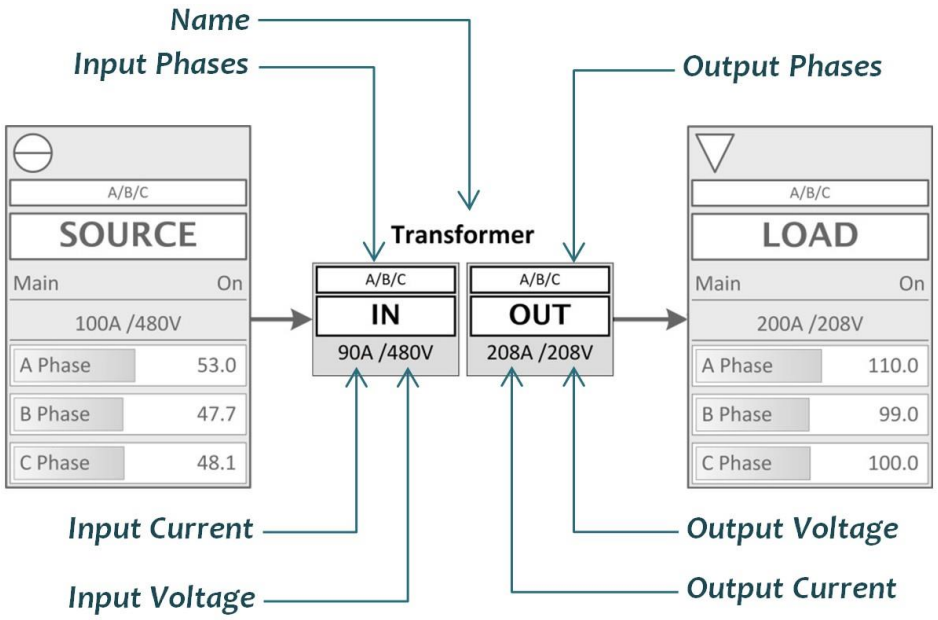

Figure 7. The Transformer Object

The Transformer Object

The transformer object is unique variant of the distribution family of objects. A transformer is used to change the voltage from its input side to its output side. The transformer imposes a conversion cost on the transformation process using an efficiency variable. This means that if the transformer has an efficiency value of 0.90 (90% efficient), then 10% of the total wattage will be lost *(and converted to heat)* as it passes from the input side to the output side.

Because the transformation process maintains the loads along each phase, the formulas used to perform the conversion are relatively simple. For each phase, the current being drawn on the output side is multiplied by the ratio of the input voltage and the output voltage to transform the power. The transformed value is then divided by the efficiency value to incur the cost of transformation. This value is then transferred to the appropriate phase on the input side of the transformer.

Take the transformation that is shown in *Figure 7* as an example. For the *A Phase:*

1) Identify the current being drawn:

   A Phase = 110 Amps

2) Multiply that value by the ratio of the voltages that are being transformed:

   *110 Amps * (208V / 480V) = 47.67 Amps*

3) Divide the transformed amperage by the efficiency.

   47.67 / 0.90 = 52.96 Amps

This value, which has been rounded in the figure above, is the load that will be drawn on the *A Phase* of the source object.

The following attributes are specific to the transformer object.

a. **Transformer** *(Hidden)*
   This is an integer field that identifies this object as a transformer type of a distribution object. It is used for application development and dynamic type discovery.

b. **Incoming Phases / Outgoing Phases**
   Because the transformer object may have different incoming and outgoing phases, the *Phases* attribute has been replaced with the *Incoming Phases* and *Outgoing Phases* attributes.

c. **Incoming Capacity**
   Only the incoming capacity of the transformer object is specified. The *Outgoing Capacity* is a

hidden attribute and is computed using the incoming capacity, voltages and efficiency.

d. **Efficiency** *(Hidden)*

This is the efficiency of the power transformation that occurs within the transformer object. This value is specified by the user and must be greater than 0 and less than or equal to 1.0.

The Switch Object

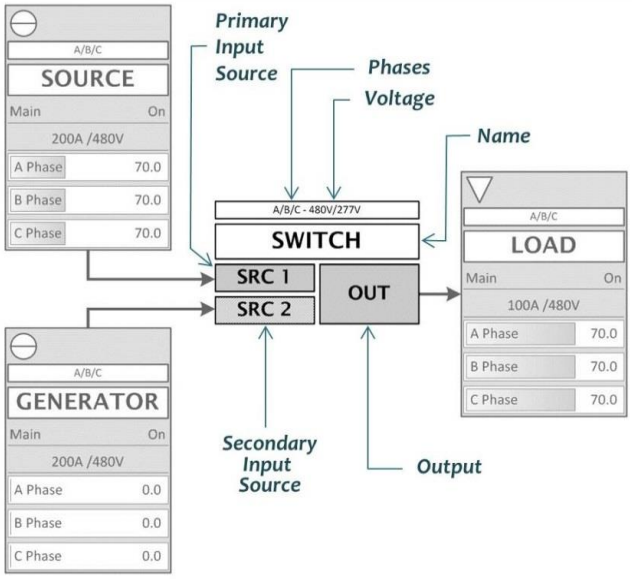

Figure 8. The Switch Object

The switch object receives power from two sources and then delivers the selected source downstream to the subordinate distribution objects. Unlike other distribution objects where the inbound connector can attach anywhere, for the switch object inbound connections MUST attach at the connector points defined on the *SRC 1* and *SRC 2* boxes. Using these connection points allows the underlying application to determine which parent is connected to which poll of the switch. For clarity, a connection point is also included on the *Out* box, but using that connection point is optional.

In examining *Figure 8* the user will note some peculiarities that are unique to the switch object. First, the selected source *(SRC 1)* has a solid fill, while the unselected source *(SRC 2)* is hatched. If the user switches the device to *SRC 2*, then that box will become solid and *SRC 1* will become hatched. Additionally, the user may select neither source, in which case both *SRC 1* and *SRC 2* will be filled with a hatch pattern and no power will be delivered downstream.

Unlike other distribution objects, the switch object has no notion of capacity. This object's capacity is essentially unlimited and it can delivery any amount of power from either of its sources to its downstream loads. If the user desires a capacity limitation for a switch, then breaker objects should be installed between the switch object and its sources.

The switch object has the following attributes.

a. **Switch** *(Hidden)*

This is an integer field that identifies this object as a switch type of a distribution object and is used for application development.

b. **Capacity**

The capacity field on the switch object is hidden. The field is set to an initial value of 1,000,000 amps. This value may be increased if necessary.

c. **Selected Source**

This field is used to identify the currently selected source. The following values are used:

0) No source is selected,

1) SRC 1 is selected,

2) SRC 2 is selected.

The Automatic Transfer Switch Object

This type of switch is visual identical to the *switch* object, however, because it is an automatic switch it is governed by the following rules.

1) SRC 1 is the primary source, if SRC 1 is energized and the automatic transfer switch is powered on, then power will be delivered to the downstream loads from SRC 1.

2) If SRC 1 is unavailable, the automatic transfer switch will immediately transfer the load to SRC 2 if it is available and the object is powered on.

3) If the automatic transfer switch is using SRC 2, when the primary source becomes available, the switch will automatically return to using SRC 1.

4) If neither the primary nor the secondary source is available, the *automatic transfer switch* will remain switched to SRC 1.

In addition to all fields defined on the switch object, the automatic transfer switch also has an ***Automatic*** attribute to distinguish it from its standard counterpart.

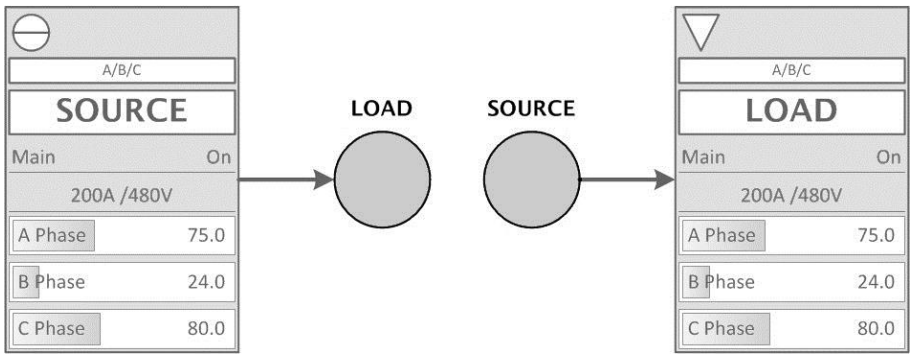

Figure 9. Proxy Objects

The Proxy Object

The final object in this collection is the proxy object. This object is designed to maintain connections and relationships across multiple pages. To be clear, a proxy object represents another distribution object that exists somewhere else in the current model. The proxy contains live references to all of the data that is stored in the actual object and can be used to distribute or accumulate load across the model.

Figure 9 contains two proxy objects. The one labeled *Load* is a proxy for the load object on the right, while the one labeled *Source* is a proxy for the source object on the left. As you can see from the diagram, the power being drawn by the load object is being drawn from the source object even though they are not physically connected.

When a proxy object is added to the page, the user will be prompted to identify the object that it is *Linked To*. This value is entered using Visio's page notation. For instance, if we are creating a proxy for an object with ID=12 on the page named Page-1, the *Linked To* attribute would contain: *Pages[Page-1]!Sheet.12*.

Note: If the object is on the same page, then the Pages part of the reference need not be included.

If a valid reference is entered, then the name displayed above the *proxy* object will be the same as the name of the object that it is referencing.

IV. CONSTRUCTING ELECTRICAL INFRASTRUCTURE MODELS

While complete guidance for using the Visio application to create diagrams is beyond the scope of this document, there are a few details that are important. The objects that are described in this document are designed to automatically interact and create relationships with one another when they are used in a document where the proper application code is installed. This section discusses how that process works, as well as other considerations that are important in developing an electrical infrastructure model.

Templates and Stencils

All of the distribution objects described earlier are included in the *electrical* stencil which is part of the infrastructure modeling package. In Visio, a stencil contains a collection of related shapes that can be used within a model. Stencils can be loaded and unloaded at will, however, once a shape from a stencil is used, it will remain part of the document until it is deleted.

Beyond mere shapes, though, this system relies on a small amount of Visual Basic application code that controls how the distribution objects establish their relationships when they are connected to one another. This VBA code is included as an appendix to this document, but is also embedded in the *electrical* template that is distributed with the infrastructure modeling package. For best results, the user should start by creating a new document using the electrical template and then save it to a new document. When the new document is opened, the relationship building code will automatically start.

Connections and Relationships

For all documents that are created using the electrical template, there are a set of functions included in the document's Visual Basic module. Whenever a connection is created or deleted between two distribution objects, these functions call the *ProcessDistributionConnections* function to update the relationships between the objects. The

direction *(upstream to downstream)* of these relationships is created based on the orientation of the connection.

For each connecting line, there exists a point that is the beginning and a point that is the end. In this modeling system, power flows from the beginning to the end. For instance, if the beginning side of the connector is linked to a source and the end side is linked to a load, then power will flow properly. Conversely, if the beginning side is attached to the load and the end is attached to the source, then nothing will happen. Because of this, the user must be careful to ensure that connections are oriented properly.

The best way to ensure proper orientation is to use connecting lines that have a directional arrow on one end. Not only does this make the resultant model easier to understand, it makes it simple to see when a connecting line is installed backward.

Sources and Loads as Terminal Objects

The source object and the load object should always be treated as terminal objects. This means that they should always be the final object on any chain of distribution objects. Although the Visio application will allow a user to have one distribution object feed into a source object, the resulting relationships is nonsensical and only serves to confuse the reader. Likewise, a load object should be the last leaf at the far end of a model. Additional objects should not extend from a load object, even though that object will tolerate the connection and will distribute power downstream.

One Source at a Time

With the exception of switches and automatic transfer switches, only one upstream source should feed an object at any time. The reason for this is because when multiple sources are feeding an object, the model cannot determine which source the power should be drawn from. As a result, even though Visio allows multiple sources to be attached to an object, the underlying Visual Basic application will ignore all but one of the connections to maintain a sensible relationship. Ensuring that only one source is connected at any time will make the models easier to read and understand.

Application Faults

Occasionally, an interaction between the user and the application will cause an error in the underlying Visual Basic application. If such an error occurs while the application is processing a connection between two distribution objects, it may cause those functions to be disabled for the remainder of the session. After this, relationships will no longer be automatically created when two distribution objects are connected.

If this happens, save the document, close it and then re-open it. This restores the underlying application to a functional state and allows connection processing to continue.

Creating Relationships Manually

Included in the Visual Basic application code is a macro named *MapDistributionNodes*. If the relationships between nodes are not being automatically created, the user can run this macro to regenerate all of the relationships between connected *distribution* objects. Note that this macro will examine every shape in the document to confirm that the relationships are correctly defined. This can be a *very* time-consuming process and should be avoided in favor of using the event based relationship builder that automatically evaluates connections as they occur.

V. CONCLUSIONS

This document has described a technique for constructing a model that describes an industrial electrical system. This infrastructure modeling system has been developed using commercial software that is readily available and is common in most workplaces. While the modeling system does rely on some locally developed software, the amount of code that was created is very small, well-documented and should be approachable by any developer who is versed in the Microsoft Office suite of applications.

While the development of models like this is a meaningful contribution to good configuration management, it is only the first step. Management of a large electrical infrastructure demands that modeling be used in conjunction with good record keeping and periodic audits that ensure the records are accurate and up-to-date. Accordingly, infrastructure models, like one-line diagrams and panel schedules, must be audited and updated regularly. Likewise, the models generated from the modeling approach described here should be coupled with diligent record keeping and regular auditing to be most effective.

While the goal of this project was to rapidly deploy a modeling system to support an electrical infrastructure, this product is only a starting point. The future developments, described below, should leverage these techniques to improve our understanding of the systems we work with and expand the utility of these models.

Future Directions

The objects and the modeling techniques described in this document are only the first steps in developing a simple and inexpensive technique for managing industrial infrastructure. While this document focuses specifically on electrical systems, the foundation is laid for expanding this technique to include modeling HVAC systems, process cooling water and, potentially, cryogenic systems.

The future directions that this research will pursue include techniques for automating the conversion of electrical schedules and diagrams stored in other formats into dynamic models like those described here. As with this system, the development objectives will continue to be, a) using commercially available software, b) minimizing locally developed code, c) creating an environment that is easy for a *non-expert* to use, and d) keeping the price point as low as possible. Further developments will include the design and implementation of an alarm system that alerts users when the critical or essential systems will be impacted by either maintenance activities, scheduled outages or failures.

Beyond these rudimentary applications, the development of an extrapolation model that allows users to estimate the amount of power consumed by elements within the electrical system is also a future possibility. This approach would allow the user to enter system readings or estimates where they are known, after which the application would distribute power throughout the system to make the estimated loads conform to the entered readings. To improve performance, this application will likely be developed through the use of an application *add-in* or a dynamically linked library that uses a higher performing language.

Source Code

Included as appendices to this document are a variety of Visual Basic Application sources. These include the following:

a. **Appendix 1: Document Event Handlers**
   This is the VBA code that must be included in the *ThisDocument* module of each document to cause new connections to automatically update the relationships between distribution objects.

b. **Appendix 2: Distribution Code Module**
   These are functions and macros that specifically apply to distribution objects in the Visio environment.

c. **Appendix 3: Utility Code Module**
   These are lightweight utilities that are provided for working with generic Visio shapes and objects.

d. **Factories Module**
   While the original collection of distribution objects described in this document were constructed by hand, it quickly became more practical to develop a set of Visual Basic functions that automatically generate the objects. These functions allow new features to be added within the source code and immediately instantiated into new objects for distribution. Because this module is very large and complex, it is included as part of the online software distribution rather than as an appendix.

Obtaining this Modeling Package

All of the source code, stencils and templates described within this document are available online via anonymous ftp from:

ftp://ftp.jlab.org/pub/modeling

VI. REFERENCES

1. Boulding, K. (1966). *The Impact of Social Sciences*. New Brunswick, NJ: Rutgers University Press.
2. Donley, M. (2011, October 17). *History of Sketchup*. Retrieved March 30, 2018, from masterSketchup.com: https://mastersketchup.com/history-of-sketchup/
3. Johnson, T. (2013, September 14). *Timeline*. Retrieved March 30, 2018, from Visio: http://www.visiocorp.info/timeline.aspx
4. Kempf, K. (1961). *Electronic Computers within the Ordnance Corps*. Aberdeen Proving Ground, MD: United States Army.
5. Warfield, J. (1993). Structural Thinking: Producing Effective Organizational Change. *15th Annual Meeting of the Association for Integrative Studies*. Detroit.
6. Weisberg, D. E. (2011, November 18). *The Engineering Design Revolution*. Retrieved March 30, 2018, from http://cadhistory.net

Appendix 1: Document Event Handlers

```
Option Explicit

Private WithEvents m_page As Visio.Page

Private Sub Document_DocumentOpened(ByVal doc As IVDocument)
    Set m_page = Visio.ActivePage
End Sub

Private Sub Document_PageChanged(ByVal Page As IVPage)
    Set m_page = Visio.ActivePage
End Sub

Private Sub m_page_ConnectionsDeleted(ByVal Connects As IVConnects)
    ProcessDistributionConnections Connects
End Sub

Private Sub m_page_ConnectionsAdded(ByVal Connects As IVConnects)
    ProcessDistributionConnections Connects
End Sub

Private Sub m_page_BeforeShapeDelete(ByVal shp As IVShape)
    If shp.LineStyle = "Connector" Then
        shp.Disconnect visConnectorBothEnds, 0.1, 0.1, 0
    End If
End Sub
```

Appendix 2: Distribution.BAS - Distribution Code Module

```
Attribute VB_Name = "Distribution"

' ================================================================================
' Distribution Module
' Walt Akers
' Thomas Jefferson National Accelerator Facility
'
' This module contains the functions necessary to manage and create
' relationships between various distribution objects.
' ================================================================================

' ================================================================================
' MapDistributionNodes:
'   Maps Loads and Sources.
' ================================================================================
Sub MapDistributionNodes()
    MapDistributionLoads
    MapDistributionSources
End Sub

' ================================================================================
' mapDistributionNode:
'   This function calls the mapDistributionLoad and mapDistributionSource for
'   a single shape.
' ================================================================================
Function mapDistributionNode(vsoShape As Visio.Shape)
    MapDistributionLoad vsoShape
    MapDistributionSource vsoShape
End Function


' ================================================================================
' MapDistributionLoads:
'   This function walks through all of the shapes and creates the connections
'   necessary to distribute loads from the bottom to the top.
' ================================================================================
Function MapDistributionLoads()
```

```
    Dim vsoShapes As Visio.Shapes
    Dim vsoShape As Visio.Shape

    Set vsoShapes = Visio.ActivePage.Shapes
    MapDistributionShapesLoads vsoShapes
End Function


' ================================================================================
' MapDistributionShapesLoads:
'   This is an overloaded version of the MapDistributionLoads function that
'   passes a vsoShapes object as a parameter.  This function will examine
'   each shape.  Shapes to be processed are one of two types.
'
'   Distribution Object: (Has a Property cell named Distribution)
'       This object will be processed as a Distribution Object.
'
'   DistributionParent: (Has a Property cell named DistributionParent)
'       The shapes contained within this shape will be passed (recursively) to
'       this function to be evaluated and processed.
' ================================================================================
Function MapDistributionShapesLoads(vsoShapes As Shapes)
    Dim vsoShape As Visio.Shape

    For Each vsoShape In vsoShapes
        If (Not vsoShape.LineStyle = "Connector") Then
            If (vsoShape.CellExistsU("Prop.DistributionParent", False) And _
                vsoShape.Shapes.Count > 0) Then _
                    MapDistributionShapesLoads vsoShape.Shapes
            If (vsoShape.CellExistsU("Prop.Distribution", False)) Then _
                MapDistributionLoad vsoShape
        End If
    Next vsoShape

End Function


' ================================================================================
' MapDistributionLoad:
'   This funcction will examine all shapes that are connected to this shape
'   as Outgoing Nodes.  If the Outgoing Node is also a distribution node, then
'   the A/B/C values on that Distribution object will be added to the
'   A_LOAD/B_LOAD/C_LOAD values on this Distribution object.
'
'   This effectively propogates the consumption of amperage from the bottom
'   of the model (the Loads) to the top of the model (the Sources).
'
'   Because all objects have an embedded Distribution property, it is assumed
'   that they have all of the required Distribution fields.  The Properties
'   that are used by this function include
'
'   Distribution    Indicates the object is a distribution node
'   A               Amperage consumed by the node on the A phase
'   B               Amperage consumed by the node on the B phase
'   C               Amperage consumed by the node on the C phase
'   A_Load          Amperage accumulated on the nodes A Phase from subordinates
'   B_Load          Amperage accumulated on the nodes B Phase from subordinates
'   C_Load          Amperage accumulated on the nodes C Phase from subordinates
'
'   Note that if that a connected distribution object is a switch, then it will
'   be connected to this object based on the identifier of the connection
'   point.
'
'   For instance, if this distribution object is connected to port 1 on a
'   Switch, the load will be collected from Switch!Prop.A1, Switch.Prop.B1,
'   and Switch.Prop.C1 accordingly.
'
'    If this object has a connection to a Switch that is not attached to a
'   connection point, then it will neither accumulate load nor distribute
```

```
'   power to that node.
' ============================================================================
Function MapDistributionLoad(vsoShape As Visio.Shape)
    Dim vsoChild As Visio.Shape
    Dim childCP As String
    Dim conn() As ConnectedShapes
    Dim connCt As Integer
    Dim cnt  As Integer
    Dim aVal As String
    Dim bVal As String
    Dim cVal As String
    Dim aSrc As String
    Dim bSrc As String
    Dim cSrc As String

    cnt = 0
    aVal = "0"
    bVal = "0"
    cVal = "0"

    If (vsoShape.CellExistsU("Prop.Distribution", False)) Then

        conn = ReadConnectivity(vsoShape)
        Err.Clear
        On Error Resume Next
        connCt = UBound(conn)
        If (Err.Number <> 0) Then connCt = 0

        For i = 1 To connCt
            If (conn(i).ConnectCount = 2 And Not conn(i).Incoming) Then
                Set vsoChild = conn(i).ToShape
                ' ============================================================
                ' Only process this operation if the underlying child is a
                ' distribution object.
                ' ============================================================
                If (vsoChild.CellExistsU("Prop.Distribution", False)) Then
                    ' ========================================================
                    ' Determine the proper name for each load Phase on the
                    ' child.  For normal distributions this will be Prop.A,
                    ' Prop.B and Prop.C.  For Switch Distributions, this will
                    ' be Prop.An, Prop.Bn and Prop.Cn where n is the number of
                    ' the connection point.
                    ' ========================================================
                    If (vsoChild.CellExistsU("Prop.Switch", False)) Then
                        childCP = conn(i).ToConnectionPoint
                        aSrc = "Prop.A" & childCP
                        bSrc = "Prop.B" & childCP
                        cSrc = "Prop.C" & childCP
                    Else
                        aSrc = "Prop.A"
                        bSrc = "Prop.B"
                        cSrc = "Prop.C"
                    End If

                    ' ========================================================
                    ' Before adding the relationship, make sure that each of
                    ' the properties exists on the child.
                    ' ========================================================
                    If (vsoChild.CellExistsU(aSrc, False) And _
                         vsoChild.CellExistsU(bSrc, False) And _
                         vsoChild.CellExistsU(cSrc, False)) Then
                        If (cnt = 0) Then
                            aVal = vsoChild.NameID & "!" & aSrc
                            bVal = vsoChild.NameID & "!" & bSrc
                            cVal = vsoChild.NameID & "!" & cSrc
                            cnt = cnt + 1
                        Else
                            aVal = aVal & "+" & vsoChild.NameID & "!" & aSrc
                            bVal = bVal & "+" & vsoChild.NameID & "!" & bSrc
```

```
                            cVal = cVal & "+" & vsoChild.NameID & "!" & cSrc
                        End If
                        cnt = cnt + 1
                    End If
                End If
            End If
        Next

        SetPropertyValue vsoShape, "A_LOAD", aVal
        SetPropertyValue vsoShape, "B_LOAD", bVal
        SetPropertyValue vsoShape, "C_LOAD", cVal
    End If
End Function


' ============================================================================
' MapDistributionSources:
'   This function walks through all of the shapes and creates the connections
'   necessary to provision power downward from the top to the bottom.
' ============================================================================
Function MapDistributionSources()
    Dim vsoShapes As Visio.Shapes
    Dim vsoShape As Visio.Shape

    Set vsoShapes = Visio.ActivePage.Shapes
    MapDistributionShapesSources vsoShapes
End Function


' ============================================================================
' MapDistributionShapesSources:
'   This is an overloaded version of the MapDistributionSources function that
'   passes a vsoShapes object as a parameter.  This function will examine
'   each shape.  Shapes to be processed are one of two types.
'
'   Distribution Object: (Has a Property cell named Distribution)
'       This object will be processed as a Distribution Object.
'
'   DistributionParent: (Has a Property cell named DistributionParent)
'       The shapes contained within this shape will be passed (recursively) to
'       this function to be evaluated and processed.
' ============================================================================
Function MapDistributionShapesSources(vsoShapes As Shapes)
    Dim vsoShape As Visio.Shape

    For Each vsoShape In vsoShapes
        If (Not vsoShape.LineStyle = "Connector") Then
            If (vsoShape.CellExistsU("Prop.DistributionParent", False) And _
                vsoShape.Shapes.Count > 0) Then _
                    MapDistributionShapesSources vsoShape.Shapes
            If (vsoShape.CellExistsU("Prop.Distribution", False)) Then _
                MapDistributionSource vsoShape
        End If
    Next vsoShape

End Function


' ============================================================================
' MapDistributionSource:
'   This function creates a relationship between a distribution node, with the
'   Distribution nodes that exist above it to determine if the node is:
'
'   a) Powered and
'   b) Phase Mismatch
'
'   A Distribution node should have one and only one Distribution parent.
'   Still, this function will tolerate multiple parent distributions, by
'   OR-ing them to produce a value.
```

```
'
'   Because all objects have an embedded Distribution property, it is assumed
'   that they have all of the required Distribution fields.  The Properties
'   that are used by this function include
'
'   Distribution    Indicates the object is a distribution node
'   Powered         Boolean flag indicating the object is powered from above
'   On              Boolean flag indicating the object is switched on
'   Tripped         Boolean flag indicating the object is tripped
'   Phases          A bit field indicating the phases supported by the object
'   PhaseMismatch       Boolean indicates a phase mismatch between parent and child
'
'   Note that if this object is a Distribution and a Source, then it will
'   automatically be marked as Powered.
'
'   Also Note:  Unless a distribution object is a switch, it should not be fed
'   from more than one source of power. If this object is receiving power from
'   more than one source (and it is not a Switch), then only the last power
'   source will be considered.
' ============================================================================
Function MapDistributionSource(vsoShape As Visio.Shape)
    Dim vsoParent As Visio.Shape
    Dim childCP As String
    Dim conn() As ConnectedShapes
    Dim connCt As Integer

    Dim parentProp  As String
    Dim poweredProp As String
    Dim phaseMismatchProp As String
    Dim phaseProp As String
    Dim voltageProp As String

    Dim parentPhaseProp As String
    Dim parentVoltageProp As String

    Dim idx As Integer

    ' ========================================================================
    ' Confirm that this object is a Distribution before processing it.
    ' ========================================================================
    If (vsoShape.CellExistsU("Prop.Distribution", False)) Then
        On Error Resume Next
        ' ====================================================================
        ' If this is a Switch, the first essential step is to walk through
        ' all of the connections and clear them before this process begins.
        ' ====================================================================
        If (vsoShape.CellExistsU("Prop.Switch", False)) Then

            Dim iRow As Integer
            iRow = 0

            While vsoShape.RowExists(visSectionConnectionPts, iRow, False)
                childCP = vsoShape.CellsSRC(visSectionConnectionPts, iRow, 0).RowName
                If childCP <> "" Then
                    parentProp = "Prop.Parent" & childCP
                    poweredProp = "Prop.Powered" & childCP
                    phaseMismatchProp = "Prop.PhaseMismatch" & childCP

                    If (vsoShape.CellExistsU(parentProp, False)) Then _
                        vsoShape.CellsU(parentProp).FormulaU = """"""
                    If (vsoShape.CellExistsU(poweredProp, False)) Then _
                        vsoShape.CellsU(poweredProp).FormulaU = "False"
                    If (vsoShape.CellExistsU(phaseMismatchProp, False)) Then _
                        vsoShape.CellsU(phaseMismatchProp).FormulaU = "False"
                End If
                iRow = iRow + 1
            Wend
        ' ====================================================================
        ' If this is not a switch, then the properties that are being
```

```
        ' updated are FIXED and should be set here.  Otherwise, they will be
        ' updated on each iteration to reflect the connectionID.
        ' ================================================================
        Else
            parentProp = "Prop.Parent"
            poweredProp = "Prop.Powered"
            phaseMismatchProp = "Prop.PhaseMismatch"

            If (vsoShape.CellExistsU(parentProp, False)) Then vsoShape.CellsU(parentProp).FormulaU = _
                """"""""
            If (vsoShape.CellExistsU(poweredProp, False)) Then vsoShape.CellsU(poweredProp).FormulaU = _
                "False"
            If (vsoShape.CellExistsU(phaseMismatchProp, False)) Then _
                vsoShape.CellsU(phaseMismatchProp).FormulaU = "False"
        End If

        ' ================================================================
        ' Determine the names of the properties that will be addressed in this
        ' object.  If it is a transformer object, then the Voltage
        ' property will be prepended with an "i" to differentiate it from
        ' teh output voltages.
        ' ================================================================
        phaseProp = "Prop.Phases"
        If vsoShape.CellExistsU("Prop.Transformer", False) Then
            voltageProp = "Prop.iVoltage"
        Else
            voltageProp = "Prop.Voltage"
        End If

        ' ================================================================
        ' If this Distribution object is a Source, then it should be "Powered"
        ' by default. Further, sources are assumed to have no upstream
        ' connections.
        ' ================================================================
        If (vsoShape.CellExistsU("Prop.Source", False)) Then
            vsoShape.CellsU(poweredProp).FormulaU = "True"
            valCnt = 1
        Else
            ' ============================================================
            ' Get a list of connections that are feeding into this shape
            ' ============================================================
            conn = ReadConnectivity(vsoShape)
            Err.Clear
            connCt = UBound(conn)
            If (Err.Number <> 0) Then connCt = 0

            ' ============================================================
            ' Iterate through the connections.  Only process them if the
            ' connection has two connected shapes, it is an incoming connection
            ' to this object and the shape that it is connecting from is a
            ' Distribution object.
            ' ============================================================
            For i = 1 To connCt
                If (conn(i).ConnectCount = 2 And _
                    conn(i).Incoming And _
                    conn(i).FromShape.CellExistsU("Prop.Distribution", False)) Then
                    Set vsoParent = conn(i).FromShape
                    childCP = conn(i).ToConnectionPoint

                    ' ========================================================
                    ' If this device is a Switch, then the properties must have
                    ' the connection point identifier appended to them.
                    ' ========================================================
                    If (vsoShape.CellExistsU("Prop.Switch", False)) Then
                        parentProp = "Prop.Parent" & childCP
                        poweredProp = "Prop.Powered" & childCP
                        phaseMismatchProp = "Prop.PhaseMismatch" & childCP
                    End If
```

```
                    ' ========================================================
                    ' Set the name, phase and voltage property names
                    ' ========================================================
                    parentPhaseProp = vsoParent.NameID & "!Prop.Phases"
                    parentVoltageProp = vsoParent.NameID & "!Prop.Voltage"

                    ' ========================================================
                    ' This conditional prevents the object from accidentally
                    ' overwriting the master Powered and PhaseMismatch properties
                    ' on a Switch object.
                    ' ========================================================
                    If (Not vsoShape.CellExistsU("Prop.Switch", False) Or _
                        childCP <> "") Then

                        ' ========================================================
                        ' Set the Parent property.
                        ' ========================================================
                        If (vsoShape.CellExistsU(parentProp, False)) Then
                            vsoShape.CellsU(parentProp).FormulaU = vsoParent.NameID & "!Prop.Name"
                        End If


                        ' ========================================================
                        ' Set the Powered property.
                        ' ========================================================
                        If (vsoShape.CellExistsU(poweredProp, False)) Then
                            vsoShape.CellsU(poweredProp).FormulaU = vsoParent.NameID & "!Prop.Powering"
                        End If

                        ' ========================================================
                        ' Set the PhaseMismatch calculation which also considers phase
                        ' mismatches between the objects voltages.
                        ' ========================================================
                        If (vsoShape.CellExistsU(phaseMismatchProp, False)) Then
                            vsoShape.CellsU(phaseMismatchProp).FormulaU = _
                            "OR((BITAND(" & phaseProp & "," _
                            & parentPhaseProp & ")<>" _
                            & phaseProp & ")," _
                            & "NOT(STRSAME(" & voltageProp & "," _
                            & parentVoltageProp & ")))"
                        End If
                    End If
                End If
            Next
        End If
    End If

End Function


' ============================================================================
' This function is called whenever a connection is changed (added or removed)
' ============================================================================
Function ProcessDistributionConnections(ByVal Connects As IVConnects)
    Dim cnct As Visio.Connect
    Dim subCnt As Visio.Connect
    Dim vsoShapes As Visio.Shapes
    Dim vsoShape As Visio.Shape
    Dim processed As String

    On Error Resume Next

    For Each cnct In Connects
        '========================================================================
        ' Begin with the FromSheet that is listed in the connection. If it
        ' is valid, then process it.
        ' ========================================================================
        If (Not (cnct.FromSheet Is Nothing)) Then
            ' ========================================================
```

```
        ' If the Shape referred to by FromSheet has a Distribution property
        ' then it should be processed.
        ' ============================================================
        If (cnct.FromSheet.CellExistsU("Prop.Distribution", False)) Then
            mapDistributionNode cnct.FromSheet

        ' ============================================================
        ' Conversely, if the FromSheet has no Distribution property, but it
        ' does have its own set of connections, then they should be processed.
        ' ============================================================
        ElseIf (Not (cnct.FromSheet.Connects Is Nothing)) Then
            For Each subCnt In cnct.FromSheet.Connects
                ' ========================================================
                ' If the FromSheet in this subConnection is valid and has a
                ' Distribution property - process it.
                ' ========================================================
                If (Not (subCnt.FromSheet Is Nothing) And _
                    subCnt.FromSheet.CellExistsU("Prop.Distribution", False)) Then
                    mapDistributionNode subCnt.FromSheet
                End If

                ' ========================================================
                ' If the ToSheet in this subConnection is valid and has a
                ' Distribution property - process it.
                ' ========================================================
                If (Not (subCnt.ToSheet Is Nothing) And _
                    subCnt.ToSheet.CellExistsU("Prop.Distribution", False)) Then
                    mapDistributionNode subCnt.ToSheet
                End If
            Next
        End If
    End If

    '====================================================================
    ' Next examine the ToSheet - If it is valid, then process it.
    ' ====================================================================
    If (Not (cnct.ToSheet Is Nothing)) Then

        ' ============================================================
        ' If the Shape referred to by ToSheet has a Distribution property
        ' then it should be processed.
        ' ============================================================
        If (cnct.ToSheet.CellExistsU("Prop.Distribution", False)) Then
            mapDistributionNode cnct.ToSheet

        ' ============================================================
        ' Conversely, if the ToSheet has no Distribution property, but it
        ' does have its own set of connections, then they should be processed.
        ' ============================================================
        ElseIf (Not (cnct.ToSheet.Connects Is Nothing)) Then
            For Each subCnt In cnct.ToSheet.Connects

                ' ========================================================
                ' If the FromSheet in this subConnection is valid and has a
                ' Distribution property - process it.
                ' ========================================================
                If (Not (subCnt.FromSheet Is Nothing) And _
                    subCnt.FromSheet.CellExistsU("Prop.Distribution", False)) Then
                    mapDistributionNode subCnt.FromSheet
                End If

                ' ========================================================
                ' If the ToSheet in this subConnection is valid and has a
                ' Distribution property - process it.
                ' ========================================================
                If (Not (subCnt.ToSheet Is Nothing) And _
                    subCnt.ToSheet.CellExistsU("Prop.Distribution", False)) Then
                    mapDistributionNode subCnt.ToSheet
                End If
```

```
                Next
            End If
        End If
    Next
End Function
```

Appendix 3: Utility.BAS - Utility Code Module

```
Attribute VB_Name = "Utility"

' ============================================================================
' Utility Module
' Walt Akers
' Thomas Jefferson National Accelerator Facility
'
' This module contains a collection of utility functions that support Visio
' applications and shapes. These functions include:
'
' Function GotoPage(PageName As String)
'       Goes to a named page and creates it if necessary.
'
' Function SetShapeDimensions(...)
'       A utility function that sets the dimension settings of a shape.
'
' Function SetShapeProtections(...)
'       A utility function that sets the protection settings of a shape.
'
' Function SetShapeMisc(...)
'       A utility function that sets the miscellaneous settings of a shape.
'
' Function SetShapeLineFormat(...)
'       A utility function that sets the shape's line format.
'
' Function SetShapeFillFormat(...)
'       A utility function that sets the shape's fill format.
'
' Function SetShapeMargins(...)
'       A utility function that sets the shape's internal margins.
'
' Function SetShapeCharFormat(...)
'       A utility function that sets the shape's internal margins.
'
' Function SetPropertyParams(...)
'       Creates a property in the Visio shape (if necesary) and then
'       sets the parameters of the property.
'
' Function DeleteProperty(...)
'       Deletes a property from the shape.
'
' Function SetPropertyValue(...)
'       Creates a property in the Visio shape (if necessary) and then
'       sets the value property.
'
' Function GetPropertyValue(...)
'       Reads a property in the Visio shape and then returns it to the caller.
'
' Function SetPropertyVisibility(...)
'       This function will hide or unhide properties.
'
' Function ConnectShapes(...)
'       Connect the specified shapes, using the connecting points if
'       they are specified.
'
' Function ReadConnectivity(...)
'       Reads the connectivity information of a shape.
' ============================================================================
```

```
' ================================================================================
' The ConnectedShapes object is used by the ReadConnectivity function to
' produce a list of all connections that are coming into (or goiong out of)
' an object.
'
' The Connector object is connecting line that runs between the two shapes.
'
' The ConnectCount variable is a count of connections on the connector - if it
' is less than 2, then this is a dangling connector.
'
' If the connection is coming into the shape, then the Incoming
' flag will be set and the FromShape will be the remote object that is
' connected to the evaluated shape.
'
' If the connection is going out of the shape (and ConnectCount > 1), then the
' Incoming flag will be false and the ToShape will be the remote object that
' is connected to the evaluated shape.
' ================================================================================
Public Type ConnectedShapes
    connector As Visio.Shape
    ConnectCount As Integer
    FromShape As Visio.Shape
    FromConnectionPoint As String
    ToShape As Visio.Shape
    ToConnectionPoint As String
    Incoming As Boolean
End Type

' ================================================================================
' GotoPage
' This is a utility function that moves to a named page and creates it if
' necessary.
' ================================================================================
Function GotoPage(PageName As String)
    On Error Resume Next
    Err.Clear

    Application.ActiveWindow.Page = Application.ActiveDocument.Pages.ItemU(PageName)

    If (Err.Number <> 0) Then
        Dim vsoPage1 As Visio.Page
        Set vsoPage1 = ActiveDocument.Pages.Add
        vsoPage1.Name = PageName
        Application.ActivePage.Name = PageName
        Application.ActivePage.NameU = PageName
    End If

End Function

' ================================================================================
' SetShapeDimensions
' This is a utility function that sets the dimension settings of a shape.
' ================================================================================
Function SetShapeDimensions(vsoShape As Visio.Shape, _
    Optional width As String = "", _
    Optional height As String = "", _
    Optional pinX As String = "", _
    Optional pinY As String = "", _
    Optional locPinX As String = "", _
    Optional locPinY As String = "")

    On Error Resume Next
    If (width <> "") Then vsoShape.CellsU("Width").FormulaForceU = width
    If (height <> "") Then vsoShape.CellsU("Height").FormulaForceU = height
    If (pinX <> "") Then vsoShape.CellsU("PinX").FormulaForceU = pinX
    If (pinY <> "") Then vsoShape.CellsU("PinY").FormulaForceU = pinY
    If (locPinX <> "") Then vsoShape.CellsU("LocPinX").FormulaForceU = locPinX
    If (locPinY <> "") Then vsoShape.CellsU("LocPinY").FormulaForceU = locPinY
```

```
    On Error GoTo 0

End Function

' ============================================================================
' SetShapeProtections
' This is a utility function that sets the protection settings of a shape.
' ============================================================================
Function SetShapeProtections(vsoShape As Visio.Shape, _
    Optional lockWidth As String = "", _
    Optional lockHeight As String = "", _
    Optional lockAspect As String = "", _
    Optional lockMoveX As String = "", _
    Optional lockMoveY As String = "", _
    Optional lockRotate As String = "", _
    Optional lockBegin As String = "", _
    Optional lockEnd As String = "", _
    Optional lockDelete As String = "", _
    Optional lockSelect As String = "", _
    Optional lockFormat As String = "", _
    Optional lockCustProp As String = "", _
    Optional lockTextEdit As String = "", _
    Optional lockVtxEdit As String = "", _
    Optional lockCrop As String = "", _
    Optional lockGroup As String = "", _
    Optional lockCalcWH As String = "", _
    Optional lockFromGroupFormat As String = "", _
    Optional lockThemeColors As String = "", _
    Optional lockThemeEffects As String = "")

    On Error Resume Next
    If (lockWidth <> "") Then vsoShape.CellsU("LockWidth").FormulaForceU = lockWidth
    If (lockHeight <> "") Then vsoShape.CellsU("LockHeight").FormulaForceU = lockHeight
    If (lockAspect <> "") Then vsoShape.CellsU("LockAspect").FormulaForceU = lockAspect
    If (lockMoveX <> "") Then vsoShape.CellsU("LockMoveX").FormulaForceU = lockMoveX
    If (lockMoveY <> "") Then vsoShape.CellsU("LockMoveY").FormulaForceU = lockMoveY
    If (lockRotate <> "") Then vsoShape.CellsU("LockRotate").FormulaForceU = lockRotate
    If (lockBegin <> "") Then vsoShape.CellsU("LockBegin").FormulaForceU = lockBegin
    If (lockEnd <> "") Then vsoShape.CellsU("LockEnd").FormulaForceU = lockEnd
    If (lockDelete <> "") Then vsoShape.CellsU("LockDelete").FormulaForceU = lockDelete
    If (lockSelect <> "") Then vsoShape.CellsU("LockSelect").FormulaForceU = lockSelect
    If (lockFormat <> "") Then vsoShape.CellsU("LockFormat").FormulaForceU = lockFormat
    If (lockCustProp <> "") Then vsoShape.CellsU("LockCustProp").FormulaForceU = lockCustProp
    If (lockTextEdit <> "") Then vsoShape.CellsU("LockTextEdit").FormulaForceU = lockTextEdit
    If (lockVtxEdit <> "") Then vsoShape.CellsU("LockVtxEdit").FormulaForceU = lockVtxEdit
    If (lockCrop <> "") Then vsoShape.CellsU("LockCrop").FormulaForceU = lockCrop
    If (lockGroup <> "") Then vsoShape.CellsU("LockGroup").FormulaForceU = lockGroup
    If (lockCalcWH <> "") Then vsoShape.CellsU("LockCalcWH").FormulaForceU = lockCalcWH
    If (lockFromGroupFormat <> "") Then vsoShape.CellsU("LockFromGroupFormat").FormulaForceU = _
        lockFromGroupFormat
    If (lockThemeColors <> "") Then vsoShape.CellsU("LockThemeColors").FormulaForceU = _
        lockThemeColors
    If (lockThemeEffects <> "") Then vsoShape.CellsU("LockThemeEffects").FormulaForceU = _
        lockThemeEffects
    On Error GoTo 0

End Function



' ============================================================================
' SetShapeMisc
' This is a utility function that sets the miscellaneous settings of a shape.
' ============================================================================
Function SetShapeMisc(vsoShape As Visio.Shape, _
    Optional noObjHandles As String = "", _
    Optional noCtlHandles As String = "", _
    Optional noAlignBox As String = "", _
    Optional langID As String = "", _
```

```
    Optional hideText As String = "", _
    Optional updateAlignBox As String = "", _
    Optional dynFeedback As String = "", _
    Optional noLiveDynamics As String = "", _
    Optional calendar As String = "", _
    Optional objType As String = "", _
    Optional isDropSource As String = "", _
    Optional comment As String = "", _
    Optional dropOnPageScale As String = "", _
    Optional localizeMerge As String = "")

    On Error Resume Next
    If (noObjHandles <> "") Then vsoShape.CellsU("NoObjHandles").FormulaForceU = noObjHandles
    If (noCtlHandles <> "") Then vsoShape.CellsU("NoCtlHandles").FormulaForceU = noCtlHandles
    If (noAlignBox <> "") Then vsoShape.CellsU("NoAlignBox").FormulaForceU = noAlignBox
    If (langID <> "") Then vsoShape.CellsU("LangID").FormulaForceU = langID
    If (hideText <> "") Then vsoShape.CellsU("HideText").FormulaForceU = hideText
    If (updateAlignBox <> "") Then vsoShape.CellsU("UpdateAlignBox").FormulaForceU = updateAlignBox
    If (dynFeedback <> "") Then vsoShape.CellsU("DynFeedback").FormulaForceU = dynFeedback
    If (noLiveDynamics <> "") Then vsoShape.CellsU("NoLiveDynamics").FormulaForceU = noLiveDynamics
    If (calendar <> "") Then vsoShape.CellsU("Calendar").FormulaForceU = calendar
    If (objType <> "") Then vsoShape.CellsU("ObjType").FormulaForceU = objType
    If (isDropSource <> "") Then vsoShape.CellsU("IsDropSource").FormulaForceU = isDropSource
    If (comment <> "") Then vsoShape.CellsU("Comment").FormulaForceU = comment
    If (dropOnPageScale <> "") Then vsoShape.CellsU("DropOnPageScale").FormulaForceU = dropOnPageScale
    If (localizeMerge <> "") Then vsoShape.CellsU("LocalizeMerge").FormulaForceU = localizeMerge
    On Error GoTo 0

End Function

' ================================================================================
' SetShapeLineFormat
' This is a utility function that sets the shape's line format.
' ================================================================================
Function SetShapeLineFormat(vsoShape As Visio.Shape, _
    Optional linePattern As String = "", _
    Optional lineWeight As String = "", _
    Optional lineColor As String = "", _
    Optional lineCap As String = "", _
    Optional beginArrow As String = "", _
    Optional endArrow As String = "", _
    Optional lineColorTrans As String = "", _
    Optional beginArrowSize As String = "", _
    Optional endArrowSize As String = "", _
    Optional rounding As String = "")

    On Error Resume Next
    If (linePattern <> "") Then vsoShape.CellsU("LinePattern").FormulaForceU = linePattern
    If (lineWeight <> "") Then vsoShape.CellsU("LineWeight").FormulaForceU = lineWeight
    If (lineColor <> "") Then vsoShape.CellsU("LineColor").FormulaForceU = lineColor
    If (lineCap <> "") Then vsoShape.CellsU("LineCap").FormulaForceU = lineCap
    If (beginArrow <> "") Then vsoShape.CellsU("BeginArrow").FormulaForceU = beginArrow
    If (endArrow <> "") Then vsoShape.CellsU("EndArrow").FormulaForceU = endArrow
    If (lineColorTrans <> "") Then vsoShape.CellsU("LineColorTrans").FormulaForceU = lineColorTrans
    If (beginArrowSize <> "") Then vsoShape.CellsU("BeginArrowSize").FormulaForceU = beginArrowSize
    If (endArrowSize <> "") Then vsoShape.CellsU("EndArrowSize").FormulaForceU = endArrowSize
    If (rounding <> "") Then vsoShape.CellsU("Rounding").FormulaForceU = rounding
    On Error GoTo 0
End Function


' ================================================================================
' SetShapeFillFormat
' This is a utility function that sets the shape's fill format.
' ================================================================================
Function SetShapeFillFormat(vsoShape As Visio.Shape, _
    Optional fillForegnd As String = "", _
    Optional fillForegndTrans As String = "", _
    Optional fillBkgnd As String = "", _
```

```
    Optional fillBkgndTrans As String = "", _
    Optional fillPattern As String = "", _
    Optional shdwForegnd As String = "", _
    Optional shdwForegndTrans As String = "", _
    Optional shdwBkgnd As String = "", _
    Optional shdwBkgndTrans As String = "", _
    Optional shdwPattern As String = "", _
    Optional shapeShdwOffsetX As String = "", _
    Optional shapeShdwOffsetY As String = "", _
    Optional shapeShdwType As String = "", _
    Optional shapeShdwObliqueAngle As String = "", _
    Optional shapeShdwScaleFactor As String = "")

    On Error Resume Next
    If (fillForegnd <> "") Then vsoShape.CellsU("FillForegnd").FormulaForceU = fillForegnd
    If (fillForegndTrans <> "") Then vsoShape.CellsU("FillForegndTrans").FormulaForceU = _
        fillForegndTrans
    If (fillBkgnd <> "") Then vsoShape.CellsU("FillBkgnd").FormulaForceU = fillBkgnd
    If (fillBkgndTrans <> "") Then vsoShape.CellsU("FillBkgndTrans").FormulaForceU = fillBkgndTrans
    If (fillPattern <> "") Then vsoShape.CellsU("FillPattern").FormulaForceU = fillPattern
    If (shdwForegnd <> "") Then vsoShape.CellsU("ShdwForegnd").FormulaForceU = shdwForegnd
    If (shdwForegndTrans <> "") Then vsoShape.CellsU("ShdwForegndTrans").FormulaForceU = _
        shdwForegndTrans
    If (shdwBkgnd <> "") Then vsoShape.CellsU("ShdwBkgnd").FormulaForceU = shdwBkgnd
    If (shdwBkgndTrans <> "") Then vsoShape.CellsU("ShdwBkgndTrans").FormulaForceU = shdwBkgndTrans
    If (shdwPattern <> "") Then vsoShape.CellsU("ShdwPattern").FormulaForceU = shdwPattern
    If (shapeShdwOffsetX <> "") Then vsoShape.CellsU("ShapeShdwOffsetX").FormulaForceU = _
        shapeShdwOffsetX
    If (shapeShdwOffsetY <> "") Then vsoShape.CellsU("ShapeShdwOffsetY").FormulaForceU = _
        shapeShdwOffsetY
    If (shapeShdwType <> "") Then vsoShape.CellsU("ShapeShdwType").FormulaForceU = shapeShdwType
    If (shapeShdwObliqueAngle <> "") Then vsoShape.CellsU("ShapeShdwObliqueAngle").FormulaForceU = _
        shapeShdwObliqueAngle
    If (shapeShdwScaleFactor <> "") Then vsoShape.CellsU("ShapeShdwScaleFactor").FormulaForceU = _
        shapeShdwScaleFactor
    On Error GoTo 0

End Function


' ============================================================================
' SetShapeMargins
' This is a utility function that sets the shape's internal margins.
' ============================================================================
Function SetShapeMargins(vsoShape As Visio.Shape, _
    Optional leftMargin As String = "", _
    Optional rightMargin As String = "", _
    Optional topMargin As String = "", _
    Optional bottomMargin As String = "")

    On Error Resume Next
    If (leftMargin <> "") Then vsoShape.CellsU("LeftMargin").FormulaForceU = leftMargin
    If (rightMargin <> "") Then vsoShape.CellsU("RightMargin").FormulaForceU = rightMargin
    If (topMargin <> "") Then vsoShape.CellsU("TopMargin").FormulaForceU = topMargin
    If (bottomMargin <> "") Then vsoShape.CellsU("BottomMargin").FormulaForceU = bottomMargin
    On Error GoTo 0
End Function



' ============================================================================
' SetShapeCharFormat
' This is a utility function that sets the shape's internal margins.
' ============================================================================
Function SetShapeCharFormat(vsoShape As Visio.Shape, _
    Optional cFont As String = "", _
    Optional cSize As String = "", _
    Optional cScale As String = "", _
    Optional cSpacing As String = "", _
```

```
    Optional cColor As String = "", _
    Optional cTransparency As String = "", _
    Optional cStyle As String = "", _
    Optional cCase As String = "")

    On Error Resume Next
    If (cFont <> "") Then vsoShape.CellsU("Char.Font").FormulaForceU = cFont
    If (cSize <> "") Then vsoShape.CellsU("Char.Size").FormulaForceU = cSize
    If (cScale <> "") Then vsoShape.CellsU("Char.Scale").FormulaForceU = cScale
    If (cSpacing <> "") Then vsoShape.CellsU("Char.Spacing").FormulaForceU = cSpacing
    If (cColor <> "") Then vsoShape.CellsU("Char.Color").FormulaForceU = cColor
    If (cTransparency <> "") Then vsoShape.CellsU("Char.Transparency").FormulaForceU = cTransparency
    If (cStyle <> "") Then vsoShape.CellsU("Char.Style").FormulaForceU = cStyle
    If (cCase <> "") Then vsoShape.CellsU("Char.Case").FormulaForceU = cCase
    On Error GoTo 0
End Function


' ============================================================================
' SetPropertyParams:
' This function creates a property in the Visio shape (if necesary) and then
' sets the parameters of the property in accordance with the values passed to
' the function.  Note that the "\0" is not a 0-length string (as in C), instead
' it is an actual string that is used to detect a Null value, because an
' undefined optional parameter cannot be set to Null.
' ============================================================================
Function SetPropertyParams( _
    vsoShape As Visio.Shape, _
    Name As String, _
    Optional Label As String = "\0", _
    Optional PropType As String = "\0", _
    Optional Format As String = "\0", _
    Optional Value As String = "\0", _
    Optional Ask As String = "\0", _
    Optional Invisible As String = "\0") As Integer

    Dim Property As String
    Dim Row As Integer

    Property = "Prop." + Name

    On Error Resume Next
    Err.Clear

    If (Not vsoShape.CellExistsU(Property, False)) Then
        Row = vsoShape.AddRow(visSectionProp, visRowLast, visTagDefault)
        vsoShape.Section(visSectionProp).Row(Row).NameU = Name
    End If

    Row = vsoShape.Cells(Property).Row

    If (Label <> "\0") Then vsoShape.CellsSRC(visSectionProp, Row, visCustPropsLabel).FormulaU = Label
    If (PropType <> "\0") Then vsoShape.CellsSRC(visSectionProp, Row, visCustPropsType).FormulaU = _
        PropType
    If (Format <> "\0") Then vsoShape.CellsSRC(visSectionProp, Row, visCustPropsFormat).FormulaU = _
        Format
    If (Value <> "\0") Then vsoShape.CellsSRC(visSectionProp, Row, visCustPropsValue).FormulaU = Value
    If (Ask <> "\0") Then vsoShape.CellsSRC(visSectionProp, Row, visCustPropsAsk).FormulaU = Ask
    If (Invisible <> "\0") Then vsoShape.CellsSRC(visSectionProp, Row, visCustPropsInvis).FormulaU = _
        Invisible

    SetPropertyParams = Row

    If Err.Number <> 0 Then
        Debug.Print Error(Err.Number)
    End If
    On Error GoTo 0

End Function
```

```
' ========================================================================
' DeleteProperty:
' This function deletes a property from the shape.
' ========================================================================
Function DeleteProperty( _
    vsoShape As Visio.Shape, _
    Name As String)

    Dim Property As String
    Dim Row As Integer

    Property = "Prop." + Name

    On Error Resume Next
    Err.Clear

    If (vsoShape.CellExistsU(Property, False)) Then
        Row = vsoShape.Cells(Property).Row
        vsoShape.DeleteRow visSectionProp, Row
        DeleteProperty = 1
    Else
        DeleteProperty = 0
    End If

    If Err.Number <> 0 Then
        Debug.Print Error(Err.Number)
    End If
    On Error GoTo 0

    End Function

' ========================================================================
' SetPropertyValue:
' This function creates a property in the Visio shape (if necesary) and then
' sets the value property to the value specified in the parameter.
' ========================================================================
Function SetPropertyValue( _
    vsoShape As Visio.Shape, _
    Name As String, _
    Value As String) As Integer

    Dim Property As String
    Dim Row As Integer

    Property = "Prop." + Name

    On Error Resume Next
    Err.Clear

    If (Not vsoShape.CellExistsU(Property, False)) Then
        Row = vsoShape.AddRow(visSectionProp, visRowLast, visTagDefault)
        vsoShape.Section(visSectionProp).Row(Row).NameU = Name
    End If

    Row = vsoShape.Cells(Property).Row
    vsoShape.CellsSRC(visSectionProp, Row, visCustPropsValue).FormulaU = Value

    SetPropertyValue = Row

    If Err.Number <> 0 Then
        Debug.Print Error(Err.Number)
    End If
    On Error GoTo 0

End Function


' ========================================================================
```

```
' GetPropertyValue:
' This function reads a property in the Visio shape and then returns it
' to the caller.
' ============================================================================
Function GetPropertyValue( _
    vsoShape As Visio.Shape, _
    Name As String) As String

    Dim Property As String
    Dim Row As Integer

    Property = "Prop." + Name

    On Error Resume Next
    Err.Clear

    If (vsoShape.CellExistsU(Property, False)) Then
        GetPropertyValue = vsoShape.CellsU(Property).ResultStr(0)
    Else
        GetPropertyValue = ""
    End If

    If Err.Number <> 0 Then
        Debug.Print Error(Err.Number)
    End If
    On Error GoTo 0

End Function


' ============================================================================
' SetPropertyVisibility:
' This function will hide or unhide properties
' ============================================================================
Function SetPropertyVisibility( _
    vsoShape As Visio.Shape, _
    Name As String, _
    Hidden As Boolean)
    Dim Property As String
    Dim Row As Integer

    Property = "Prop." + Name

    On Error Resume Next
    Err.Clear

    If (vsoShape.CellExistsU(Property, False)) Then
        Row = vsoShape.Cells(Property).Row
        vsoShape.CellsSRC(visSectionProp, Row, visCustPropsInvis).FormulaU = Hidden
    End If


    If Err.Number <> 0 Then
        Debug.Print Error(Err.Number)
    End If
    On Error GoTo 0

End Function


' ============================================================================
' ConnectShapes
'   This function will connect the specified shapes.  It will use connecting
'   points if they are specified.
' ============================================================================
Function ConnectShapes( _
    FromShape As Visio.Shape, _
    ToShape As Visio.Shape, _
    Optional FromCP As String = "", _
```

```
    Optional ToCP As String = "", _
    Optional zOrder As Integer)

    Dim vsoConn As Visio.Shape
    Dim vsoCell1 As Visio.Cell
    Dim vsoCell2 As Visio.Cell

    If (ToShape.CellExistsU("Prop.Distribution", False) And _
        ToShape.CellExistsU("Prop.Switch", False)) Then
        If (ToCP = "") Then ToCP = "1"
    End If

    On Error Resume Next
    Set vsoConn = ActivePage.Drop(ActiveDocument.Masters.ItemU("Dynamic connector"), 0#, 0#)

    If (zOrder > 0) Then
        vsoConn.CellsU("DisplayLevel").FormulaU = zOrder
    End If

    Set vsoCell1 = vsoConn.CellsU("BeginX")

    Err.Clear
    If (FromCP <> "") Then
        Set vsoCell2 = FromShape.CellsU("Connections." & FromCP)
        If (Err.Number = 0) Then vsoCell1.GlueTo vsoCell2
    End If

    If (FromCP = "" Or Err.Number <> 0) Then
        Set vsoCell2 = FromShape.CellsSRC(1, 1, 0)
        vsoCell1.GlueTo vsoCell2
    End If

    Set vsoCell1 = vsoConn.CellsU("EndX")

    Err.Clear
    If (ToCP <> "") Then
        Set vsoCell2 = ToShape.CellsU("Connections." & ToCP)
        If (Err.Number = 0) Then vsoCell1.GlueTo vsoCell2
    End If

    If (ToCP = "" Or Err.Number <> 0) Then
        Set vsoCell2 = ToShape.CellsSRC(1, 1, 0)
        vsoCell1.GlueTo vsoCell2
    End If

End Function

' ============================================================================
' ReadConnectivity:
'   This function will evaluate the vsoShape object and will return an array
'   of ConnectedShapes objects that represents all of the connections that
'   are going in to or coming out of vsoShape.
' ============================================================================
Function ReadConnectivity(vsoShape As Visio.Shape) As ConnectedShapes()
    Dim result() As ConnectedShapes
    Dim vsoConn As Visio.Shape
    Dim partID As Integer

    ' ========================================================
    ' Test if this item is not a connector and has connections.
    ' Only procede if both of those things are true.
    ' ========================================================
    If ((Not vsoShape.LineStyle = "Connector") And (vsoShape.FromConnects.Count > 0)) Then
        ' ============================================================
        ' Allocate the result array.
        ' ============================================================
        ReDim result(vsoShape.FromConnects.Count)

        ' ============================================================
```

```
        ' Walkthrough the connections and process each one as you go.
        ' ============================================================
        For i = 1 To vsoShape.FromConnects.Count
            Set vsoConn = vsoShape.FromConnects(i).FromSheet
            Set result(i).connector = vsoConn
            Set result(i).FromShape = Nothing
            Set result(i).ToShape = Nothing
            result(i).ConnectCount = 0
            result(i).FromConnectionPoint = ""
            result(i).ToConnectionPoint = ""
            result(i).Incoming = False

            If vsoConn.Connects.Count > 0 Then
                Set result(i).FromShape = vsoConn.Connects(1).ToSheet
                result(i).ConnectCount = 1

                partID = vsoConn.Connects(1).ToPart
                If (partID >= visConnectionPoint) Then
                    If (vsoConn.Connects(1).ToCell.RowName <> "") Then
                        result(i).FromConnectionPoint = vsoConn.Connects(1).ToCell.RowName
                    Else
                        result(i).FromConnectionPoint = "Row_" & CStr(partID - visConnectionPoint + 1)
                    End If
                End If
            End If

            If vsoConn.Connects.Count > 1 Then
                Set result(i).ToShape = vsoConn.Connects(2).ToSheet
                result(i).ConnectCount = 2

                partID = vsoConn.Connects(2).ToPart
                If (partID >= visConnectionPoint) Then
                    If (vsoConn.Connects(2).ToCell.RowName <> "") Then
                        result(i).ToConnectionPoint = vsoConn.Connects(2).ToCell.RowName
                    Else
                        result(i).ToConnectionPoint = "Row_" & CStr(partID - visConnectionPoint + 1)
                    End If
                End If

                If (result(i).ToShape.ID = vsoShape.ID) Then
                    result(i).Incoming = True
                End If
            End If
        Next
    End If

ReadConnectivity = result
End Function
```