EPRI | ELECTRIC POWER RESEARCH INSTITUTE

Smart Distribution Applications and Technologies

Evaluation of Distribution Reconfiguration Functions in Advanced Distribution Management Systems, Example Assessments of Distribution Automation Using Open Distribution Systems Simulator

1020090

Smart Distribution Applications and Technologies

Evaluation of Distribution Reconfiguration Functions in Advanced Distribution Management Systems, Example Assessments of Distribution Automation Using Open Distribution Systems Simulator

1020090

Technical Update, September 2011

EPRI Project Managers

M. McGranaghan
M. Olearczyk
R. Dugan

ELECTRIC POWER RESEARCH INSTITUTE
3420 Hillview Avenue, Palo Alto, California 94304-1338 ▪ PO Box 10412, Palo Alto, California 94303-0813 ▪ USA
800.313.3774 ▪ 650.855.2121 ▪ askepri@epri.com ▪ www.epri.com

DISCLAIMER OF WARRANTIES AND LIMITATION OF LIABILITIES

THE FOLLOWING ORGANIZATION, UNDER CONTRACT TO EPRI, PREPARED THIS REPORT:

MelTran, Inc.

This is an EPRI Technical Update report. A Technical Update report is intended as an informal report of continuing research, a meeting, or a topical study. It is not a final EPRI technical report.

NOTE

For further information about EPRI, call the EPRI Customer Assistance Center at 800.313.3774 or e-mail askepri@epri.com.

ACKNOWLEDGMENTS

The following organization, under contract to the Electric Power Research Institute (EPRI), prepared this report:

MelTran, Inc.
90 Clairton Boulevard, Suite A
Pittsburgh, PA 15236

Principal Investigator
T. McDermott

This document describes research sponsored by EPRI.

This publication is a corporate document that should be cited in the literature in the following manner:

Smart Distribution Applications and Technologies: Evaluation of Distribution Reconfiguration Functions in Advanced Distribution Management Systems, Example Assessments of Distribution Automation Using Open Distribution Systems Simulator. EPRI, Palo Alto, CA: 2011. 1020090.

PRODUCT DESCRIPTION

In 2008, the Electric Power Research Institute (EPRI) provided its Distribution System Simulator (DSS) software tool to the industry as open-source software called OpenDSS. One of the main purposes for making it open source was to provide a convenient platform for users to develop and test algorithms for advanced distribution automation (ADA) functions. This report describes the basics of how to apply the OpenDSS program to assess ADA algorithms that might be found in a distribution management system (DMS).

Results and Findings

Three examples are provided:

- Civinlar's branch exchange reconfiguration example
- A substation control example
- A volt/VAR control example

These examples are useful for researchers and other users who need some assistance getting started with ADA algorithm simulation. They would also be helpful to DMS vendors who would like to test and debug their control algorithms. The examples are demonstrated using the familiar Microsoft® Excel spreadsheet program via Visual Basic for Applications (VBA). This is a commonly available tool that should allow most readers to duplicate the results easily.

Challenges and Objectives

This report is intended for researchers investigating ADA algorithms. In addition to utility planners, this report would be especially beneficial to academics with limited knowledge of distribution system behavior who are investigating new ideas and want to determine how the utility distribution system will respond. The goals are to 1) demonstrate the use and value of OpenDSS in modern grid design and 2) show end users how to customize OpenDSS for advanced applications.

Applications, Value, and Use

ADA algorithms are implemented for testing in software outside the OpenDSS program. This may be software such as MATLAB, Excel, C/C++, and Python. Many potential users are not only unfamiliar with distribution systems, but also are not familiar with how to use programs through the Windows Component Object Model (COM) automation interface. This report demonstrates how to drive and control OpenDSS through its COM interface.

This report makes a special contribution that should accelerate the advancement of tools for planning and analysis of various issues related to the deployment of the smart grid. The expected outcome is that improved algorithms for ADA will be produced more quickly and will find their way into DMS products that will benefit utilities.

Approach

The goal of this project was to demonstrate how to use OpenDSS to model ADA control algorithms. Three example problems were selected, and computer code was developed to drive OpenDSS. The code is documented to serve as a starting point for users of the program.

Keywords
Advanced distribution automation (ADA)
Distribution management system (DMS)
Distribution reconfiguration
OpenDSS
Substation automation

CONTENTS

LIST OF FIGURES

1
INTRODUCTION

As utilities modernize their distribution systems to include wider use of advanced distribution automation (ADA) schemes, they need to assess their distribution circuit designs and configurations, including the associated control and protection systems. The simulation of these ADA schemes and control systems can lead to a better understanding of the value of these emerging capabilities and the ways they can be most effectively used.

The Distribution System Simulator (DSS) is a comprehensive simulation tool for electric utility distribution systems. It has been in use since 1997 as a tool to investigate distribution system analysis problems that required innovative approaches or that were difficult to solve with conventional tools. The program was initially developed to support nearly all aspects of distribution planning with distributed generation (DG). In September 2008, the Electric Power Research Institute (EPRI) released the program as an open source project called OpenDSS [1]. One motive was to spur the advancement of "smart grid" efforts by providing researchers and developers with a flexible and powerful distribution system modeling tool for investigating new algorithms and control schemes.

Problem Statement

There are two complementary goals for this project [2]:

- Demonstrate the use and value of OpenDSS in modern grid design
- Show how end users can customize OpenDSS for advanced applications

To meet the first objective, an existing OpenDSS feeder model has been identified for the simulation of centralized reactive power control on a feeder. This involves collection of measurements from points on a feeder, processing by a centralized algorithm, and then dispatch of capacitor banks and regulator taps at different points on the feeder. Simulation of communication system bandwidth, latency, and reliability is not in the scope for this project.

To meet the second objective, Microsoft Excel was used to implement the controls and to supervise OpenDSS. Three examples in Visual Basic for Applications (VBA) show automated switch control for loss minimization, adaptive tap changer control in a substation, and reactive power control on a realistic feeder model. Utilities and researchers can use these examples to develop their own applications. Control parameters, measurement points, and algorithms can all be optimized using this type of simulation platform.

Applications

Over the past 20 years, university researchers have developed computer-based control algorithms for distribution circuits like the ones in the following list. In many cases, these methods have been developed well in advance of market needs, but some have been adopted in commercial products. With the availability of more measurement data, communication channels, and control devices, the body of existing academic work may provide some guidance on possible smart grid applications for the OpenDSS platform.

- Capacitor bank optimization – Finds the locations, sizes, and operating strategies for capacitor banks to reduce system losses or to improve voltage profiles.
- Reconfiguration for loss reduction – Opens or closes feeder tie switches to reduce total system losses.
- Load and phase balancing – This is a special case of switch reconfiguration for loss reduction. The objective is to balance load currents among feeders and among individual phases on the feeders. This also tends to improve the reserve capacity.
- Service restoration – This is another special case of switch reconfiguration in which the system starts in a radial but unconnected state, meaning that some loads are not served. The objective is to restore as much load as possible, subject to voltage and current constraints, while keeping the system radial. Some components are faulted, which will preclude some switching operations.
- Modern optimization techniques – Most of the current academic research in this area makes use of genetic algorithms (GA), simulated annealing (SA), automated neural networks (ANN), ant colony optimization (ACO), or some other modern optimization techniques adapted from other research fields. Generally, this involves linking a general-purpose optimization package to a power system model. The most significant challenge is representing the power system characteristics and constraints (especially radial configuration) in a form that is suitable for the optimization technique. This is especially true for GA, but recent work with matroids has been more promising. For OpenDSS, there would be a benefit to interfacing with widely used optimization packages in order to take advantage of powerful techniques already developed in a larger research community. This is an important reason for supporting MATLAB interfaces to OpenDSS.
- Agent-based methods – There have been several recent papers using agent-based methods for distributed control of the power distribution system. Using local data and processing, multiple software agents can act to achieve system-level objectives, if the algorithms have been designed properly. The advantage would be much less reliance on communication channels. OpenDSS could simulate these agents through a DLL or COM interface, providing a test bed for their development.

In addition to these, distribution system state estimation is a key enabler for smart grid applications, and it presents different technical challenges than the well-developed transmission system state estimators do. Other projects funded by EPRI, the U.S. Department of Energy, the Centre for Energy Advancement Through Technological Innovation (CEATI), and the California Energy Commission are addressing the need for state estimation.

Interest in DG also remains high, especially from renewable energy sources. The original purpose for OpenDSS was to provide analysis and planning functions for DG, and that work continues. If a utility can use DG as part of a reactive dispatch scheme, that becomes a form of ADA. Even though, according to IEEE Std. 1547, DG is not supposed to actively regulate voltage, the utility could dispatch DG reactive power to help manage voltage profiles.

2
FEEDER MODELS

This section describes feeder models that were used for ADA simulation examples in OpenDSS.

Civanlar's Reconfiguration Example

Figure 2-1 depicts a simple three-feeder test system used in a classical paper on distribution system reconfiguration to minimize losses [3]. The substation bus voltage is 23 kV, the node voltage outputs are on a 120-volt base, and the branch flows are in amperes (A). Loads and capacitor banks are connected only at the numbered buses, but each line segment can be switched in or out of service. There are no adjustable transformer taps in this model. The total system losses are reduced by opening segment 9-11 instead of 5-11, and by opening segment 8-10 instead of 10-14.

This example was chosen to provide a simple introduction and to encourage participation by academic researchers. Most graduate students in distribution system modeling would already be familiar with this test circuit and application.

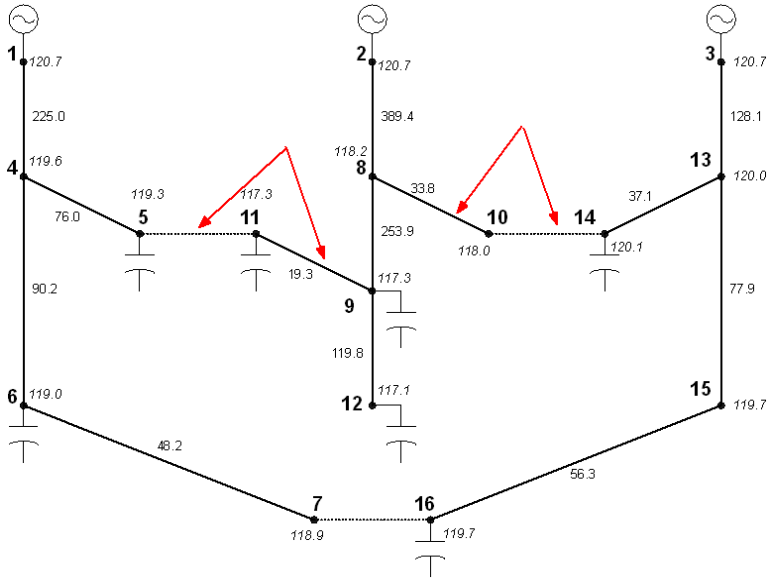

Figure 2-1
Civanlar's Reconfiguration Example with Three Feeders [3]

Substation Control Example

Figure 2-2 depicts a 138-kV/13.8-kV substation serving a low-voltage secondary network through 29 primary feeders. All four transformers are heavily loaded. The system operators control the substation capacitor banks manually and also manually adjust the substation transformer load tap changers (LTC) according to the total substation power demand. Figure 2-2 includes the LTC setting schedule in tabular format, with target voltage based on the actual

potential transformer (PT) ratio. In addition, the LTC has line drop compensator (LDC) settings that are fixed. A simplified version of this circuit was used for the second example, illustrating capacitor and regulator controls. The simplifications include:

- Combining the four transformers into a parallel equivalent. The regulator current transformer (CT) ratings are scaled up to use the actual R and X settings.
- Paralleling the 29 feeders and the network load into an equivalent on the 13.8-kV bus.
- Increasing the total available substation capacitance, allowing more compensation of the load power factor.

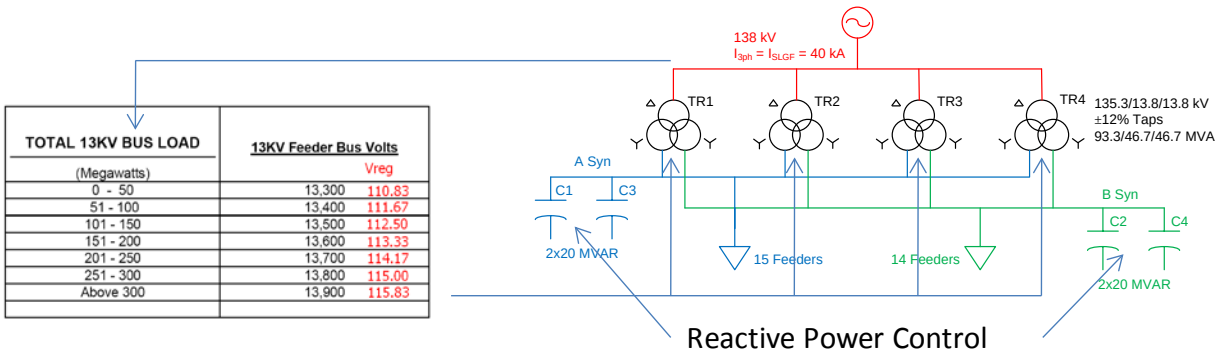

Figure 2-2
Substation Model with Controlled Capacitors and Tap Changers

Volt/VAR Control Example

Figure 2-3 shows the layout of line segments for two feeders already modeled in OpenDSS for a previous EPRI project [4]. The segment widths in the figure are weighted by current flow, and the substation is located near the left end of the thickest blue lines as indicated. With loading set at 30% of the connected transformer kVA, the total power delivered was 6.86 MW, with losses of 3.9%. The customer load is all defined on service points, which are fed from service transformers having their own kVA rating. The loads were allocated for OpenDSS using 400 A per phase at the head of each feeder. The model includes the following components:

- 1 substation source at 12.47 kV
- 2 feeders, denoted 34232 and 34236
- 1175 transformers (some associated with 79 three-phase banks, which were not used in this DSS model)
- 1037 service points
- 2351 line segments, including both overhead and underground
- 5 shunt capacitors, two of which are supervisory control and data acquisition (SCADA) controlled, by phase. The sizes are 300 and 900 kilovolt amperes reactive (kVAR), and three 600-kVAR banks
- 57 switches
- 10 reclosers, two of which are SCADA controlled
- 1 sectionalizer

- 136 fuses
- 4 voltage regulators (Three of these are auto-boosters with just four taps operating in the boost direction only from 100% to 110%. The other is a standard line regulator.)

The feeders also include six line post sensors for capacitor bank control and other monitoring functions. A line post sensor measures real power, voltage, current, and power factor by phase, along with total current and voltage distortion. These measurements are integrated into the SCADA system at 15-minute intervals. Both of the SCADA-controlled capacitor banks have line post sensors, as indicated in Figure 2-3. Neither SCADA-controlled recloser has a line post sensor.

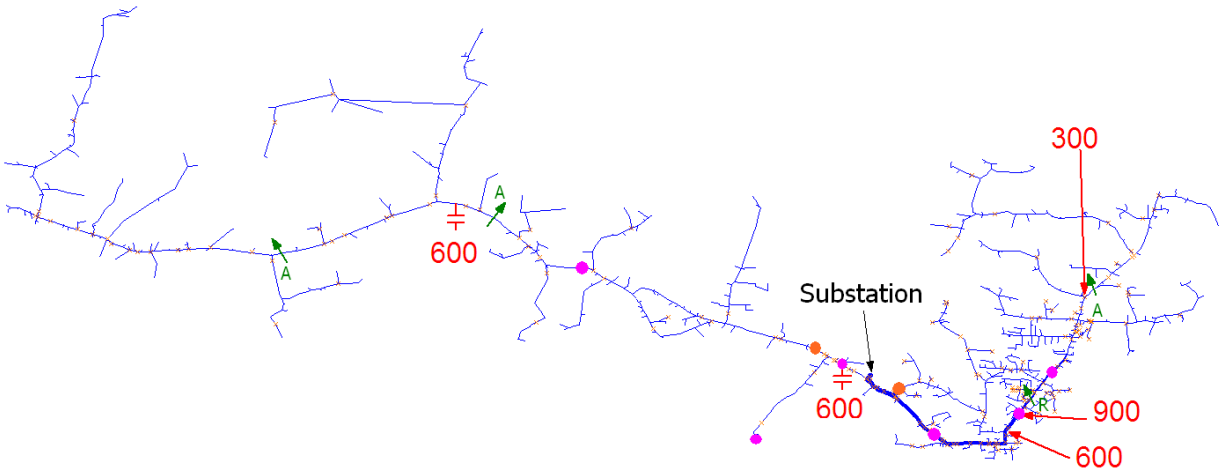

Figure 2-3
Feeders for Capacitor Switching Control Example [4]

3
MODELING PLATFORM DEVELOPMENTS

EPRI report *Control System Simulation in the Distribution System Simulator (DSS)* [2] outlined a development plan to upgrade OpenDSS capabilities for ADA simulation. This section updates the plan as built. More detailed documentation can be found in the Tech Notes and the Wiki at http://sourceforge.net/projects/electricdss/ [1].

Circuit Elements

The base circuit element attributes were expanded, both to support ADA simulation and to support common information model (CIM) interoperability testing:

- Added a long integer ID, which can serve as a device handle for external controllers in ADA simulation. This does not have to be persistent each time OpenDSS loads a model.
- Added an optional display name, which does not have to be unique. These are used in most commercial packages.
- Added a globally unique identifier (GUID), which is mandatory for CIM testing. The GUID is unique not only within an OpenDSS model, but also among other software packages. It is also persistent each time OpenDSS executes. This persistence is required for incremental updates to work in CIM, regardless of the time between data exchanges. OpenDSS creates GUIDs automatically, only when needed, and when a GUID does not already exist. Commands were provided to read and write GUIDs, similar to the handling of bus coordinates.

Transformer library support was expanded with the *XfmrCode* class for transformers that share parameters. An optional XfmrBank attribute was added to collect single-phase transformers into a bank. The bank structure is important for both CIM and MultiSpeak data transfer and may be a useful addition to capacitors, load, switches, etc.

Line library support was expanded with a *LineSpacing* class that holds only X and Y wire coordinates. It is used with the Spacing and Wires attributes of the Line object. These support a line library of pole or structure types, with independent wire selection.

Common object model (COM) interface enumerations were added for LoadStatus, LoadModels, CapControlModes, and ActionCodes (detailed later). These enumerations provide support for tool-tip help in Excel VBA and other programming languages.

SwtControl Class

Any component in OpenDSS has an inherent switch in each terminal, but there is no easy way to identify and keep track of the “real” switches. Following the established pattern of separate control classes for capacitor banks, tap changers, and overcurrent protection, a new *SwtControl* class was implemented. This allows switches to receive special handling in both algorithms and graphical display, since the *SwtControl* information keeps a reference to the switched component and terminal. The parameters include:

- *Name* – The text identifier, so that SwtControl.Name is unique (inherited parameter).
- *Enabled* – Takes the controller in or out of the solution (inherited parameter).
- *Like* – Copies Action and Delay from another *SwtControl* (inherited parameter).
- *SwitchedObj* – The fully qualified name of the switched object, such as Line.Segment2.
- *SwitchedTerm* – The number of the switched terminal, typically 1 or 2.
- *Action* – Sets the switch open or closed. Since the *SwtControl* does not have automatic operation on a time-current curve (TCC), this is not the same as Action for the overcurrent protection controls, for which the *Action* parameter overrides the TCC operation.
- *Locked* – If set, no switch operations are allowed until the Lock is removed. This means that *Action* parameter changes are ignored if the Lock is active.
- *Delay* – The time in seconds between setting the *Action* parameter and when the switch actually opens or closes. This could be 1 second for automatic operation or 7200 seconds for manual operation by a crew that has to be dispatched.

Standard COM Action Codes

Long integer IDs from the new *CktElement* interface can serve as device handles for the *CtrlQueue* events. In addition, standard action codes are defined for these events, allowing any controller to handle and interpret these events in a consistent way. The standard action codes include:

- dssActionOpen, applies to *SwtControl, CapControl, Relay, Recloser, Fuse*
- dssActionClose, applies to *SwtControl, CapControl, Relay, Recloser, Fuse*
- dssActionReset, applies to any controller
- dssActionLock, applies to *SwtControl*
- dssActionUnlock, applies to *SwtControl*
- dssActionTapUp, applies to *RegControl* and possibly *CapControl* (that is, step up)
- dssActionTapDown, applies to *RegControl* and possibly *CapControl* (that is, step down)
- dssActionNone, applies to any control

EPRI may wish to assign different starting numbers to different “vendors” to avoid potential conflicts. For example, an external controller might be operating a particular device, identified by its handle. When inspecting the control queue, this external code might find other events pertaining to that device handle. In that situation, we do not want different controllers using the same action codes that mean different things. Vendor or Developer IDs could help address this issue.

Sensors COM Interface

A complete read/write COM interface was added to support other state estimation projects. This interface is also useful to ADA simulations that incorporate measurements and state estimation algorithms. The sensor is typically defined where V, I, P, and/or Q are physically measured in a circuit. The sensor accounts for PT ratio and connection, measurement error estimate, weighting, and other parameters associated with state estimation. In some cases, the monitor is more appropriate for modeling measurements in ADA.

Topology COM Interface

OpenDSS already had some functions to analyze circuit element connectivity, but these were not exposed through the COM interface. To support reconfiguration and other switch control applications, a new COM interface was defined. Internally, a tree-like structure of branches is built on demand, based on the same process used to construct EnergyMeter zones. The COM interface does not require that EnergyMeters be present in the OpenDSS model. However, the use of EnergyMeters may offer better control over where the topology tree defines end points and loop points. The OpenDSS user manual has more background on how meter zones are built.

The topology tree is rebuilt after switching operations or other model changes that might invalidate the tree. In the OpenDSS executive, a new "show topology" command illustrates most of the features described below for the loaded circuit model.

The topology properties are:

- *NumLoops* – Returns the number of loops in the circuit.
- *NumIsolatedLoads* – Returns the number of loads with no voltage.
- *NumIsolatedBranches* – Returns the number of branches (that is, power delivery elements, including shunt capacitors) that are not connected.
- *AllLoopedPairs* – Returns a variant array of branch pair names that are connected to make a loop. Each connection appears twice in the list, as "from-to" and again as "to-from".
- *AllIsolatedLoads* – Returns a variant array of the load names that have no voltage.
- *AllIsolatedBranches* – Returns a variant array of the fully qualified branch names that are not connected to the source.
- *First* – Sets the tree iterator to the first branch, which is always the main voltage source. Also sets the active circuit element to that branch. Returns 1 if successful, 0 if no more branches.
- *Next* – Sets the next tree branch active, and sets the active circuit element to that branch. Returns 1 if successful, 0 if no more branches.
- *BranchName* – A read/write property that either sets the active tree branch by fully qualified name or gets the fully qualified name of the active branch. At present, this is the only means of re-selecting a particular branch in the topology tree.
- *BusName* – A read/write property that either sets the active tree branch by the name of the bus furthest from the source or gets the name of the active branch bus furthest from the source.
- *ActiveBranch* – In this version, this read-only property returns ActiveLevel. In a future version, this may become a read/write property that uses a numerical branch index for more efficient repeat method calls.
- *ActiveLevel* – Returns the topological level of the active branch. This is the number of branches away from the main voltage source, which is at level 0.
- *FirstLoad* – Sets the active circuit element to the first load connected at the topology ActiveBranch. Returns 0 if no more.
- *NextLoad* – Sets the active circuit element to the next load connected at the topology ActiveBranch. Returns 0 if no more.

- *ForwardBranch* – Sets the ActiveBranch to the next one in the tree. This branch may not have a lower ActiveLevel; the level will decrease if the tree iterator is already at a branch end. Returns 0 if no more.
- *BackwardBranch* – Sets the ActiveBranch to the parent branch in the tree. If successful, this always moves back toward the source, and the ActiveLevel will decrease.
- *LoopedBranch* – Sets the active circuit element to one that is connected in a loop to the currently active topology branch. Returns 0 if there is no loop connection here. This does not change the topology tree's ActiveBranch. Also, directly parallel branches are handled in a separate method; LoopedBranch returns 0 when there is only a parallel branch.
- *ParallelBranch* – Sets the active circuit element to one directly connected in parallel to the currently active topology branch. Returns 0 if there is no directly parallel branch here. This does not change the topology tree's ActiveBranch.

The topology processing code in OpenDSS was made more efficient for this project. In the future, the COM interface can be further improved with numerical indexing of the tree branches. Numerical tree indexing can also be used in future support of multiple loops and multiple parallel branches at the same connection point.

CktElement COM Interface

The following properties were added to the *CktElement* interface. The first three were added to support CIM and geographic information system (GIS) integration. The last four were added to support ADA simulation.

- *DisplayName* – The new non-unique display name (read/write)
- *ID* – The new long integer ID (read only)
- *GUID* – The new GUID (read only)
- *EnergyMeter* – The meter (that is, feeder) that this element has been assigned to (read only)
- *Controller* – The fully qualified name of the *CapControl*, *RegControl*, or *SwtControl* as appropriate
- *HasVoltControl* – True if a *CapControl* or *RegControl* manages this element
- *HasSwtControl* – True if a *SwtControl* manages this element

Meters COM Interface

The following read-only properties were added to the *Meters* interface. They provide some functionality of the new *Topology* interface by individual EnergyMeter.

- *CountEndElements* – Returns the number of open end points in the active meter zone.
- *AllEndElements* – Returns a collection of *CktElement* names at the open ends, that is, elements that have no children in a tree view of the meter zone. This can be used to identify points where low voltage might occur.
- *CountBranches* – Returns the total number of (power delivery) branches in the active meter zone.
- *AllBranchesInZone* – Returns a collection of branch names in the active meter zone.

COM Interfaces to Existing Controller Classes

Some of the circuit element classes already had their own custom COM interfaces, in addition to *CktElement*. This included *Lines* and *Generators*, with *Loads* under development. In addition, there were custom COM interfaces for *Buses*, *Meters*, and *Monitors*. This list expands as OpenDSS continues development. For the ADA simulation examples in this project, the following custom COM interfaces were needed and implemented immediately:

- *CapControl*, along with a *Capacitor* interface for its controlled element
- *Capacitors*, allowing convenient reading and setting of kVAR sizes for ADA
- *RegControl*, along with a *Transformer* interface for its controlled element
- *Transformers*, allowing convenient reading and setting of taps for ADA

The related *SwtControl* interface was new, and described earlier. All of these can be accessed through the *Circuit* interface.

DSS Events COM Interface

OpenDSS provides two different ways for an external program or script to interact with time step and control step solutions. Figure 3-1 shows the general structure of a time step solution within OpenDSS. First, the voltages and currents are solved ("solve circuit"), and then all of the controls are checked for any changes ("check controls" and then "control actions done?"). The solution converges when all of the control responses have settled out in addition to a converged solution for voltage and currents. The solution then proceeds to the next time step. A static solution also works as depicted in Figure 3-1; time steps are taken until all control responses have settled, even though the reported solution time does not change.

Through a COM *Solution* interface, OpenDSS allows an external program or script to control each step in the solution loop through individual COM interface calls. The **blue** labels in Figure 3-1 illustrate that method. The external code takes responsibility for proper sequencing of the steps, and there is no way for two different scripts to manage OpenDSS at the same time.

In this project, a second method of interaction has been implemented using a COM event source, called *DSSEvents*. Three call-back function stubs have been defined, indicated with **red** labels in Figure 3-1.

- *InitControls* – Called when OpenDSS is ready to begin a new solution. External code should use this opportunity to initialize internal state variables, allocate or connect to other resources, make initial settings, etc.
- *CheckControls* – Called when OpenDSS has finished iteration for voltage and current, but before it has checked internal controls (currently *CapControl*, *RegControl*, and *SwtControl*). External code should examine solved voltages and currents, decide on making any control changes, and implement them into the system model through other COM interfaces.
- *StepControls* – Called when OpenDSS has converged for controls, voltage, and current at the present time step. External code should update internal state variables, log results, and take other actions necessary to prepare for the next solution time.

This kind of program interface has been implemented via “callback functions” on earlier software development platforms. With COM, a program uses *DSSEvents* by implementing an event sink and subscribing to events from the *DSSEvents* interface. The next section of this report provides an example in Excel VBA.

There are two main advantages of the *DSSEvents* interface:

- OpenDSS remains in control of the solution loop. The external code has to manage less detail, so there should be a lower learning curve, less risk of error, and less risk of breaking changes in OpenDSS.
- Any number of COM programs may subscribe to *DSSEvents*. That means that different controllers, possibly even from different software developers, can work together in the same simulation.

The same pattern can be used for third-party power delivery or power conversion elements interfaced to OpenDSS using COM events. To fully realize these advantages, OpenDSS would need more functionality to find, register, and load the event sinks. Furthermore, these event-sink models should be callable from the standalone executable version of OpenDSS, which presently has no COM interface at all. These points should be considered in a possible re-partitioning of OpenDSS to support non-Windows platforms, console-mode version, service or daemon version, 32-bit and 64-bit operating systems, etc.

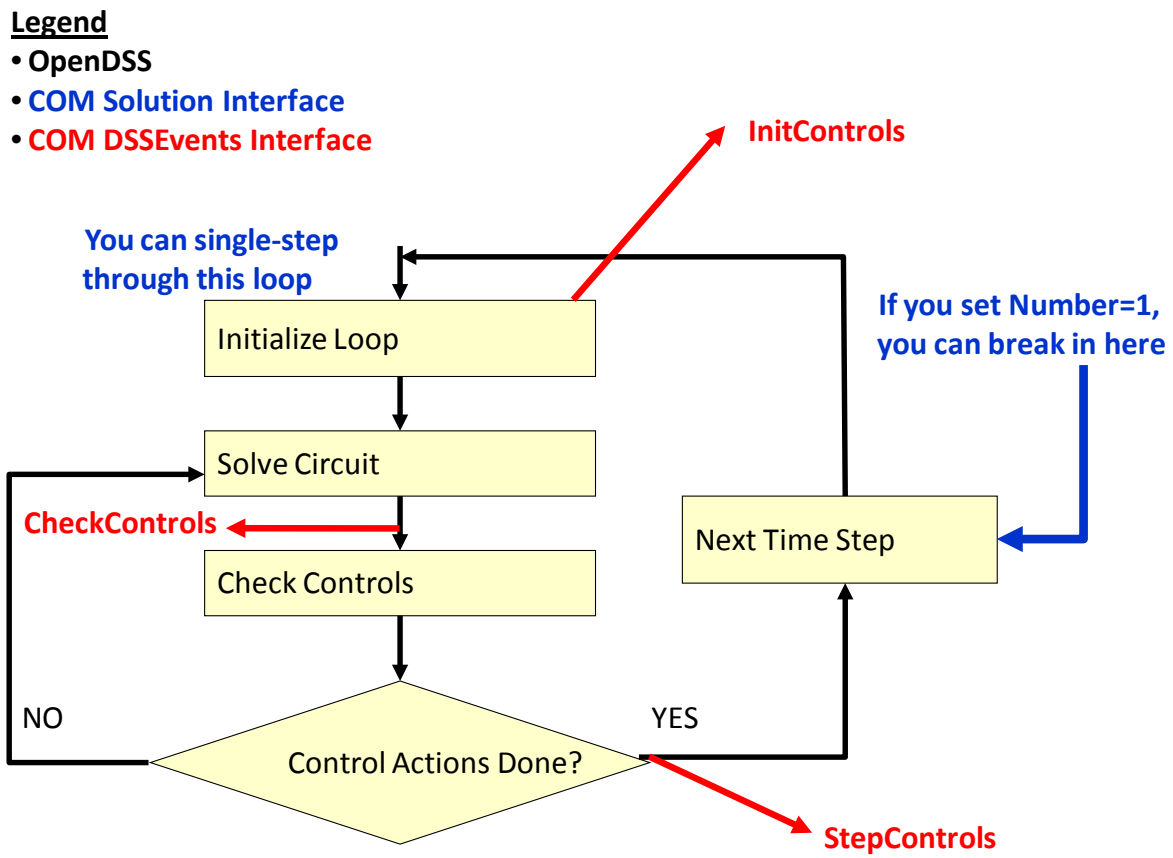

Figure 3-1
OpenDSS Solution Loop with DSSEvents and Solution Interfaces

4
EXAMPLES

This section describes the implementation of ADA simulation for three examples. All simulations were controlled from Microsoft Excel, using the OpenDSS COM interface called from Visual Basic for Applications (VBA). Each example includes VBA code annotations that focus on the new material. In order to follow and understand these annotations, the reader should be familiar with available OpenDSS training materials on COM scripting [1]. Familiarity with the Excel object model is also important; please consult the Microsoft on-line help or one of the trade books on VBA programming or macro programming in Excel.

Most of the ADA examples use a "Preamble" function like the one following. Annotations are provided in ***bold italics*** for this report; they are not comments in the VBA code.

```
Private Sub Preamble()
    gPath = Range("DataPath")        Read the model file and path names from Excel sheet
    gBase = Range("BaseFile")

    Set eng = CreateObject("OpenDSSEngine.DSS")      Starts  OpenDSS
    eng.start (0)
    Set txt = eng.Text               Load base file name using the Text interface of  OpenDSS
    txt.Command = "clear"
    txt.Command = "compile " & gPath & gBase

    Set ckt = eng.ActiveCircuit      Circuit interface
    Set swt = ckt.SwtControls        new SwtControl interface on the active circuit
    Set cap = ckt.CapControls        CapControl interface
    Set reg = ckt.RegControls        RegControl interface
    Set mon = ckt.Monitors           Monitors interface
    Set mtr = ckt.Meters             (Energy)Meters interface
    Set topo = ckt.Topology          new Topology interface
End Sub
```

Reconfiguration Example

This example uses the *Topology* and *SwtControl* interfaces on Civanlar's Figure 2-1 circuit. The base model is contained in *Civanlar.DSS*, with VBA code in *DSS_Reports.XLS*. Figure 4-1 shows test outputs from exercising the following VBA code on the base configuration. It is the first half of a function called "SwitchingSummary," which is called as an Excel macro.

```
Public Sub SwitchingSummary()
    Dim i, r As Integer
    Dim V1, V2 As Double
    Dim an As Variant, s As String

    Preamble                 Loads  OpenDSS and the Civanlar.DSS circuit, sets up ckt object

    Set ws = ActiveWorkbook.Worksheets("Switching")

    ckt.Solution.Solve       Base case solution as defined in Civanlar.DSS

    i = swt.First      Start a loop over all SwtControls in the circuit
    r = 2
    While i > 0
        ws.Cells(r, 1) = swt.name            write SwtControl name and status to the sheet
        ws.Cells(r, 2) = swt.SwitchedObj
        ws.Cells(r, 3) = swt.SwitchedTerm
        ws.Cells(r, 4) = swt.IsLocked
        ws.Cells(r, 5) = swt.Action
```

```
    ws.Cells(r, 6) = ""          blank out the three voltage output columns
    ws.Cells(r, 7) = ""
    ws.Cells(r, 8) = ""
    Set elem = ckt.CktElements(swt.SwitchedObj)  looks up the switched element by name
    If swt.Action = dssActionOpen Then   if the switch is actually open
        V1 = elem.SeqVoltages(1)                get positive sequence voltage at each end
        V2 = elem.SeqVoltages(4)
        ws.Cells(r, 6) = CStr(V1)        write each switch-end voltage to the sheet
        ws.Cells(r, 7) = CStr(V2)
        On Error Resume Next ' skip NaN  if one bus is isolated, DSS returns "not a number"
        ws.Cells(r, 8) = CStr(Abs(V1 - V2))      voltage difference across switch
    End If

    i = swt.Next                         move to next SwtControl, increment sheet row number
    r = r + 1
Wend
```

Figure 4-1 lists three switches open as they are in the base case. Switch 10_14 has the largest voltage difference magnitude and is the first candidate for branch exchange.

SwtControl	Element	Terminal	Lock	State	V1	V2	Vdiff
1_4	line.1_4	1	FALSE	2			
4_5	line.4_5	1	FALSE	2			
4_6	line.4_6	1	FALSE	2			
6_7	line.6_7	1	FALSE	2			
2_8	line.2_8	1	FALSE	2			
8_9	line.8_9	1	FALSE	2			
8_10	line.8_10	1	FALSE	2			
9_11	line.9_11	1	FALSE	2			
9_12	line.9_12	1	FALSE	2			
3_13	line.3_13	1	FALSE	2			
13_14	line.13_14	1	FALSE	2			
13_15	line.13_15	1	FALSE	2			
15_16	line.15_16	1	FALSE	2			
5_11	line.5_11	1	FALSE	1	13100.4	12881.1	219.3
10_14	line.10_14	1	FALSE	1	12960.2	13193.0	232.8
7_16	line.7_16	1	FALSE	1	13062.6	13145.9	83.4

Figure 4-1
SwtControl Interface Test Outputs

Figure 4-2 presents output from a simple test of the new *Topology* interface. The code for this test follows; it is the second half of "SwitchingSummary." Isolated loads can be created by opening switches, and loops can be created by closing switches. After making such changes to the base model, you can run the macro again to see output for loops and isolated loads in Figure 4-2. The last segment of this listing shows the possible benefit of numerical indexing. When finding the forward and backward branches, it is necessary to reset the active branch by name instead of a number or handle. Internally, that involves a linear search of the tree by OpenDSS.

In Figure 4-2, the forward trace listing is in the same order as the numerical index. In following a forward trace, it is possible to tell when a zone end point has been reached because the next branch will have a lower or equal topological level. The backward trace listing always moves one level back toward the main voltage source, which itself has no backward branch. In this example, every branch has a *SwtControl* except the artificial bus-splitting branches f1, f2, and f3.

```
    r = topo.NumLoops         write number of loops to the sheet
    ws.Cells(2, 19) = CStr(r)
    If r > 0 Then             for all loops in the topology, write branch names
        an = topo.AllLoopedPairs      an is a variant array of strings (names)
        For i = 1 To r
            s = an(2 * i - 2) first name in a pair is "from", write on row 2
            ws.Cells(2, 19 + i) = s
            s = an(2 * i - 1) second name in a pair is "to", write on row 3
            ws.Cells(3, 19 + i) = s
        Next i
    End If

    r = topo.NumIsolatedLoads write number of isolated loads
    ws.Cells(4, 19) = CStr(r)
    If r > 0 Then      if any isolated loads actually exist
        an = topo.AllIsolatedLoads    "an" is now a variant array of load names
        For i = 1 To r
            s = an(i - 1)             write each load name to the sheet
            ws.Cells(4, 19 + i) = s
        Next i
    End If

    r = topo.NumIsolatedBranches      number of isolated PD branches written to sheet
    ws.Cells(5, 19) = CStr(r)
    If r > 0 Then      if any isolated PD branches actually exist
        an = topo.AllIsolatedBranches "an" is now a variant array of branch names
        For i = 1 To r
            s = an(i - 1)             write each branch name to the sheet
            ws.Cells(5, 19 + i) = s
        Next i
    End If

    i = topo.First     start at the tree's root, to illustrate various traces
    r = 8              tabular output starts on row 8, below the loop/isolated summary
    While i > 0
        Set elem = ckt.ActiveCktElement       ckt element corresponding to topo active branch
        ws.Cells(r, 18) = topo.BranchName     write the branch name and level to the sheet
        ws.Cells(r, 19) = CStr(topo.ActiveLevel)
        ws.Cells(r, 25) = CStr(elem.HasSwitchControl)       tells us if this can be ADA switched

        If topo.LoopedBranch = 1 Then if looped here, write the looped branch name
            ws.Cells(r, 23) = elem.DisplayName
        Else
            ws.Cells(r, 23) = ""
        End If
        If topo.ParallelBranch = 1 Then      if paralleled here, write the directly parallel name
            ws.Cells(r, 24) = elem.DisplayName
        Else
            ws.Cells(r, 24) = ""
        End If

        If topo.FirstLoad > 0 Then ws.Cells(r, 22) = elem.DisplayName name of first load here

        i = topo.Next
        r = r + 1
    Wend

    ' go back and set each topo node by name, find the forward and backward items
    r = r - 1
    For i = 8 To r     loop over the topo rows just written
        s = ws.Cells(i, 18)   read the branch name from sheet
        ws.Cells(i, 20) = ""
        ws.Cells(i, 21) = ""
        topo.BranchName = s   set the topo active branch by name
        If topo.ForwardBranch > 0 Then ws.Cells(i, 20) = topo.BranchName   forward branch?
        topo.BranchName = s   step back to the active branch for this row
        If topo.BackwardBranch > 0 Then ws.Cells(i, 21) = topo.BranchName  backward branch?
    Next i
End Sub
```

Topology Test							
N loops	0.00						
Iso loads	0.00						
Iso branch	3.00	Line.5_11	Line.10_14	Line.7_16			
Active	**Level**	**Forward**	**Backward**	**First Load**	**Loop**	**Parallel**	**Switch?**
Vsource.source	0	Line.f3					FALSE
Line.f3	1	Line.3_13	Vsource.source				FALSE
Line.3_13	2	Line.13_15	Line.f3	Load_13			TRUE
Line.13_15	3	Line.15_16	Line.3_13	Load_15			TRUE
Line.15_16	4	Line.13_14	Line.13_15	Load_16			TRUE
Line.13_14	3	Line.f2	Line.3_13	Load_14			TRUE
Line.f2	1	Line.2_8	Vsource.source	Load_6			FALSE
Line.2_8	2	Line.8_10	Line.f2	Load_8			TRUE
Line.8_10	3	Line.8_9	Line.2_8	Load_10			TRUE
Line.8_9	3	Line.9_12	Line.2_8	Load_9			TRUE
Line.9_12	4	Line.9_11	Line.8_9	Load_12			TRUE
Line.9_11	4	Line.f1	Line.8_9	Load_11			TRUE
Line.f1	1	Line.1_4	Vsource.source	Load_8			FALSE
Line.1_4	2	Line.4_6	Line.f1	Load_4			TRUE
Line.4_6	3	Line.6_7	Line.1_4	Load_6			TRUE
Line.6_7	4	Line.4_5	Line.4_6	Load_7			TRUE
Line.4_5	3		Line.1_4	Load_5			TRUE

Figure 4-2
Topology Interface Test Outputs

Figure 4-3 presents output from the branch exchange algorithm, implemented for loss minimization. The algorithm finds the open switch with the greatest magnitude of voltage difference across the open contacts. Then, starting from the side with lowest voltage, it looks for the next closed switch in a path back to the source. It is assumed that swapping the closed/open state of the switch pair should tend to equalize the voltage difference across the open switches and, thereby, tend to reduce losses. If all the open switches had zero volts across them, the system losses should be the same as if completely meshed, which is known to be the minimum-loss configuration.

In Civanlar et al. [3], refinements are described that involve summing resistance values over the candidate switching paths. This refinement could be implemented using the Forward and Backward methods of the *Topology* interface, identifying which traced elements are actually lines and then accumulating R1 attribute values.

The exchange algorithm is “greedy” and will always try to make a switch exchange. The main stopping criterion is when the losses increase after an exchange, meaning that the algorithm has found and passed through a local minimum. The exchange method should never create a loop or isolate a load. If either occurs, the loop halts under what might be considered an error condition.

In Figure 4-3, note the minimum loss condition was found at iteration 3, with the three switches to open as listed. To prepare a switching plan, the exchanged pair could have been written for each step. The system stayed in a radial configuration, and no branches were isolated in each

step. This basic algorithm can be improved in many ways; these improvements are the subject of many graduate student theses in power systems. This example serves as a starting point for using OpenDSS in academic projects, with access to realistic feeder models already built for the simulator.

```
Public Sub BranchExchange()
    Dim iter, c, r, i, k As Integer
    Dim done As Boolean
    Dim Vdiff, Vmax As Double
    Dim LastLoss, ThisLoss As Double
    Dim ToClose, ToOpen, LowBus As String

    Preamble
    Set ws = ActiveWorkbook.Worksheets("Switching")

    iter = 1   this is the number of branch exchange trials, limited to 10
    done = False
    LastLoss = 1E+99
    While Not done
        r = iter + 1
        ws.Cells(r, 10) = iter
        ckt.Solution.Solve    solve the current system
        ThisLoss = ckt.Losses(0)
        ws.Cells(r, 11) = ThisLoss    write current losses, # loops, # isolated loads to sheet
        ws.Cells(r, 12) = CStr(ckt.Topology.NumLoops) & " _ " & _
                          CStr(ckt.Topology.NumIsolatedLoads)

        Vmax = 0#      track the maximum voltage difference across any open switch
        ToClose = ""
        ToOpen = ""
        LowBus = ""
        c = 14 column number for output
        i = swt.First  check all SwtControls
        While i > 0 ' find the open switch with biggest delta-V
            If swt.Action = dssActionOpen Then       check only open switches
                ws.Cells(r, c) = swt.name
                Set elem = ckt.CktElements(swt.SwitchedObj)
                Vdiff = Abs(elem.SeqVoltages(1) - elem.SeqVoltages(4))     V1 across switch
                If Vdiff > Vmax Then  if highest V1 difference so far…
                    LowBus = FindLowBus      which side of open switch has lowest V?
                    topo.BusName = LowBus    start from that bus in the topology
                    Set elem = ckt.ActiveCktElement
                    k = 1
                    While (Not elem.HasSwitchControl) And (k > 0) trace back from low bus to src
                        k = topo.BackwardBranch                    until we find a closed switch
                    Wend
                    If elem.HasSwitchControl Then    if we found a switch to close…
                        Vmax = Vdiff         keep this as the highest voltage difference found
                        ToClose = swt.name   we will close this currently-open switch
                        ToOpen = Mid(elem.Controller, 12)   and open the switch from back-trace
                    End If
                End If
                c = c + 1
            End If
            i = swt.Next
        Wend
        ws.Cells(r, 13) = CStr(Vmax)

        done = True ' unless we found a switch pair to exchange
        If Len(ToOpen) > 0 And Len(ToClose) > 0 Then        found a switch pair to exchange
            swt.name = ToClose               do the switch close-open via SwtControl interface
            swt.Action = dssActionClose
            swt.name = ToOpen
            swt.Action = dssActionOpen
            done = False ' try again  i.e., run solution again and look for the next exchange
        End If
```

```
iter = r
        ' stop if too many iterations, system is non-radial, or losses go up
        If iter > 10 Or ckt.Topology.NumIsolatedLoads > 0 Or ThisLoss > LastLoss Then
            done = True       met one of the three stopping criteria
        End If
        LastLoss = ThisLoss   best loss total found so far
    Wend
End Sub

This function is called from the loop above
"elem" is a CktElement already set to the open branch, from the loop calling FindLowBus
Private Function FindLowBus() As String
    Dim i As Integer
    Dim v As Double

    FindLowBus = ""
    v = 9.9E+100
    For i = 0 To elem.NumTerminals - 1       loop over all element terminals
        If elem.BusNames(i) <> "0" Then
            Set b = ckt.Buses(elem.BusNames(i))      look up the Bus connected to terminal
            If b.SeqVoltages(1) < v Then     track the lowest bus positive sequence voltage
                v = b.SeqVoltages(1)
                FindLowBus = elem.BusNames(i)       return name of bus with lowest V1
            End If
        End If
    Next i
End Function
```

Iteration	Losses	Loops/Iso	Max Vdiff	Open Switches		
1	488945.67	0 _ 0	232.77	5_11	10_14	7_16
2	464241.59	0 _ 0	191.21	8_10	5_11	7_16
3	447937.91	0 _ 0	188.76	8_10	9_11	7_16
4	657613.59	0 _ 0	475.18	8_9	8_10	7_16

Figure 4-3
Branch Exchange Iterations Using the Topology Interface

Substation Volt/VAR Example

This example illustrates the *RegControl* and *DSSEvents* interfaces, using the circuit shown in Figure 2-2. The base model is in *Regulator.dss*, a load duration curve in *LDC_2007.dat*, and VBA code in *Events.xlsm*. The base model includes a *CapControl* to manage the substation capacitor bank under reactive power control. The VBA code reads and sets the *RegControl* R and X from a worksheet and does not change them. Those elements are not covered in detail here.

The VBA event handling code, which is the main focus of this example, follows. First, one of the main VBA code modules, called Module1 by default, must have a global declaration of the event handling class and initialize after starting the DSS engine. The following code lines (not shown in context) handle these steps. These are the only steps needed to "hook up" the event handler to OpenDSS.

```
Dim Evt As New DSSEventHandler

Set eng = CreateObject("OpenDSSEngine.DSS")
eng.start (0)
Set Evt.Evt = eng.Events      note: Evt subscribes to eng.Events, not the other way around
```

A second VBA "class module," called DSSEventHandler, must also appear in the VBA project. In the development sequence, this class module must actually be written before it is referenced. The first line of the class module code must be written as shown, using the keywords

"WithEvents" and "As DSSEvents." Once that has been done, the combo boxes in Excel's VBA editor will be able to insert code templates for the three event-handling procedures. In this example, helper functions WriteTrace, WriteOutput, and SetControls do the actual work. iControl is a flag set from the main VBA code to enable or disable the controller. In Evt_InitControls, it is important to set the *Circuit* interface or any other helper variables. When the main VBA code invokes ckt.Solution.Solve, the OpenDSS solution loop will call Evt_InitControls before the call to ckt.Solution.Solve returns. In the Preamble code for all these examples, the ckt interface variable is not set until ckt.Solution.Solve returns. Therefore, it needs to be set in the InitControls handler as well.

Basically, the controls should be set whenever Evt_InitControls or Evt_CheckControls is called from OpenDSS. Save some output for plotting whenever Evt_StepControls is called, since the solution has converged. If iTrace is positive, also log each call to the event handlers.

```
Public WithEvents Evt As DSSEvents

Private Sub Evt_CheckControls()
    WriteTrace ("Check")
    If iControl > 0 Then Call SetControls
End Sub

Private Sub Evt_InitControls()
    Set ckt = eng.ActiveCircuit      is typically called before ckt.Solution.Solve returns
    WriteTrace ("Init")
    If iControl > 0 Then Call SetControls
    Call WriteOutput
End Sub

Private Sub Evt_StepControls()
    WriteTrace ("Step")
    Call WriteOutput
End Sub
```

The above code provides the complete template for a class handling *DSSEvents*. The actual work for this example is done in three helper functions.

In WriteTrace, a debug log of event type, solution time, and solution mode is written onto a worksheet accessed through the wsTrace variable. The variable iTrace serves as a flag and row index for the log. Examination of this log shows that the first solution at light load, about 30 MW, required 12 control iterations. The capacitor bank ratchets down one step at a time, according to its reactive power control. The next solution, at about 99 MW, required four control iterations. The remaining solutions required only one control iteration, except for one solution that used two control iterations.

```
Private Sub WriteTrace(Label As String)
    If iTrace > 0 Then this merely writes a debugging trace output onto one of the worksheets
        wsTrace.Cells(iTrace, 1) = Label
        wsTrace.Cells(iTrace, 2) = CStr(ckt.Solution.dblHour)
        wsTrace.Cells(iTrace, 3) = CStr(ckt.Solution.Mode)
        iTrace = iTrace + 1
    End If
End Sub
```

WriteOutput places numerical values onto a separate sheet for plotting. The variable iPlot serves as a flag and row index for these outputs. Only the first capacitor, line, and transformer are important because the circuit is very simple. The ForwardVreg output is especially important in order to verify that the controller sets the regulator in each time step to match the actual load.

```
Private Sub WriteOutput()
    Dim elem As CktElement, i As Integer
    If iPlot > 0 Then  this writes numerical output onto a sheet for plotting
        wsPlot.Cells(iPlot, 1) = CStr(ckt.Solution.dblHour)
        i = ckt.Capacitors.First      first capacitor is the substation bank
        Set elem = ckt.ActiveCktElement
        wsPlot.Cells(iPlot, 2) = CStr(0.001 * elem.SeqPowers(3)) ' Q cap
        i = ckt.Lines.First   first line is the primary feeder equivalent
        Set elem = ckt.ActiveCktElement
        wsPlot.Cells(iPlot, 3) = CStr(0.001 * elem.SeqPowers(2)) ' P feeder
        wsPlot.Cells(iPlot, 4) = CStr(0.001 * elem.SeqPowers(3)) ' Q feeder
        i = ckt.Transformers.First    first transformer is the substation equivalent
        wsPlot.Cells(iPlot, 5) = CStr(ckt.Transformers.Tap) ' Tap
        i = ckt.RegControls.First     only one RegControl in this example
        wsPlot.Cells(iPlot, 6) = CStr(ckt.RegControls.ForwardVreg) ' Vset
        i = ckt.Loads.First           only one Load in this example
        Set elem = ckt.ActiveCktElement
        wsPlot.Cells(iPlot, 7) = CStr(elem.SeqVoltages(1)) ' Vload
        wsPlot.Cells(iPlot, 8) = CStr(0.001 * elem.SeqPowers(2)) ' Pload
        wsPlot.Cells(iPlot, 9) = CStr(0.001 * elem.SeqPowers(3)) ' Qload
        iPlot = iPlot + 1     increment the row counter
    End If
End Sub
```

The SetControls function will be called at the beginning of the time step loop and after each voltage and current solution has converged. The control quantity is the power into the 13.8-kV substation bus. In SetControls, this is the magnitude of positive sequence power in terminal 2 of the substation transformer. Since power flows out of this terminal, SeqPowers(8) is negative, and the absolute value is taken. In WriteOutputs above, the same value is obtained from positive sequence power in terminal 1 of the feeder line. A sequence of If statements implements the lookup table as displayed in Figure 2-2, and the looked-up value is finally applied to the first RegControl.

Within OpenDSS, that assignment to ForwardVreg occurs before the internal control check is done. If the new ForwardVreg value causes a tap change, then OpenDSS does a new control iteration. If there is no tap change, possibly because ForwardVreg has the same value as before, then this RegControl will not force another control iteration. However, the CapControl could independently force a control iteration, as happened many times in the initial solution.

```
Private Sub SetControls()
    Dim elem As CktElement, i As Integer, p As Double, vset As Double
    i = ckt.Transformers.First        the first transformer is in the substation
    ' ckt.Transformers.name = "sub"   another way of selecting the transformer by name
    Set elem = ckt.ActiveCktElement
    p = Abs(elem.SeqPowers(8) * 0.001)       positive sequence power [MW] through terminal 2
    vset = 110.83                     implement a lookup table for Vset vs. P
    If p > 50# Then vset = 111.67
    If p > 100# Then vset = 112.5
    If p > 150# Then vset = 113.33
    If p > 200# Then vset = 114.17
    If p > 250# Then vset = 115#
    If p > 300# Then vset = 115.83
    i = ckt.RegControls.First         select the first RegControl, and update ForwardVreg
    ckt.RegControls.ForwardVreg = vset
End Sub
```

As implemented, this example does not use the *CtrlQueue* COM interface, which effectively means that its ForwardVreg assignments preempt any other control action. However, the inherent *RegControl* time delay still applies, so it coordinates properly with the *CapControl*. There could be other applications where *CtrlQueue* is necessary for time coordination.

Figure 4-4 shows the regulator set point versus feeder power over the simulated load duration curve. The voltage setting properly tracks the lookup table. A tap change may occur either because the voltage setting changed or by action of the line drop compensator R and X settings. Figure 4-5 shows that one or three taps are actually used at each set point.

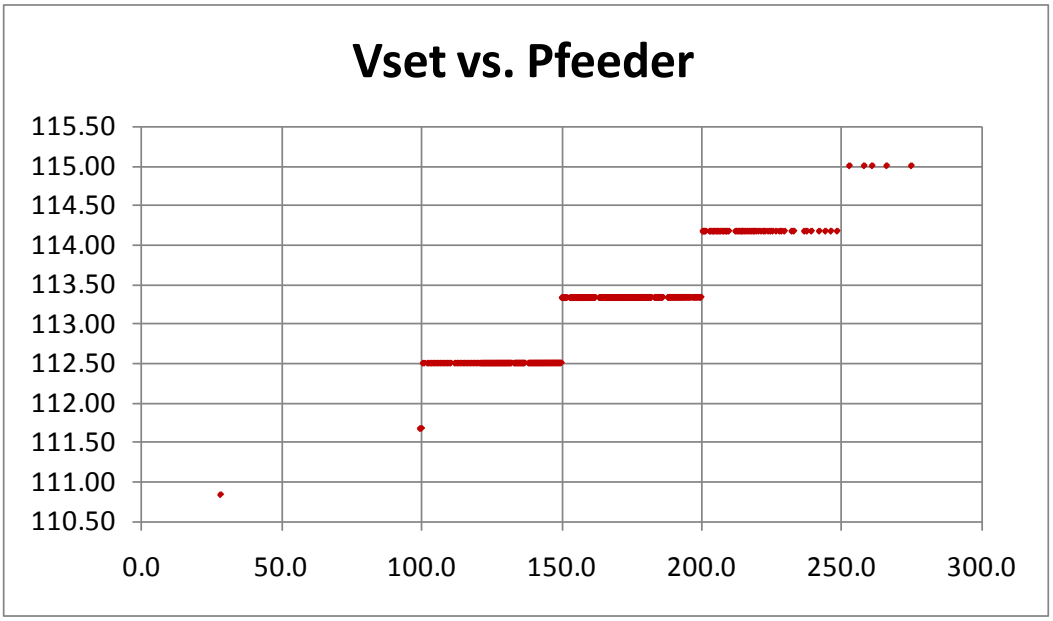

Figure 4-4
Regulator Set Point vs. Feeder Power

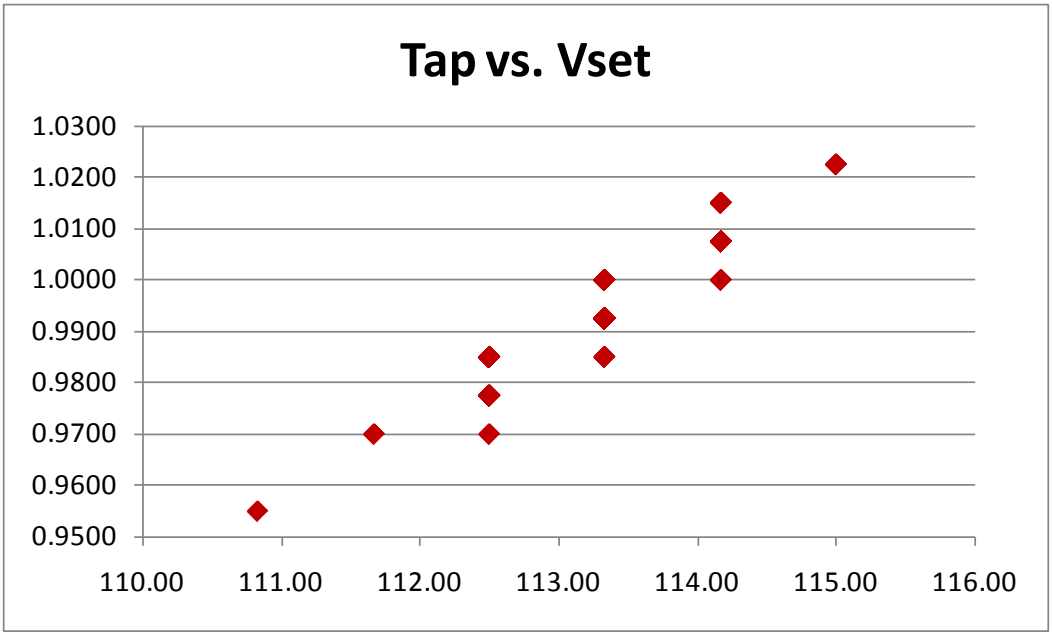

Figure 4-5
Tap vs. Regulator Set Point

Figure 4-6 shows the substation capacitor bank's reactive power versus feeder power. For a load power factor of 0.88, the *CapControl* switches 200 MVAR in 10 steps to approximately compensate the load, using the reactive power control mode. There is some linear variation of Q vs. P, due to changes in bus voltage at the substation.

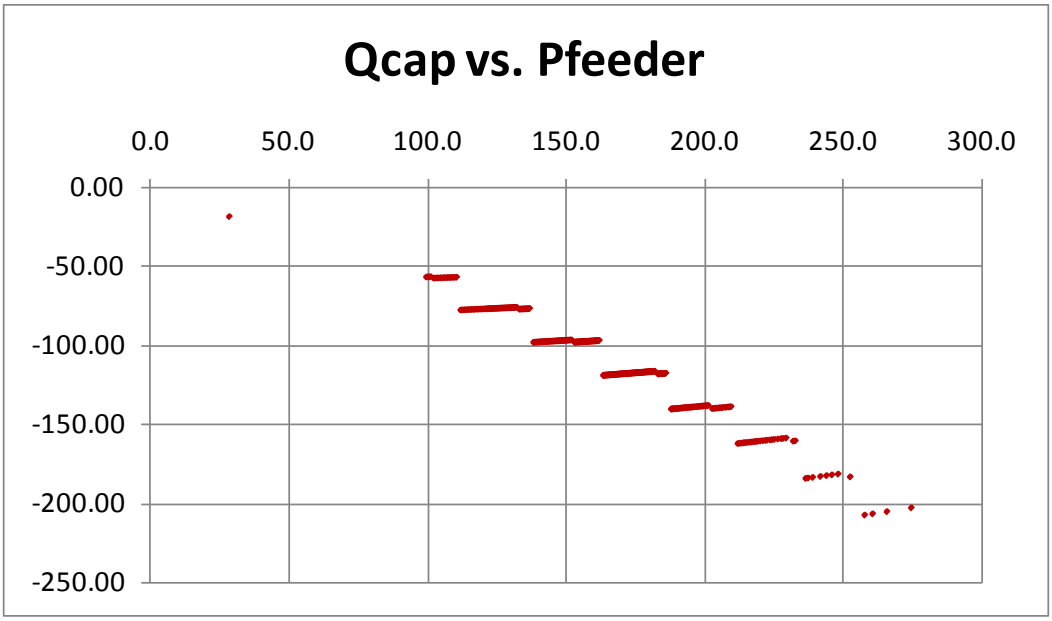

Figure 4-6
Substation Capacitor MVAR vs. Feeder Power

Figure 4-7 shows the load voltage versus feeder power on a 120-volt base. There is a different linear segment for each tap change and a slope due to voltage drop along the feeder impedance.

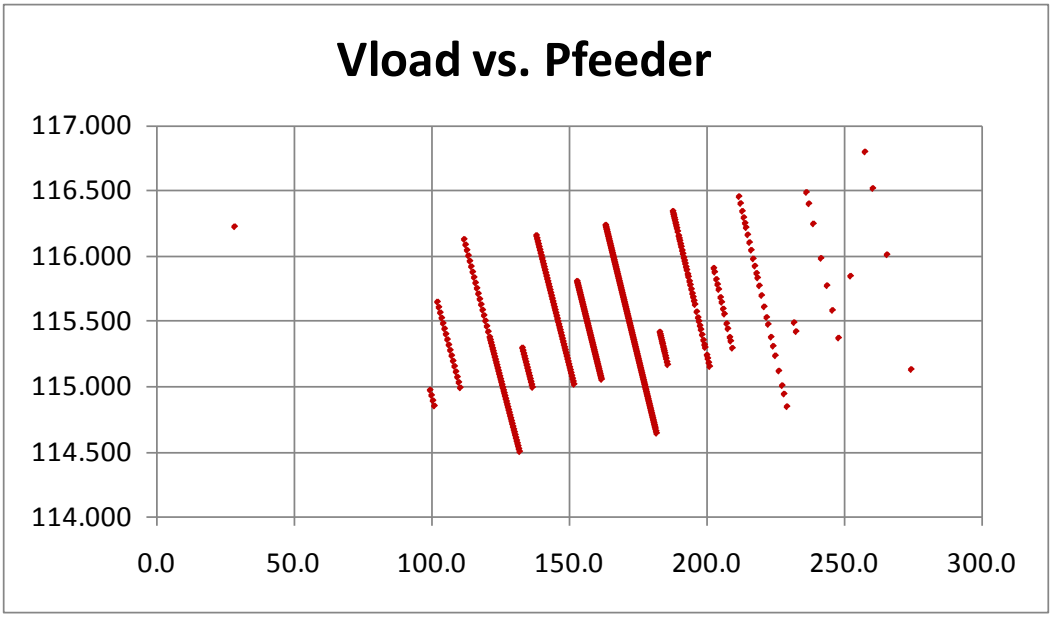

Figure 4-7
Load Voltage vs. Feeder Power

Feeder Volt/VAR Example

This example uses the two-feeder system in Figure 2-3 to illustrate the *CapControl*, *RegControl*, and *Monitor* interfaces. The base model is in *Southern_Co.dss*, which includes other files, and the VBA code is in *DSS_Reports.xls*. The function VoltVarSummary is listed below. It simulates regulator voltage reduction by remote control. At least two regulator vendors offer this feature. Typically, there are three settings that may be programmed from 0% to 10% voltage reduction. The settings are activated by local control from a panel or by pulse signals to terminals on the regulator.

Figure 4-8 shows the resulting *CapControl* list. The two banks with the "use" flag set to 1, c45102 and 45103, have SCADA control. The other three banks have local control only. Figure 4-9 shows the resulting *RegControl* list; only those with "Volt Red" set to 1 participate in the automatic voltage reduction. Figure 4-10 summarizes the total power, losses, and voltage range for a snapshot load flow. With no voltage reduction (that is, all regulators fixed at 125 volts), the load increases to 6565.39 kW with 3.94% losses and nearly the same voltage range. Therefore, this voltage reduction has reduced the load by 2.9%. Actual response will depend on the load conservation voltage reduction (CVR) factors. The various **helper functions** called from VoltVarSummary are documented below.

```
Public Sub VoltVarSummary()
    Dim i, r As Integer
    Dim Vmin As Double, Vmax As Double, Pload As Double

    Preamble   start DSS engine and load the base system model, set interface variables

    Set ws = ActiveWorkbook.Worksheets("VoltVar")

    ' set the solution options from the spreadsheet
    ckt.Solution.LoadMult = CDbl(ws.Cells(8, 18))    adjustable load scaling
    SetVoltageReduction       selected RegControls placed on voltage reduction

    ckt.Solution.Solve        re-solve the circuit with voltage reduction

    i = cap.First             list all CapControl bank sizes and control modes
    r = 2
    While i > 0
        ws.Cells(r, 1) = cap.name
        ckt.Capacitors.name = cap.Capacitor
        ws.Cells(r, 2) = ckt.Capacitors.kvar
        ws.Cells(r, 3) = cap.Mode

        i = cap.Next
        r = r + 1
    Wend

    i = reg.First      list all RegControl set points
    r = 2
    While i > 0
        ws.Cells(r, 6) = reg.name
        ws.Cells(r, 7) = reg.ForwardVreg
        i = reg.Next
        r = r + 1
    Wend

    i = mon.First      list all Monitors; these implement line post sensors
    r = 2
    While i > 0
        ws.Cells(r, 10) = mon.name
        ws.Cells(r, 11) = mon.Mode
        i = mon.Next
        r = r + 1
    Wend

    i = mtr.First      list all EnergyMeters
    r = 2
    While i > 0
        ws.Cells(r, 14) = mtr.name
        i = mtr.Next
        r = r + 1
    Wend

    Pload = GetLoadPower
    ws.Cells(1, 18) = Format(0.001 * ckt.Losses(0), "0.000")
    ws.Cells(2, 18) = Format(0.001 * ckt.Losses(1), "0.000")
    Call GetExtremeVoltages(Vmin, Vmax)
    ws.Cells(3, 18) = Format(Vmin, "0.0000")
    ws.Cells(4, 18) = Format(Vmax, "0.0000")
    ws.Cells(5, 18) = Format(Pload, "0.000")

    mon.SampleAll
    mon.SaveAll
    ShowMonitors 11, 17
End Sub
```

CapControl	CapSize	Mode	Use
c81	600	1	0
c45102	600	1	1
c80	300	1	0
c79	600	1	0
45103	900	1	1

Figure 4-8
Feeder Capacitor Banks for Volt/VAR Control

RegControl	Vreg	Volt Red
85b	118.0	1
85c	118.0	1
85a	118.0	1
r88a	125.0	0
r88c	125.0	0
r88b	125.0	0
156b	125.0	0
156a	125.0	0
156c	125.0	0
w155c	125.0	0
w155a	125.0	0
w155b	125.0	0

Figure 4-9
Regulators and Autoboosters for Volt/VAR Control

Base Loss P [kW]	257.981
Base Loss Q [kVAR]	601.51
Base Vmin	0.9112
Base Vmax	1.1218
Base Load P [kW]	6374.60
Loss % of Total	3.89%
Reduce Voltage	**118.00**
Load Multiplier	**0.30**

Figure 4-10
Losses and Voltage Limits with Volt/Var Control

The SetVoltageReduction function reads new regulator set points from the worksheet. If the "Use" flag has been set to 1, those values are written to the *RegControl* objects.

```
Private Sub SetVoltageReduction()
    Dim Vnew As Double
    Dim r As Integer, use As Integer, name As String

    Vnew = CDbl(ws.Cells(7, 18))      reads the AVR set-point from highlighted cell in Fig 4-10
    r = 2
    name = ws.Cells(r, 6)
    While Len(name) > 0       read down the RegControl list in Fig 4-9
        use = CInt(ws.Cells(r, 8))
        If use > 0 Then       if this RegControl uses AVR, change its ForwardVreg value
            reg.name = name
            reg.ForwardVreg = Vnew
        End If

        r = r + 1
        name = ws.Cells(r, 6)
    Wend

End Sub
```

The GetLoadPower function enumerates over all power conversion elements in the circuit and accumulates their power values. The load terminal powers come through the COM interface as a variant array, alternating P and Q values. Because this model has a mixture of single-phase and three-phase loads, the variant array size is not known in advance. If the model includes generators, this function should be modified to either exclude them or output the total generated power separately.

```
Private Function GetLoadPower() As Double
    Dim i As Integer, j As Integer, n As Integer, ap As Variant
    Dim p As Double

    p = 0#     accumulate power in this variable
    i = ckt.FirstPCElement
    While i > 0        loop over all power conversion elements
        ap = ckt.ActiveCktElement.Powers     ap is a variant array of doubles
        n = ckt.ActiveCktElement.NumPhases
        For j = 0 To n
            p = p + ap(2 * j) select real power from even indices; reactive power in odd indices
        Next j
        i = ckt.NextPCElement
    Wend
    GetLoadPower = p
End Function
```

The GetExtremeVoltages function enumerates over all buses in the circuit, obtaining a variant array of their terminal voltages. This variant array alternates real and imaginary parts of the terminal voltages, and the array size depends on the number of phases actually present at the bus. This function returns two values, Vmin and Vmax, as a VBA subroutine. These values can serve as constraints on simulated voltage reduction.

```
Private Sub GetExtremeVoltages(Vmin As Double, Vmax As Double)
    Dim i As Integer, ni As Integer, j As Integer, nj As Integer
    Dim v As Double, vr As Double, vi As Double, av As Variant

    Vmin = 1E+99       retain Vmin and Vmax over all bus terminals
    Vmax = 0#
    ni = ckt.NumBuses - 1
    For i = 0 To ni
        Set b = ckt.Buses(i)  a Bus has ready access to puVoltages
        nj = b.NumNodes - 1
        av = b.puVoltages
        For j = 0 To nj       loop over all Bus nodes (terminals)
            vr = av(2 * j)    real part of voltage from even indices
            vi = av(2 * j + 1)       imaginary part of voltage from odd indices
            v = Sqr(vr * vr + vi * vi)       magnitude of this node voltage
            If v < Vmin And v > 0# Then Vmin = v     track Vmin and Vmax over all nodes
            If v > Vmax Then Vmax = v
        Next j
    Next i
End Sub
```

The ShowMonitors function enumerates over Monitors in the circuit, and outputs their solved values. In this example, each line post sensor encapsulates two Monitors in the model. One measures P and Q by phase, while the other measures V and I by phase. The Monitor name consists of the line post sensor name, followed by _s for a PQ monitor or _vi for a VI monitor. OpenDSS provides Monitor data in binary form, which VBA can handle with Variant and Byte variables.

A Windows kernel function called RtlMoveMemory copies binary data from COM into the VBA variables. This function must be declared as follows, before using it in VBA.

```
Private Declare Sub CopyMemory Lib "KERNEL32" Alias "RtlMoveMemory" (hpvDest As Any, _
    hpvSource As Any, ByVal cbCopy As Long)
```

The ShowMonitors function code is a recipe for extracting binary Monitor data in VBA. Figure 4-11 shows the resulting list of Monitor names and values. This function provides one way of extracting sensor values for use in ADA simulations, when the sensors are represented with Monitors in OpenDSS.

```
Private Sub ShowMonitors(row As Integer, col As Integer)
    Dim i As Integer, j As Integer, count As Integer, k As Integer
    Dim str As Variant, ba() As Byte
    Dim mSig As Long, mVer As Long, mRec As Long, mMode As Long, mCond As Long
    Dim mHour As Single, mSec As Single
    Dim mRe As Single, mIm As Single
    Dim idx As Long, samp As Long, baLen As Long, reqLen As Long

    i = mon.First
    While i > 0       loop over all monitors
        ws.Cells(row, col) = mon.name

        ' read the monitor
        count = mon.SampleCount
        str = mon.ByteStream
        ba = str

        CopyMemory mSig, ba(0), 4
        CopyMemory mVer, ba(4), 4
        CopyMemory mRec, ba(8), 4
        CopyMemory mMode, ba(12), 4

        ' this should hold 16 byte header (above), 256-byte buffer, and
        ' then 8 (for time) + 4 * mRec * SampleCount
```

```
        baLen = UBound(ba)
        reqLen = 16 + 256 + 8 + 4 * mRec - 1

        If reqLen > baLen Then GoTo Skip
        idx = 16 + 256
        For j = 1 To count
            CopyMemory mHour, ba(idx), 4
            CopyMemory mSec, ba(idx + 4), 4
            idx = idx + 8

            For k = 1 To mRec / 2
                CopyMemory mRe, ba(idx), 4
                CopyMemory mIm, ba(idx + 4), 4
                If mMode = 0 Then     mode 0 is polar format, we want magnitude only
                    ws.Cells(row, col + k) = CStr(mRe)
                Else                  mode 1 is rectangular format, we output both (P and Q)
                    ws.Cells(row, col + k) = CStr(mRe)
                    ws.Cells(row, col + mRec / 2 + k) = CStr(mIm)
                End If
                idx = idx + 8
            Next k
        Next j
Skip:
        i = mon.Next
        row = row + 1
    Wend
End Sub
```

Monitor Values	V and I Magnitude, or P and Q					
fdr2_vi	7289.17	7444.59	7463.69	81.17	37.85	33.10
fdr2_s	-580.05	-275.36	-225.97	-116.50	59.92	99.75
fdr6_vi	7288.76	7444.38	7463.47	378.36	207.44	206.27
fdr6_s	-2511.70	-1518.34	-1520.98	-1138.64	-281.95	-237.84
c45102_vi	7265.96	7452.00	7464.91	90.56	42.35	32.08
c45102_s	658.00	315.57	239.46	28.80	29.18	26.95
c45103_vi	6670.08	7480.87	7334.20	312.21	160.84	159.76
c45103_s	2082.44	1203.20	1171.70	29.64	25.68	25.24
x7168_vi	7245.67	7452.25	7461.29	85.42	42.09	27.68
x7168_s	618.90	313.67	206.56	28.74	29.25	26.89
xc941_vi	7092.26	7419.37	7455.06	45.70	40.06	10.13
xc941_s	324.15	297.23	75.50	27.90	28.87	26.00
q3495_vi	7249.38	7452.73	7460.11	0.00	0.00	0.00
q3495_s	0.01	0.01	0.01	-97.07	-87.50	-85.93
q3180_vi	7147.31	7445.78	7433.17	38.37	30.59	12.65
q3180_s	274.27	227.75	94.04	28.47	28.02	27.66
g3373_vi	6938.95	7458.80	7385.36	335.96	175.12	181.73
g3373_s	2331.24	1306.18	1342.17	21.64	5.57	5.71
fay45404_vi	6886.74	7465.50	7429.09	127.68	76.81	59.22
fay45404_s	879.26	573.42	439.98	24.08	18.87	15.88

Figure 4-11
Monitor Names and Sample Values

5
REFERENCES

1. OpenDSS [Online]. Available: http://sourceforge.net/projects/electricdss/.
2. *Control System Simulation in the Distribution System Simulator (DSS)*. EPRI, Palo Alto, CA: 2009. 1016035.
3. S. Civanlar, J. J. Grainger, Y. Yin, S. S. Lee, "Distribution Feeder Reconfiguration for Loss Reduction," *IEEE Trans. on Power Delivery*, Vol. 3, no. 3. July 1988, pp. 1217–1223.
4. *Program on Technology Innovation: Distribution Common Information Model (CIM) Modeling of Two North American Feeders*. EPRI, Palo Alto, CA: 2009. 1018281.

A
GETTING STARTED WITH THE COM INTERFACE

The Microsoft® Windows Component Object Model (COM) interface provides a convenient method for automating, or controlling, one program from another. COM is exploited in OpenDSS for a number of purposes. For example, there is no looping in the scripting language. Users who wish to write custom algorithms that require looping are expected to control the looping in another program through the COM interface.

COM may be implemented in basically two forms:

1. An *in-process server*, in which the server is implemented in a DLL and becomes part of the process memory space when it is loaded.
2. An *out-of-process server*, in which the interface is on a separate program that runs in a different process space. An example of this is automating Word or Excel programs.

The OpenDSS COM server is currently implemented only as a 32-bit in-process server. This is expected to change in the future as the software is updated to support more platforms. This server is registered as OpenDSS*Engine.DSS* when the program is installed. This is the only COM interface that is registered. There are actually many interfaces that make up the OpenDSS COM interface, each with multiple properties, methods, and constants. These are created by the *DSS* interface when it is initialized.

The interfaces can be confusing to new users. Also, as evidenced by this report, there are new features added frequently to meet the needs of particular studies, and the on-line documentation is generally not completely up to date. Users who would like to write custom algorithms are encouraged to acquire a *type library* tool that is capable of exposing the interfaces in a convenient manner. Nearly all software development environments designed for the Windows environment have some sort of type library (TLB) tool. There are also several TLB documentation tools available from the Internet.

While you can drive OpenDSS with many programs and computer languages, one of the easiest to use and most widely available is Microsoft® Office tools with Visual Basic for Applications (VBA). That is why it was chosen for the examples in this report. The Object Browser in VBA does an excellent job of showing you the elements in the COM interface. Also, the code editor will automatically help you develop the code by prompting you with the choices that can be entered at any time.

Figure A-1 shows an example of the display that the Object Browser in VBA shows the programmer. Even when using other programming tools, such as MATLAB, it is often advisable to keep the VBA Object Browser visible to assist with the programming.

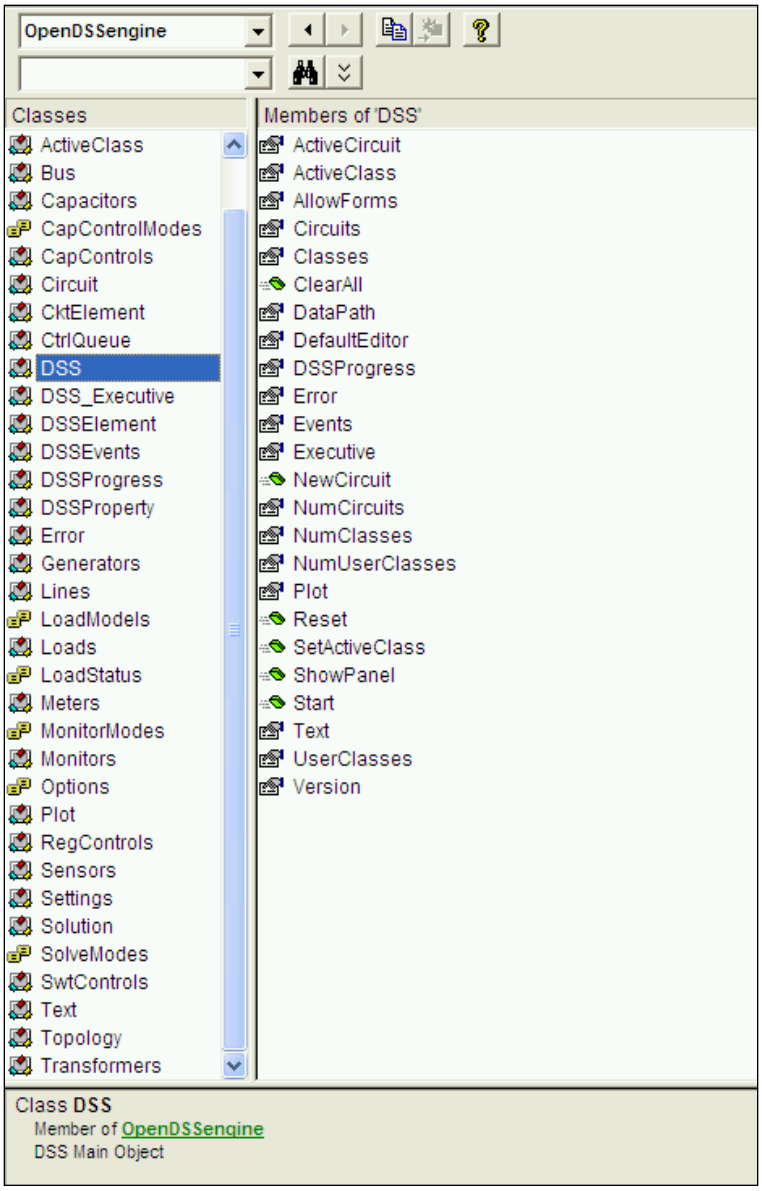

Figure A-1
Screen Capture of OpenDSS Interface as Exposed in Excel VBA

On the right side of Figure A-1 are the members of the main DSS interface. This points to the top level of the other interfaces. It is usually convenient to assign local variables to these interfaces after the program is started and initialized. For example, you might do something like this in VBA:

```
Set DSSobj = New OpenDSSEngine.DSS  ‘ sets to DSS interface
DSSobt.Start(0)

Set DSSText = DSSObj.Text
Set DSSCircuit = DSSObj.ActiveCircuit
Set DSSSolution = DSSCircuit.Solution
```

This example loads the OpenDSS in-process COM server named OpenDSSEngine.DSS and starts it. This creates all the interfaces. Then it assigns three variables to different interfaces for convenience and efficiency in coding.

Nearly all interface classes employ the philosophy of acting on the *active element*, whether that element is the entire circuit or a selected electrical or control element in that circuit. In each class, there are two or more ways of selecting an object in that class, which is subsequently the object that is acted upon by the other properties and methods in that class.

A brief listing of all interface classes and their properties and methods are given in Appendix B. The reader will notice that many of the properties return *Variant* values. Languages like VBA handle this transparently. This enables the user to easily exploit one of the strengths of OpenDSS in which many things are expressed in terms of arrays. For example, one may obtain all the per unit node voltage magnitudes in a variant array with one statement:

```
Dim MyVoltages as Variant

[… other code … ]

MyVoltages = DSSCircuit.AllBusVmagPu
```

The *MyVoltages* variable is then processed as an array of doubles (double-precision floating-point values).

Other examples may be obtained from the Wiki site and the Examples folder of the open source sharing site. See:

http://sourceforge.net/apps/mediawiki/electricdss/index.php?title=OpenDSS_Links

B
OPENDSS COM INTERFACE LISTING

This appendix contains a listing of the OpenDSS COM interface for Version 7.4.1. The descriptions of the Properties and Methods are provided in text, similar to what would be used in a VBA program.

Many of the classes employ a similar scheme for selecting a particular instance of a class:

- The *Name* property can be used to either set or retrieve the name of the active object.
- The *First..Next* properties can be used in a loop to iterate through all objects in a particular class. These properties return 0 when there are no more objects.

For example, this code snippet demonstrates iterating through the *Lines* class collection to extract the R and X matrices for inserting into an Excel worksheet:

```
iRow = 2
' cycle through the Lines elements
i = DSSLines.First   ' sets first Load active
Do While i > 0
     ' Name goes in first column
        WorkingSheet.Cells(iRow, 1).Value = DSSLines.Name
    ' Load the R and X matrixes
        Rmat = DSSLines.Rmatrix
        Xmat = DSSLines.Xmatrix
    ' Put the sum of all conductors at first terminal (only one) into Cells
    ' Use Lbound and UBound because you don't know the actual range
        iCol = 2
        For j = LBound(Rmat) To UBound(Rmat)
            WorkingSheet.Cells(iRow, iCol).Value = Rmat(j)
            WorkingSheet.Cells(iRow, iCol + 1).Value = Xmat(j)
            iCol = iCol + 2
        Next j
       iRow = iRow + 1
   i = DSSLines.Next    ' go to next Line
Loop
```

Some collections classes such as *Loads* also allow you to select object by an integer index, *idx*.

ActiveClass

This class can be used to set a particular circuit element class to be the active class or obtain a pointer to the active OpenDSS class. See also the *SetActiveClass* method in the *Circuit* class.

Properties:

- Property **ActiveClassName** As String [r/o]
- Property **AllNames** As Variant [r/o]
- Property **Count** As Long [r/o]
- Property **First** As Long [r/o]
- Property **Name** As String [r/w]
- Property **Next** As Long [r/o]
- Property **NumElements** As Long [r/o]

Bus

This class operates on the active bus. The bus is selected by either the *ActiveBus, SetActiveBus, or SetActiveBusi* properties and the methods of the *Circuit* class. The x,y coordinates can be set for the active bus. All other properties are read only. **Note:** Bus objects do not exist until either the circuit is solved or the MakeBusList command has been issued.

Properties:

- Property **Coorddefined** As Boolean [r/o]
- Property **CplxSeqVoltages** As Variant [r/o]
- Property **Distance** As Double [r/o]
- Property **Isc** As Variant [r/o]
- Property **kVBase** As Double [r/o]
- Property **Name** As String [r/o]
- Property **Nodes** As Variant [r/o]
- Property **NumNodes** As Long [r/o]
- Property **puVoltages** As Variant [r/o]
- Property **SeqVoltages** As Variant [r/o]
- Property **Voc** As Variant [r/o]
- Property **Voltages** As Variant [r/o]
- Property **x** As Double [r/w]
- Property **y** As Double [r/w]
- Property **YscMatrix** As Variant [r/o]
- Property **Zsc0** As Variant [r/o]
- Property **Zsc1** As Variant [r/o]
- Property **ZscMatrix** As Variant [r/o]

Methods:

- Function **GetUniqueNodeNumber**(ByVal **StartNumber** As Long) As Long
- Function **ZscRefresh**() As Boolean

Capacitors

This is a collection class for *Capacitor* objects in the circuit. Note that only a few of the properties of *Capacitor* objects are exposed through this interface. Use the *Text* interface to set or get other properties using OpenDSS script.

Properties:

- Property **AllNames** As Variant [r/o]
- Property **Count** As Long [r/o]
- Property **First** As Long [r/o]
- Property **IsDelta** As Boolean [r/w]
- Property **kV** As Double [r/w]
- Property **kvar** As Double [r/w]
- Property **Name** As String [r/w]
- Property **Next** As Long [r/o]
- Property **NumSteps** As Long [r/w]

CapControls

This is a collection class for CapControl objects in the circuit.

Properties:

- Property **AllNames** As Variant [r/o]
- Property **Capacitor** As String [r/w]
- Property **Count** As Long [r/o]
- Property **CTratio** As Double [r/w]
- Property **DeadTime** As Double [r/w]
- Property **Delay** As Double [r/w]
- Property **DelayOff** As Double [r/w]
- Property **First** As Long [r/o]
- Property **Mode** As CapControlModes [r/w]
- Property **MonitoredObj** As String [r/w]
- Property **MonitoredTerm** As Long [r/w]
- Property **Name** As String [r/w]
- Property **Next** As Long [r/o]
- Property **OFFSetting** As Double [r/w]
- Property **ONSetting** As Double [r/w]
- Property **PTratio** As Double [r/w]
- Property **UseVoltOverride** As Boolean [r/w]
- Property **Vmax** As Double [r/w]
- Property **Vmin** As Double [r/w]

Circuit

This is the top-level interface class for the presently defined circuit. It is used frequently and contains the means to obtain values for the entire circuit as well as references to the collections classes for navigating through the various objects in the circuit. Use this interface to set the active bus, the active circuit element, etc.

Properties:

- Property **ActiveBus** As Bus [r/o]
- Property **ActiveCktElement** As CktElement [r/o]
- Property **ActiveClass** As ActiveClass [r/o]
- Property **ActiveDSSElement** As DSSElement [r/o]
- Property **ActiveElement** As CktElement [r/o]
- Property **AllBusDistances** As Variant [r/o]
- Property **AllBusNames** As Variant [r/o]
- Property **AllBusVmag** As Variant [r/o]
- Property **AllBusVmagPu** As Variant [r/o]
- Property **AllBusVolts** As Variant [r/o]
- Property **AllElementLosses** As Variant [r/o]
- Property **AllElementNames** As Variant [r/o]
- Property **AllNodeDistances** As Variant [r/o]
- Property **AllNodeDistancesByPhase**(ByVal **Phase** As Long) As Variant [r/o]
- Property **AllNodeNames** As Variant [r/o]
- Property **AllNodeNamesByPhase**(ByVal **Phase** As Long) As Variant [r/o]
- Property **AllNodeVmagByPhase**(ByVal **Phase** As Long) As Variant [r/o]

- Property **AllNodeVmagPUByPhase**(ByVal **Phase** As Long) As Variant [r/o]
- Property **Buses**(ByVal **Index** As Variant) As Bus [r/o]
- Property **Capacitors** As Capacitors [r/o]
- Property **CapControls** As CapControls [r/o]
- Property **CktElements**(ByVal **Idx** As Variant) As CktElement [r/o]
- Property **CtrlQueue** As CtrlQueue [r/o]
- Property **Generators** As Generators [r/o]
- Property **LineLosses** As Variant [r/o]
- Property **Lines** As Lines [r/o]
- Property **Loads** As Loads [r/o]
- Property **Losses** As Variant [r/o]
- Property **Meters** As Meters [r/o]
- Property **Monitors** As Monitors [r/o]
- Property **Name** As String [r/o]
- Property **NumBuses** As Long [r/o]
- Property **NumCktElements** As Long [r/o]
- Property **NumNodes** As Long [r/o]
- Property **RegControls** As RegControls [r/o]
- Property **Sensors** As Sensors [r/o]
- Property **Settings** As Settings [r/o]
- Property **Solution** As Solution [r/o]
- Property **SubstationLosses** As Variant [r/o]
- Property **SwtControls** As SwtControls [r/o]
- Property **SystemY** As Variant [r/o]
- Property **Topology** As Topology [r/o]
- Property **TotalPower** As Variant [r/o]
- Property **Transformers** As Transformers [r/o]

Methods:

- Function **Capacity**(ByVal **Start** As Double, ByVal **Increment** As Double) As Double
- Sub **Disable**(ByVal **Name** As String)
- Sub **Enable**(ByVal **Name** As String)
- Function **FirstElement**() As Long
- Function **FirstPCElement**() As Long
- Function **FirstPDElement**() As Long
- Function **NextElement**() As Long
- Function **NextPCElement**() As Long
- Function **NextPDElement**() As Long
- Sub **Sample**()
- Sub **SaveSample**()
- Function **SetActiveBus**(ByVal **BusName** As String) As Long
- Function **SetActiveBusi**(ByVal **BusIndex** As Long) As Long
- Function **SetActiveClass**(ByVal **ClassName** As String) As Long
- Function **SetActiveElement**(ByVal **FullName** As String) As Long
- Sub **UpdateStorage**()

CktElement

This interface always points to the active circuit element, which is the most recent circuit element referenced by any command or action in the program. Use this interface to obtain specific solution values for the active circuit element. There are methods for opening and closing the terminals (see also *SwtControl*). However, you can also accomplish this by reassigning the *BusNames* for the terminals using this interface (construct and pass a variant array of Unicode, or wide, strings).

Properties:

- Property **AllPropertyNames** As Variant [r/o]
- Property **BusNames** As Variant [r/w]
- Property **Controller** As String [r/o]
- Property **CplxSeqCurrents** As Variant [r/o]
- Property **CplxSeqVoltages** As Variant [r/o]
- Property **Currents** As Variant [r/o]
- Property **DisplayName** As String [r/w]
- Property **EmergAmps** As Double [r/w]
- Property **Enabled** As Boolean [r/w]
- Property **EnergyMeter** As String [r/o]
- Property **GUID** As String [r/o]
- Property **Handle** As Long [r/o]
- Property **HasSwitchControl** As Boolean [r/o]
- Property **HasVoltControl** As Boolean [r/o]
- Property **Losses** As Variant [r/o]
- Property **Name** As String [r/o]
- Property **NormalAmps** As Double [r/w]
- Property **NumConductors** As Long [r/o]
- Property **NumPhases** As Long [r/o]
- Property **NumProperties** As Long [r/o]
- Property **NumTerminals** As Long [r/o]
- Property **PhaseLosses** As Variant [r/o]
- Property **Powers** As Variant [r/o]
- Property **Properties**(ByVal **Indx** As Variant) As DSSProperty [r/o]
- Property **Residuals** As Variant [r/o]
- Property **SeqCurrents** As Variant [r/o]
- Property **SeqPowers** As Variant [r/o]
- Property **SeqVoltages** As Variant [r/o]
- Property **Voltages** As Variant [r/o]
- Property **Yprim** As Variant [r/o]

Methods:

- Sub **Close**(ByVal **Term** As Long, ByVal **Phs** As Long)
- Function **IsOpen**(ByVal **Term** As Long, ByVal **Phs** As Long) As Boolean
- Sub **Open**(ByVal **Term** As Long, ByVal **Phs** As Long)

CtrlQueue

This interface provides properties and methods for dealing with the internal control queue for delayed control actions.

Properties:

- Property **Action**(ByVal Long) [w/o]
- Property **ActionCode** As Long [r/o]
- Property **DeviceHandle** As Long [r/o]
- Property **NumActions** As Long [r/o]
- Property **PopAction** As Long [r/o]

Methods:

- Sub **ClearActions**()
- Sub **ClearQueue**()
- Sub **Delete**(ByVal **ActionHandle** As Long)
- Function **Push**(ByVal **Hour** As Long, ByVal **Seconds** As Double, ByVal **ActionCode** As Long, ByVal **DeviceHandle** As Long) As Long
- Sub **Show**()

DSS

This is the only class that is registered in the Windows Registry. It will be listed as OpenDSS*Engine.DSS*. After the class is loaded, the Start method is invoked to initialize everything else. There are properties for returning miscellaneous values as well as pointers to some key high-level interface classes such as *Text, ActiveCircuit*, and *Plot*.

Properties:

- Property **ActiveCircuit** As Circuit [r/o]
- Property **ActiveClass** As ActiveClass [r/o]
- Property **AllowForms** As Boolean [r/w]
- Property **Circuits**(ByVal **Idx** As Variant) As Circuit [r/o]
- Property **Classes** As Variant [r/o]
- Property **DataPath** As String [r/w]
- Property **DefaultEditor** As String [r/o]
- Property **DSSProgress** As DSSProgress [r/o]
- Property **Error** As Error [r/o]
- Property **Events** As DSSEvents [r/o]
- Property **Executive** As DSS_Executive [r/o]
- Property **NumCircuits** As Long [r/o]
- Property **NumClasses** As Long [r/o]
- Property **NumUserClasses** As Long [r/o]
- Property **Plot** As Plot [r/o]
- Property **Text** As Text [r/o]
- Property **UserClasses** As Variant [r/o]
- Property **Version** As String [r/o]

Methods:

- Sub **ClearAll**()
- Function **NewCircuit**(ByVal **Name** As String) As Circuit
- Sub **Reset**()
- Function **SetActiveClass**(ByVal **ClassName** As String) As Long
- Sub **ShowPanel**()
- Function **Start**(ByVal **code** As Long) As Boolean

DSSElement

This is a general interface class that acts on the active element, whether it is a circuit element or not. You can work with general library objects as well. The names of all the properties and their values (as DSSProperty types) can be conveniently obtained through this interface.

Properties:

- Property **AllPropertyNames** As Variant [r/o]
- Property **Name** As String [r/o]
- Property **NumProperties** As Long [r/o]
- Property **Properties**(ByVal **Indx** As Variant) As DSSProperty [r/o]

DSSEvents

This interface class manages the events related to the control simulations.

Events:

- Event **CheckControls**()
- Event **InitControls**()
- Event **StepControls**()

DSSProgress

This interface invokes the progress bar form in OpenDSS, which can be used for long solution processes.

Properties:

- Property **Caption**(ByVal String) [w/o]
- Property **PctProgress**(ByVal Long) [w/o]

Methods:

- Sub **Close**()
- Sub **Show**()

DSSProperty

Values and the names of the properties of the active DSS element can be retrieved or set using this interface.

Properties:

- Property **Description** As String [r/o]
- Property **Name** As String [r/o]
- Property **Val** As String [r/w]

DSS_Executive

This interface will return the names of the current OpenDSS commands and options along with the help string associated with each. This might serve as a means to automatically generate documentation for new revisions.

Properties:

- Property **Command**(ByVal **i** As Long) As String [r/o]
- Property **CommandHelp**(ByVal **i** As Long) As String [r/o]
- Property **NumCommands** As Long [r/o]
- Property **NumOptions** As Long [r/o]
- Property **Option**(ByVal **i** As Long) As String [r/o]
- Property **OptionHelp**(ByVal **i** As Long) As String [r/o]
- Property **OptionValue**(ByVal **i** As Long) As String [r/o]

Error

Check this interface after a command or other action has been executed to determine if an error has occurred.

Properties:

- Property **Description** As String [r/o]
- Property **Number** As Long [r/o]

Generators

This interface class manages a collection of the *Generator* objects in the circuit and exposes selected properties.

Properties:

- Property **AllNames** As Variant [r/o]
- Property **Count** As Long [r/o]
- Property **First** As Long [r/o]
- Property **ForcedON** As Boolean [r/w]
- Property **kV** As Double [r/w]
- Property **kvar** As Double [r/w]
- Property **kW** As Double [r/w]
- Property **Name** As String [r/w]

- Property **Next** As Long [r/o]
- Property **PF** As Double [r/w]
- Property **Phases** As Long [r/w]
- Property **RegisterNames** As Variant [r/o]
- Property **RegisterValues** As Variant [r/o]

Lines

This class is frequently used to retrieve or set values for *Line* objects for various analyses. For radial feeders, you can use the Parent property to trace back through the circuit if there is an EnergyMeter object at the head of the feeder. Note that you can obtain or set line impedance values in either matrix form (variant array of doubles) or symmetrical component values. The matrix values always give the values used in the simulation. If symmetrical component values were not used to define the *Line* object, the values retrieved will probably not make sense.

Properties:

- Property **AllNames** As Variant [r/o]
- Property **Bus1** As String [r/w]
- Property **Bus2** As String [r/w]
- Property **C0** As Double [r/w]
- Property **C1** As Double [r/w]
- Property **Cmatrix** As Variant [r/w]
- Property **Count** As Long [r/o]
- Property **EmergAmps** As Double [r/w]
- Property **First** As Long [r/o]
- Property **Geometry** As String [r/w]
- Property **Length** As Double [r/w]
- Property **LineCode** As String [r/w]
- Property **Name** As String [r/w]
- Property **Next** As Long [r/o]
- Property **NormAmps** As Double [r/w]
- Property **NumCust** As Long [r/o]
- Property **Parent** As Long [r/o]
- Property **Phases** As Long [r/w]
- Property **R0** As Double [r/w]
- Property **R1** As Double [r/w]
- Property **Rg** As Double [r/w]
- Property **Rho** As Double [r/w]
- Property **Rmatrix** As Variant [r/w]
- Property **Spacing** As String [r/w]
- Property **TotalCust** As Long [r/o]
- Property **X0** As Double [r/w]
- Property **X1** As Double [r/w]
- Property **Xg** As Double [r/w]
- Property **Xmatrix** As Variant [r/w]
- Property **Yprim** As Variant [r/w]

Methods:

- Function **New**(ByVal **Name** As String) As Long

Loads

This class is frequently used to retrieve or set values for *Load* objects for various analyses.

Properties:

- Property **AllNames** As Variant [r/o]
- Property **AllocationFactor** As Double [r/w]
- Property **Cfactor** As Double [r/w]
- Property **Class** As Long [r/w]
- Property **Count** As Long [r/o]
- Property **CVRcurve** As String [r/w]
- Property **CVRvars** As Double [r/w]
- Property **CVRwatts** As Double [r/w]
- Property **daily** As String [r/w]
- Property **duty** As String [r/w]
- Property **First** As Long [r/o]
- Property **Growth** As String [r/w]
- Property **Idx** As Long [r/w]
- Property **IsDelta** As Boolean [r/w]
- Property **kV** As Double [r/w]
- Property **kva** As Double [r/w]
- Property **kvar** As Double [r/w]
- Property **kW** As Double [r/w]
- Property **kwh** As Double [r/w]
- Property **kwhdays** As Double [r/w]
- Property **Model** As LoadModels [r/w]
- Property **Name** As String [r/w]
- Property **Next** As Long [r/o]
- Property **NumCust** As Long [r/w]
- Property **PctMean** As Double [r/w]
- Property **PctStdDev** As Double [r/w]
- Property **PF** As Double [r/w]
- Property **Rneut** As Double [r/w]
- Property **Spectrum** As String [r/w]
- Property **Status** As LoadStatus [r/w]
- Property **Vmaxpu** As Double [r/w]
- Property **Vminemerg** As Double [r/w]
- Property **Vminnorm** As Double [r/w]
- Property **Vminpu** As Double [r/w]
- Property **xfkVA** As Double [r/w]
- Property **Xneut** As Double [r/w]
- Property **Yearly** As String [r/w]

Meters

This class is used to retrieve or set values for EnergyMeter objects.

Properties:

- Property **AllBranchesInZone** As Variant [r/o]
- Property **AllEndElements** As Variant [r/o]
- Property **AllNames** As Variant [r/o]
- Property **AllocFactors** As Variant [r/w]
- Property **CalcCurrent** As Variant [r/w]
- Property **Count** As Long [r/o]
- Property **CountBranches** As Long [r/o]
- Property **CountEndElements** As Long [r/o]
- Property **DIFilesAreOpen** As Boolean [r/o]
- Property **First** As Long [r/o]
- Property **MeteredElement** As String [r/w]
- Property **MeteredTerminal** As Long [r/w]
- Property **Name** As String [r/w]
- Property **Next** As Long [r/o]
- Property **Peakcurrent** As Variant [r/w]
- Property **RegisterNames** As Variant [r/o]
- Property **RegisterValues** As Variant [r/o]
- Property **Totals** As Variant [r/o]

Methods:

- Sub **CloseAllDIFiles**()
- Sub **OpenAllDIFiles**()
- Sub **Reset**()
- Sub **ResetAll**()
- Sub **Sample**()
- Sub **SampleAll**()
- Sub **Save**()
- Sub **SaveAll**()

Monitors

This interface class is a collections class that is used mostly to extract values from *Monitor* objects. Monitors are file streams kept in memory. The ByteStream property is a copy of the file stream. Note that in many programming languages, such as MATLAB, which have excellent facilities for handling CSV files, it is generally simpler to export the monitor data from OpenDSS and read it into the controlling program.

Properties:

- Property **AllNames** As Variant [r/o]
- Property **ByteStream** As Variant [r/o]
- Property **Count** As Long [r/o]
- Property **FileName** As String [r/o]
- Property **First** As Long [r/o]
- Property **Mode** As Long [r/w]

- Property **Name** As String [r/w]
- Property **Next** As Long [r/o]
- Property **SampleCount** As Long [r/o]

Methods:

- Sub **Reset**()
- Sub **ResetAll**()
- Sub **Sample**()
- Sub **SampleAll**()
- Sub **Save**()
- Sub **SaveAll**()
- Sub **Show**()

Plot

This interface class provides access to the plotting functions implemented in the OpenDSS program via the *DSSGraph* DLL.

Properties:

- Property **CenterMarkerCode**(ByVal Long) [w/o]
- Property **CurveMarkerCode**(ByVal Long) [w/o]
- Property **DataColor**(ByVal Long) [w/o]
- Property **KeepAspect**(ByVal Boolean) [w/o]
- Property **LineWidth**(ByVal Long) [w/o]
- Property **MarkCenter**(ByVal Boolean) [w/o]
- Property **MarkCurves**(ByVal Boolean) [w/o]
- Property **MarkNodes**(ByVal Boolean) [w/o]
- Property **NodeMarkerCode**(ByVal Long) [w/o]
- Property **NodeMarkerWidth**(ByVal Long) [w/o]
- Property **pctRim**(ByVal Double) [w/o]
- Property **PenStyle**(ByVal Long) [w/o]
- Property **PlotCaption**(ByVal String) [w/o]
- Property **TextAlign**(ByVal Long) [w/o]
- Property **TextColor**(ByVal Long) [w/o]
- Property **TextSize**(ByVal Long) [w/o]
- Property **WindowCaption**(ByVal String) [w/o]
- Property **XLabel**(ByVal String) [w/o]
- Property **YLabel**(ByVal String) [w/o]

Methods:

- Sub **AddCentered15**(ByVal **x** As Double, ByVal **y** As Double, ByVal **Txt** As String)
- Function **AddLabel**(ByVal **x** As Double, ByVal **y** As Double, ByVal **Txt** As String) As Long
- Sub **DrawRectangle**(ByVal **XLowerLeft** As Double, ByVal **YLowerLeft** As Double, ByVal **XUpperRight** As Double, ByVal **YUpperRight** As Double)
- Sub **DrawToXY**(ByVal **x** As Double, ByVal **y** As Double)
- Sub **GetGraphProperties**(ByRef **Xmin** As Double, ByRef **Xmax** As Double, ByRef **Ymin** As Double, ByRef **Ymax** As Double, ByRef **ChartColor** As Long, ByRef **WindowColor** As Long, ByRef **Isometric** As Boolean, ByRef **Gridstyle** As Long)

- Sub **GetRange**(ByRef **Xlow** As Double, ByRef **Xhigh** As Double, ByRef **Ylow** As Double, ByRef **Yhigh** As Double)
- Sub **GetWindowParms**(ByRef **Width** As Long, ByRef **LRim** As Long, ByRef **RRim** As Long, ByRef **Height** As Long, ByRef **TRim** As Long, ByRef **Brim** As Long)
- Sub **LockInLabel**(ByVal **TxtIndex** As Long)
- Sub **MarkAtXY**(ByVal **x** As Double, ByVal **y** As Double, ByVal **MarkerCode** As Byte, ByVal **MarkerSize** As Byte)
- Sub **MoveToXY**(ByVal **x** As Double, ByVal **y** As Double)
- Sub **NewCircle**(ByVal **Xc** As Double, ByVal **Yc** As Double, ByVal **Radius** As Double)
- Sub **NewCurve**(ByVal **Xarray** As Variant, ByVal **Yarray** As Variant, ByVal **Name** As String)
- Function **NewGraph**() As Long
- Sub **NewLine**(ByVal **X1** As Double, ByVal **Y1** As Double, ByVal **X2** As Double, ByVal **Y2** As Double, ByVal **Name** As String)
- Sub **NewMarker**(ByVal **x** As Double, ByVal **y** As Double, ByVal **MarkerCode** As Byte, ByVal **MarkerSize** As Byte)
- Sub **NewText**(ByVal **X1** As Double, ByVal **Y1** As Double, ByVal **S** As String)
- Sub **SetFontStyle**(ByVal **Bold** As Boolean, ByVal **Italic** As Boolean, ByVal **Underline** As Boolean, ByVal **Strikeout** As Boolean)
- Sub **SetForClickOnDiagram**()
- Sub **SetForNoScales**()
- Sub **SetGraphProperties**(ByVal **Xmin** As Double, ByVal **Xmax** As Double, ByVal **Ymin** As Double, ByVal **Ymax** As Double, ByVal **ChartColor** As Long, ByVal **WindowColor** As Long, ByVal **Isometric** As Boolean, ByVal **Gridstyle** As Long)
- Sub **SetLabelBold**(ByVal **LblIndex** As Long)
- Sub **SetLabelLeft**(ByVal **LblIndex** As Long)
- Sub **SetRange**(ByVal **Xlow** As Double, ByVal **Xhigh** As Double, ByVal **Ylow** As Double, ByVal **Yhigh** As Double)
- Sub **Show**()

RegControls

This interface class is a collections class that manages selected properties of *RegControl* objects.

Properties:

- Property **AllNames** As Variant [r/o]
- Property **Count** As Long [r/o]
- Property **CTPrimary** As Double [r/w]
- Property **Delay** As Double [r/w]
- Property **First** As Long [r/o]
- Property **ForwardBand** As Double [r/w]
- Property **ForwardR** As Double [r/w]
- Property **ForwardVreg** As Double [r/w]
- Property **ForwardX** As Double [r/w]
- Property **IsInverseTime** As Boolean [r/w]
- Property **IsReversible** As Boolean [r/w]
- Property **MaxTapChange** As Long [r/w]
- Property **MonitoredBus** As String [r/w]
- Property **Name** As String [r/w]
- Property **Next** As Long [r/o]
- Property **PTratio** As Double [r/w]
- Property **ReverseBand** As Double [r/w]
- Property **ReverseR** As Double [r/w]

- Property **ReverseVreg** As Double [r/w]
- Property **ReverseX** As Double [r/w]
- Property **TapDelay** As Double [r/w]
- Property **TapWinding** As Long [r/w]
- Property **Transformer** As String [r/w]
- Property **VoltageLimit** As Double [r/w]
- Property **Winding** As Long [r/w]

Sensors

This interface class manages selected properties of *Sensor* objects.

Properties:

- Property **AllNames** As Variant [r/o]
- Property **Count** As Long [r/o]
- Property **Currents** As Variant [r/w]
- Property **First** As Long [r/o]
- Property **IsDelta** As Boolean [r/w]
- Property **kVARS** As Variant [r/w]
- Property **kVBase** As Double [r/w]
- Property **kVS** As Variant [r/w]
- Property **kWS** As Variant [r/w]
- Property **MeteredElement** As String [r/w]
- Property **MeteredTerminal** As Long [r/w]
- Property **Name** As String [r/w]
- Property **Next** As Long [r/o]
- Property **PctError** As Double [r/w]
- Property **ReverseDelta** As Boolean [r/w]
- Property **Weight** As Double [r/w]

Methods:

- Sub **Reset**()
- Sub **ResetAll**()

Settings

Selected options may be set directly through this interface. Otherwise, send a *Set* command through the *Text* interface.

Properties:

- Property **AllocationFactors**(ByVal Double) [w/o]
- Property **AllowDuplicates** As Boolean [r/w]
- Property **AutoBusList** As String [r/w]
- Property **CktModel** As Long [r/w]
- Property **ControlTrace** As Boolean [r/w]
- Property **EmergVmaxpu** As Double [r/w]
- Property **EmergVminpu** As Double [r/w]
- Property **LossRegs** As Variant [r/w]
- Property **LossWeight** As Double [r/w]

- Property **NormVmaxpu** As Double [r/w]
- Property **NormVminpu** As Double [r/w]
- Property **PriceCurve** As String [r/w]
- Property **PriceSignal** As Double [r/w]
- Property **Trapezoidal** As Boolean [r/w]
- Property **UEregs** As Variant [r/w]
- Property **UEweight** As Double [r/w]
- Property **VoltageBases** As Variant [r/w]
- Property **ZoneLock** As Boolean [r/w]

Solution

This interface class provides access to selected properties of the solution process in OpenDSS. Since the *Solve* and *Set* commands are essentially the same, you can set several global options with this interface as well. Use this class to write custom solution algorithms.

Properties:

- Property **AddType** As Long [r/w]
- Property **Algorithm** As Long [r/w]
- Property **Capkvar** As Double [r/w]
- Property **ControlActionsDone** As Boolean [r/w]
- Property **ControlIterations** As Long [r/w]
- Property **ControlMode** As Long [r/w]
- Property **Converged** As Boolean [r/w]
- Property **dblHour** As Double [r/w]
- Property **DefaultDaily** As String [r/w]
- Property **DefaultYearly** As String [r/w]
- Property **EventLog** As Variant [r/o]
- Property **Frequency** As Double [r/w]
- Property **GenkW** As Double [r/w]
- Property **GenMult** As Double [r/w]
- Property **GenPF** As Double [r/w]
- Property **Hour** As Long [r/w]
- Property **Iterations** As Long [r/o]
- Property **LDCurve** As String [r/w]
- Property **LoadModel** As Long [r/w]
- Property **LoadMult** As Double [r/w]
- Property **MaxControlIterations** As Long [r/w]
- Property **MaxIterations** As Long [r/w]
- Property **Mode** As Long [r/w]
- Property **ModeID** As String [r/o]
- Property **MostIterationsDone** As Long [r/o]
- Property **Number** As Long [r/w]
- Property **pctGrowth** As Double [r/w]
- Property **Random** As Long [r/w]
- Property **Seconds** As Double [r/w]
- Property **StepSize** As Double [r/w]
- Property **StepsizeHr**(ByVal Double) [w/o]
- Property **StepsizeMin**(ByVal Double) [w/o]
- Property **SystemYChanged** As Boolean [r/o]
- Property **Tolerance** As Double [r/w]

- Property **Totaliterations** As Long [r/o]
- Property **Year** As Long [r/w]

Methods:

- Sub **BuildYMatrix**(ByVal **BuildOption** As Long, ByVal **AllocateVI** As Long)
- Sub **CheckControls**()
- Sub **CheckFaultStatus**()
- Sub **DoControlActions**()
- Sub **InitSnap**()
- Sub **SampleControlDevices**()
- Sub **Sample_DoControlActions**()
- Sub **Solve**()
- Sub **SolveDirect**()
- Sub **SolveNoControl**()
- Sub **SolvePflow**()
- Sub **SolvePlusControl**()
- Sub **SolveSnap**()

SwtControls

This is a collections class for *SwtControl* objects.

Properties:

- Property **Action** As ActionCodes [r/w]
- Property **AllNames** As Variant [r/o]
- Property **Count** As Long [r/o]
- Property **Delay** As Double [r/w]
- Property **First** As Long [r/o]
- Property **IsLocked** As Boolean [r/w]
- Property **Name** As String [r/w]
- Property **Next** As Long [r/o]
- Property **SwitchedObj** As String [r/w]
- Property **SwitchedTerm** As Long [r/w]

Text

This class provides a direct interface to OpenDSS script processing. A line of text is sent to the OpenDSS command processor using the *Command* property. If the command results in an error or a value, it will appear in the read-only *Result* property immediately after executing the command. Any OpenDSS command can be executed through this interface.

Properties:

- Property **Command** As String [r/w]
- Property **Result** As String [r/o]

Topology

This is an interface for iterating backward or forward through branches in the circuit. It is useful for switching algorithms such as the branch exchange method.

Properties:

- Property **ActiveBranch** As Long [r/o]
- Property **ActiveLevel** As Long [r/o]
- Property **AllIsolatedBranches** As Variant [r/o]
- Property **AllIsolatedLoads** As Variant [r/o]
- Property **AllLoopedPairs** As Variant [r/o]
- Property **BackwardBranch** As Long [r/o]
- Property **BranchName** As String [r/w]
- Property **BusName** As String [r/w]
- Property **First** As Long [r/o]
- Property **FirstLoad** As Long [r/o]
- Property **ForwardBranch** As Long [r/o]
- Property **LoopedBranch** As Long [r/o]
- Property **Next** As Long [r/o]
- Property **NextLoad** As Long [r/o]
- Property **NumIsolatedBranches** As Long [r/o]
- Property **NumIsolatedLoads** As Long [r/o]
- Property **NumLoops** As Long [r/o]
- Property **ParallelBranch** As Long [r/o]

Transformers

This is a collections class for *Transformer* objects. Many of the properties available through this interface are read/write (r/w), allowing you to perform analyses that might require numerous parameter modifications.

Properties:

- Property **AllNames** As Variant [r/o]
- Property **Count** As Long [r/o]
- Property **First** As Long [r/o]
- Property **IsDelta** As Boolean [r/w]
- Property **kV** As Double [r/w]
- Property **kva** As Double [r/w]
- Property **MaxTap** As Double [r/w]
- Property **MinTap** As Double [r/w]
- Property **Name** As String [r/w]
- Property **Next** As Long [r/o]
- Property **NumTaps** As Long [r/w]
- Property **NumWindings** As Long [r/w]
- Property **R** As Double [r/w]
- Property **Rneut** As Double [r/w]
- Property **Tap** As Double [r/w]
- Property **Wdg** As Long [r/w]
- Property **XfmrCode** As String [r/w]
- Property **Xhl** As Double [r/w]
- Property **Xht** As Double [r/w]
- Property **Xlt** As Double [r/w]
- Property **Xneut** As Double [r/w]

ActionCodes

These are constants for *SwtControl* objects:

- **dssActionNone** As Long=0
- **dssActionOpen** As Long=1
- **dssActionClose** As Long=2
- **dssActionReset** As Long=3
- **dssActionLock** As Long=4
- **dssActionUnlock** As Long=5
- **dssActionTapUp** As Long=6
- **dssActionTapDown** As Long=7

CapControlModes

These are constants for *CapControl* object types:

- **dssCapControlVoltage** As Long=1
- **dssCapControlKvar** As Long=2
- **dssCapControlCurrent** As Long=0
- **dssCapControlPF** As Long=4
- **dssCapControlTime** As Long=3

LoadModels

These are constants for the *Model* property of *Load* objects:

- **dssLoadConstPQ** As Long=1
- **dssLoadConstZ** As Long=2
- **dssLoadMotor** As Long=3
- **dssLoadCVR** As Long=4
- **dssLoadConstI** As Long=5
- **dssLoadConstPFixedQ** As Long=6
- **dssLoadConstPFixedX** As Long=7

LoadStatus

These are constants for the *Status* property of *Load* objects. *Load* objects are variable by default.

- **dssLoadVariable** As Long=0
- **dssLoadFixed** As Long=1
- **dssLoadExempt** As Long=2

MonitorModes

These are constants for the *Mode* property of *Monitor* objects:

- **dssVI** As Long=0
- **dssPower** As Long=1
- **dssSequence** As Long=16
- **dssMagnitude** As Long=32
- **dssPosOnly** As Long=64
-

- **dssTaps** As Long=2
- **dssStates** As Long=3

Options

These are constants for general circuit options:

- **dssPowerFlow** As Long=1
- **dssAdmittance** As Long=2
- **dssNormalSolve** As Long=0
- **dssNewtonSolve** As Long=1
- **dssStatic** As Long=0
- **dssEvent** As Long=1
- **dssTime** As Long=2
- **dssMultiphase** As Long=0
- **dssPositiveSeq** As Long=1
- **dssGaussian** As Long=1
- **dssUniform** As Long=2
- **dssLogNormal** As Long=3
- **dssAddGen** As Long=1
- **dssAddCap** As Long=2

SolveModes

These are constants for the built-in solution modes:

- **dssSnapShot** As Long=0
- **dssDutyCycle** As Long=6
- **dssDirect** As Long=7
- **dssDaily** As Long=1
- **dssMonte1** As Long=3
- **dssMonte2** As Long=10
- **dssMonte3** As Long=11
- **dssFaultStudy** As Long=9
- **dssYearly** As Long=2
- **dssMonteFault** As Long=8
- **dssPeakDay** As Long=5
- **dssLD1** As Long=4
- **dssLD2** As Long=12
- **dssAutoAdd** As Long=13
- **dssHarmonic** As Long=15
- **dssDynamic** As Long=14

Export Control Restrictions

Access to and use of EPRI Intellectual Property is granted with the specific understanding and requirement that responsibility for ensuring full compliance with all applicable U.S. and foreign export laws and regulations is being undertaken by you and your company. This includes an obligation to ensure that any individual receiving access hereunder who is not a U.S. citizen or permanent U.S. resident is permitted access under applicable U.S. and foreign export laws and regulations. In the event you are uncertain whether you or your company may lawfully obtain access to this EPRI Intellectual Property, you acknowledge that it is your obligation to consult with your company's legal counsel to determine whether this access is lawful. Although EPRI may make available on a case-by-case basis an informal assessment of the applicable U.S. export classification for specific EPRI Intellectual Property, you and your company acknowledge that this assessment is solely for informational purposes and not for reliance purposes. You and your company acknowledge that it is still the obligation of you and your company to make your own assessment of the applicable U.S. export classification and ensure compliance accordingly. You and your company understand and acknowledge your obligations to make a prompt report to EPRI and the appropriate authorities regarding any access to or use of EPRI Intellectual Property hereunder that may be in violation of applicable U.S. or foreign export laws or regulations.

The Electric Power Research Institute Inc., (EPRI, www.epri.com) conducts research and development relating to the generation, delivery and use of electricity for the benefit of the public. An independent, nonprofit organization, EPRI brings together its scientists and engineers as well as experts from academia and industry to help address challenges in electricity, including reliability, efficiency, health, safety and the environment. EPRI also provides technology, policy and economic analyses to drive long-range research and development planning, and supports research in emerging technologies. EPRI's members represent more than 90 percent of the electricity generated and delivered in the United States, and international participation extends to 40 countries. EPRI's principal offices and laboratories are located in Palo Alto, Calif.; Charlotte, N.C.; Knoxville, Tenn.; and Lenox, Mass.

Together…Shaping the Future of Electricity

1020090

Electric Power Research Institute
3420 Hillview Avenue, Palo Alto, California 94304-1338 • PO Box 10412, Palo Alto, California 94303-0813 • USA
800.313.3774 • 650.855.2121 • askepri@epri.com • www.epri.com