ELECTRIC POWER RESEARCH INSTITUTE

OpenDSS Documentation

CapControl User DLL

January 26, 2015

The OpenDSS *CapControl* element model supports a user-written DLL model as described in this document. It is for representing custom control models for all types of simulations. It is invoked when the *Sample* function is called after achieving a converged power flow solution. The control will sample voltages and currents at the monitored location and make a decision on whether or not to switch on (or off) one or more capacitor steps. A switching message is pushed onto the Control Queue by the parent CapControl module to be executed at the appropriate time during the solution. Sometimes this is delayed and sometimes it is an immediate action.

The OpenDSS *Capacitor* element model is conceptually a multi-step filter bank as shown in Figure 1. By default, it is a shunt capacitor bank, but the 2nd terminal may be connected to another bus to achieve a series capacitor or some other configuration. In either case, it is a multi-step capacitor in which each branch can be optionally tuned to a selected frequency as in a shunt filter.

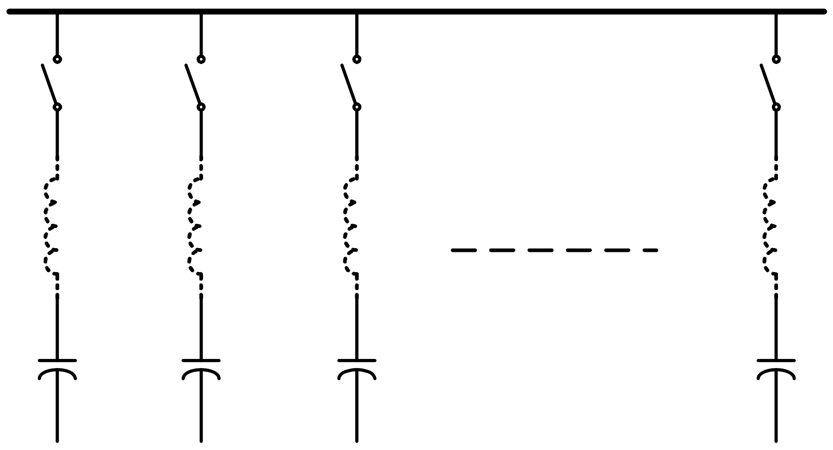

Figure 1. OpenDSS Capacitor Model

The purpose of the CapControl object is to control the number of steps active based on selected control criteria such as voltage, current, kvar, or time of day.

As with all control devices in OpenDSS, a CapControl element is connected to a terminal of a current-carrying element such as a Line or Transformer object. In addition, a CapControl element is connected to a Capacitor element in the circuit that the CapControl element controls. Therefore, the CapControl element can monitor voltages and currents in one location and control a capacitor in another location. This gives good flexibility for modeling various control strategies.

The CapControl element model is meant to represent typical capacitor controls used on utility distribution systems. It is assumed connected to the monitoring location through CTs and PTs of some sort. OpenDSS allows the connection of these devices to one or all phases of the monitoring location as is commonly done in practice. If the control quantity is kvar, all phases are assumed by default.

Control quantities in the standard control in OpenDSS are Current, Voltage, kvar, Power Factor, and Time. The values of the ONsetting and OFFsetting properties are set to correspond to the control type.

The CapControl is typically located downline (load side) of the bus to which the shunt capacitor is connected (see Figure 2). It samples the voltage and current at this location and controls the switch(es) in the Capacitor object.

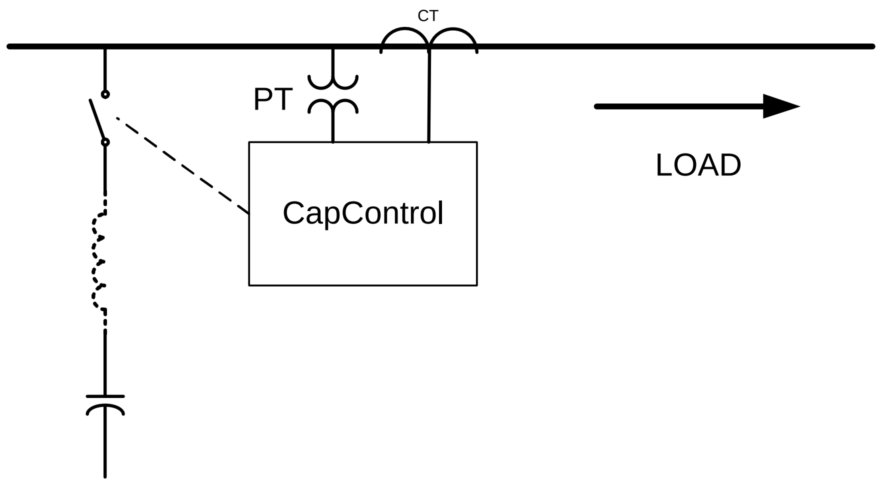

Figure 2. CapControl Object is Typically Placed Downline of Controlled Capacitor

It is installed on the downline side when using current or kvar controls so that these values reflect the quantity being controlled. If installed on the upline side, the current and kvar would change dramatically with each switching operation.

If the control quantity is voltage or time the location of the CapControl with respect to the Capacitor is less important.

Public Data Structure for CapControl

All devices in OpenDSS can have a public data structure if the programmer has provided it. A pointer to the structure may be obtained for the active circuit element through the **GetPublicDataPtr** function in the Callback routines. A pointer to the TDSSCallBacks structure is passed through argument list for the **New** function when a new instance of the user-written model is created.

The Pascal declaration of the public data structure for the Storage element is:

```
{Fixed record structure for Public CapControl variables}
   TCapControlVars = Packed Record

            FCTPhase,
            FPTPhase  :Integer;   // "ALL" is -1
            AvailableSteps :Integer;   // available steps in controlled capacitor

            ON_Value,
            OFF_Value,
            PFON_Value,
            PFOFF_Value,
            CTRatio,
            PTRatio,
            ONDelay,
            OFFDelay,
            DeadTime,
            LastOpenTime   :Double;

            Voverride      :Boolean;
            VoverrideEvent :Boolean;
            VoverrideBusSpecified  :Boolean;     // Added 8-11-11

            VOverrideBusIndex :Integer;

            Vmax             :Double;
            Vmin             :Double;
            FPendingChange   :EControlAction;
            ShouldSwitch     :Boolean;  // True: action is pending
            Armed            :Boolean;  // Control is armed for switching unless reset
            PresentState     :EControlAction;
            InitialState     :EControlAction;

            SampleP    :Complex;        // two 64-bit numbers, kW, kvar
            SampleV    :Double;
            SampleCurr :Double;

   End;
```

Public Data Variable Definitions

FCTPhase : Integer;

Number of the phase on which the CT is installed. -1 = all phases.

FPTPhase : Integer;

Number of the phase on which the PT is installed. -1 = all phases.

AvailableSteps : Integer;

Capacitor steps available to be turned on.

ON_Value: Double;

Value at which the control arms to switch the capacitor ON (or ratchet up a step). Type of Control:

Current: Line Amps / CTratio

Voltage: Line-Neutral (or Line-Line for delta) Volts / PTratio

kvar: Total kvar, all phases (3-phase for pos seq model). This is directional.

PF: Power Factor, Total power in monitored terminal. Negative for Leading.

Time: Hrs from Midnight as a floating point number (decimal). 7:30am would be entered as 7.5.

OFF_Value: Double;

Value at which the control arms to switch the capacitor OFF.

PFON_Value:Double;

For Power Factor control, Power Factor for which the control arms to switch the capacitor ON.

PFOFF_Value:Double;

For Power Factor control, Power Factor for which the control arms to switch the capacitor OFF.

CTRatio:Double;

Ratio of the CT from line amps to control ampere setting for current and kvar control types.

PTRatio:Double;

Ratio of the PT that converts the monitored voltage to the control voltage.

ONDelay:Double;

Time delay, in seconds, from when the control is armed before it sends out the switching command to turn ON. The control may reset before the action actually occur..

OFFDelay:Double;

Time delay, in seconds, for control to turn OFF when present state is ON.

DeadTime:Double;

Dead time after capacitor is turned OFF before it can be turned back ON.

LastOpenTime:Double;

Time, sec, at which the capacitor opened (before allowing time to discharge).

Voverride:Boolean;

Flag signifying voltage override is active.

VoverrideEvent: Boolean;

Set to true if last control event was due to voltage override.

VoverrideBusSpecified: Boolean;

True if a bus different than the monitored element is specified for voltage override function.

VOverrideBusIndex: Integer;

Index into the bus list of the bus specified for voltage override function.

Vmax:Double;

Maximum voltage, in volts. If the voltage across the capacitor divided by the PTRATIO is greater than this voltage, the capacitor will switch OFF regardless of other control settings.

Vmin:Double;

Minimum voltage, in volts. If the voltage across the capacitor divided by the PTRATIO is less than this voltage, the capacitor will switch ON regardless of other control settings..

PresentState   :EControlAction;

InitialState   :EControlAction;

FPendingChange   :EControlAction;

EControlAction is an enumeration defined as:

```
    EControlAction = (
        CTRL_NONE,  {0}
        CTRL_OPEN,  {1}
        CTRL_CLOSE,  {2}
        CTRL_RESET,
        CTRL_LOCK,
        CTRL_UNLOCK,
        CTRL_TAPUP,
        CTRL_TAPDOWN);
```

ShouldSwitch   :Boolean; // True: action is pending

If True, switchaction is pending. User DLL should set this to true to signify to the Sample routine to consider pushing the control message onto the control queue.

Armed   :Boolean; //

Control is armed for switching unless reset.

SampleP   :Complex;

Power sample, kW, kvar, passed to the user control. (two doubles)

SampleV   :Double;

Voltage sample, volts, passed to the user control.

SampleCurr :Double;

Current sample, amps, passed to the user control.

DLL Function Interface

The declaration of the functions and procedures in the DLL interface for a user-written DLL are defined in the *CapUserControl.pas* module in the OpenDSS source code. The name of the DLL is specified by the

UserModel=MyUserWrittenDLL

statement in the description of the CapControl element. The OpenDSS will then load the DLL and attempt to use it. **All** the functions described here will have to be found in the DLL interface before the OpenDSS will continue.

Note: The function names in the DLL interface are case sensitive, although most other names associated with OpenDSS are not.

User-written DLLs should be capable of supporting more than one instance of a model. Each of several *CapControl* elements in an OpenDSS circuit model could invoke the same DLL. The DLL is only loaded one time, but could be called by more than one *CapControl* element. Therefore, the DLL must be able to manage multiple instances. An integer handle is assigned to each instance by the DLL (see below).

All function calls use the Stdcall calling convention. This is also the convention used in the Windows API.

If you would like to study the DLL interface in OpenDSS, please see the *CapUserControl.Pas* file in the *..\Source\Controls* folder on the source code repository (currently Sourceforge.net).

Function New

(Var CallBacks : TDSSCallBacks) : Integer; Stdcall;// Make a new instance

This function creates a new instance of the model defined by the DLL. It returns an integer handle to the newly-created instance that may be used to specifically select this instance at a later time. (OpenDSS can generally take care of this automatically.)

User-written DLLs should have some mechanism for keeping track of instances of the class represented by the DLL. This could be a simple array of pointers or a linked list or some other mechanism of the programmer's choosing. There is no internal OpenDSS mechanism to support this. Each CapControl object that creates a user-written CapControl model will keep track of only the one that belongs to it and use the Select function to instruct the DLL with instance to make active.

Arguments:

CallBacks: The callback structure is passed by reference. See *OpenDSS Callback Routines* documentation. The user-written DLL would typically keep a pointer to this structure and call the functions and procedures in it to obtain specific data (e.g., voltages and currents) or to use internal OpenDSS functions such as messaging.

Procedure Delete

(var ID:Integer); Stdcall; // deletes specified instance

This procedure deletes a specific instance referenced by the ID. The ID is the value returned by the New function when the instance was created.

Function Select

(var ID:Integer):Integer; Stdcall; // Select active instance

This function selects an instance of the objects instantiated by the DLL. The ID is the value returned by the New function. ID is passed by reference (i.e., expect a pointer to an integer).

If successful, the function returns the ID as confirmation. If the return value is 0, there has been an error.

Procedure Sample

(V, I:pComplexArray); stdcall;

This procedure is invoked by the CapControl Sample function that is automatically called after a successful power flow solution (all control objects are called after a solution).

This function pushes control actions onto the control queue to control the disposition of the capacitor step switches

Procedure Edit

(EditStr:pAnsichar; Maxlen:Cardinal); Stdcall;

The Edit procedure is called when the OpenDSS encounters the

UserData=(*MyString*)

property definition in the script. *MyString* is passed to this procedure in the EditStr argument as a pointer to a null-terminated Ansi character string. It is up to the Edit procedure in the DLL to interpret the string and properly set any variables. Callbacks to the OpenDSS parser may be used to help interpret the string if an OpenDSS-like syntax is used. However, any syntax may be used; only this model needs to know how to process it.

The length of the string is passed in the *Maxlen* argument, which is pushed onto the calling stack in case it is expected by the language in which the DLL is implemented (some languages like Fortran expect this argument). Languages like C++ and Delphi do not generally require this argument.

Note that it is not necessary to support OpenDSS syntax. Any format may be used for the data required to define the model. If the model requires an extensive amount of data, one approach is to pass a file name in *MyString* and have the DLL's Edit procedure read the data from the file.

Procedure UpdateModel;

StdCall;

This procedure is called by OpenDSS when it thinks it is necessary to recalculate model parameters from the present values of the data used to describe the models. Many models will not need this, but it is provided. It is typically called after calling the Edit procedure.

Example

Code snippets are provided from an actual CapControl user DLL implementation written for XYZ company to give you an idea how to implement a similar DLL. Some proprietary information has been removed from these examples. Keep in mind that you must provide all the functions in the DLL interface.

New Function Implementation

This function is called when "UserModel=*DLLName"* is encountered in the CapControl script. OpenDSS loads the DLL unless it is already loaded and calls the New function in the DLL to make a new instance of the user-written model. In this implementation, the New functions creates a new instance of the model type by calling its constructor. The variable *ActiveModel* points to the new instance and it is added to the list of models of this type. The list is simply a list of pointers to specific instances in this case. The result that is returned to OpenDSS is a 1-based integer index into the list.

```
Function  New(Var CallBacks:TDSSCallBacks): Integer;  Stdcall;// Make a new instance
Begin
    ActiveModel := TXYZCapControlModel.Create(CallBacks);
    Result := ModelList.Add(ActiveModel)+1;
End;
```

The TXYZCapControlModel constructor called in the code above (Create function) is implemented as follows:

```
{-------------------------------------------------------------------}
constructor TXYZCapControlModel.Create(Var CallBacks:TDSSCallBacks);
{-------------------------------------------------------------------}
Var
   PublicDataSize : Integer;
   x : Pointer;

begin

   CallBack := @CallBacks;
   With CallBack^ Do
   Begin
         Fowner := GetActiveElementPtr;  // for control queue pushes
         GetDynamicsStruct(x);
         FDynamicsRec := x;
         // Connect ControlVars to public data Struct
         GetPublicDataPtr(x, PublicDataSize);
         pControlVars := x;
   End;

   PF := 0.98;  // initial power factor
   QConst := Sqrt(1.0/SQR(PF) - 1.0);

   RecalcElementData;

end;
```

While some user-written model DLLs have a fairly complex Constructor, the one for the CapControl is relatively simple. It saves the address of the OpenDSS call-back routines in a local variable (CallBack). Then it saves a pointer to the active circuit element, which is the parent CapControl object. It also saves pointers to the Dynamics struct in OpenDSS, which has the time, time step, and various sundry values, and to the PublicDataPtr for the parent CapControl object. See above for the definition of the CapControl PublicData structure. With this pointer, the user-written code can reference virtually any data value of the CapControl module. (This example uses only a few.)

Select Function Implementation

This is a key function that is used by OpenDSS to select a particular instance of custom CapControl objects that this DLL supports. The *UserModel.Exists* boolean function in the CapControl.pas model invokes the Select function. It returns True if the return value from the Select function is greater than zero. This example uses a Tlist class from the Delphi VCL, but a simple array would also suffice.

```
Function  Select(var ID:Integer):Integer; Stdcall;    // Select active instance
Begin
     Result := 0;
     If ID <= ModelList.Count Then Begin
        ActiveModel := ModelList.Items[ID-1];
        If ActiveModel <> Nil Then Result := ID;
     End;
End;
```

Sample Function Implementation

```
Procedure  Sample; stdcall; // Samples present monitored element
{
 Perform calculations related to model
}
Begin

  If ActiveModel <> Nil Then ActiveModel.Sample;

End;
```

The *Sample* procedure in the main DLL interface then invokes the *Sample* procedure of the specific active user-written CapControl that has been selected, which is implemented as follows:

```
procedure TXYZCapControlModel.Sample;
{------------------------------------------------------------------------------------------
--------------------}

Var
    S : Complex;
    Q : Double;


begin

    Begin
          //----MonitoredElement.ActiveTerminalIdx := ElementTerminal;
          S := pControlVars^.SampleP;
          Q := S.im * 0.001 + Qconst * S.re * 0.001;  // kvar for -.98 PF
       //   Q := S.im * 0.001 +  0.063341 * S.re * 0.001;  // kvar for -.998 PF

          With pControlVars^ Do
          CASE PresentState of
              CTRL_OPEN:   IF Q > ON_Value
                      THEN  Begin
                            FPendingChange := CTRL_CLOSE;
                            ShouldSwitch := TRUE;
                      End
                      ELSE // Reset
                            FPendingChange := CTRL_NONE;
              CTRL_CLOSE:  IF Q < OFF_Value
                      THEN Begin
                             FPendingChange := CTRL_OPEN;
                             ShouldSwitch := TRUE;
                      End
                      ELSE IF AvailableSteps > 0 Then Begin
                          IF Q > ON_Value Then Begin
                            FPendingChange := CTRL_CLOSE;  // We can go some more
                            ShouldSwitch := TRUE;
                          End;
                      End
                      ELSE // Reset
                            FPendingChange := CTRL_NONE;
          END;

          If DebugTrace Then  WriteTraceRecordSample;


     End;

end;
```

This procedure first retrieves the SampleP value that is computed by the parent Sample routine before calling this one. SampleP can be found in the public data structure. The XYZ company custom capacitor control attempts to hold a specific power factor close to unity by switching capacitors. Setting the variable "ShouldSwitch" will force the main CapControl.Sample routine to push a control action onto the control queue. The DoPending function (below) will execute the control action and inhibit further switching for 15 minutes after making a switching action. This is the unique behavior not represented in the main CapControl module.

Messages understood by the CapControl are from the **EControlAction** enumeration (CTRL_NONE, CTRL_OPEN, and CTRL_CLOSE – ordinals 0, 1, and 2, respectively). A CLOSE message will cause the capacitor to add a step. An OPEN message will cause the controlled capacitor to subtract a step. If there is only one step, the entire capacitor object is switched either in or out.

The CapControl is "armed" when the message is pushed onto the control queue, but doesn't actually take action until the message is popped off the queue some time later. If in the meantime, the control action switches to CTRL_NONE, the control is disarmed and reset and will not act when the message is popped off.

DoPending Function

This is the function that processes the action message that gets popped off the control queue after being pushed onto the queue by a previous Sample function call. It is called from the *DoPendingAction* procedure in the CapControl.pas module as the first step taken as follows:

```
case ControlType of
   USERCONTROL: If UserModel.Exists Then UserModel.DoPending(Code, ProxyHdl);
end;
```

The USERCONTROL option which designates that the CapControl is described by a user-written DLL is the first option trapped in the procedure. It passes the action code and the ProxyHdl variable to the *DoPending* function of the user-written control for further processing. (Note: ProxyHdl is not used in this example.) This custom capacitor control inhibits further switching for 15 min (900 s) after switching the capacitor unless there is a voltage override.

```
Procedure   DoPending(Var Code, ProxyHdl:Integer); stdcall; // Integrates any state vars
{Do the capacitor switching action}
Begin

   If ActiveModel <> Nil Then  ActiveModel.DoPending(Code, ProxyHdl);

End;
```

The *DoPending* function in the main DLL interface then calls the *DoPending* function in the specific user model that is the active model. It is implemented as follows:

```
Procedure TXYZCapControlModel.DoPending(Var Code, ProxyHdl:Integer);
{----------------------------------------------------------------------}
Var
   inthour : Integer;

Begin

     // allows one capacitor to switch every 15 min
       {XYZInhibit is a module variable and if any XYZ capcontrol sets it true all other
XYZ CapControls will simply exit}
        If XYZInhibit Then

          Begin
               If Code=XYZINHIBITRELEASE Then
               Begin
                 XYZInhibit := FALSE;
                 Exit;  // Without doing anything; just process the inhibit release if
sent
               End Else
               Begin
                   // If it is a Voverride event just process the pending change
                   // but leave the inhibit on
                   // If not, need to remove the Armed switch so capcontrol  can sample
and
                   // send the message again.
                   With pControlVars^ Do
                      If not VoverrideEvent Then
                      Begin
                           ShouldSwitch := FALSE;
                           Armed        := FALSE;   // reset control
                           Exit;  // don't do anything; just send it back
                      End;
               End;
          End

        Else

          If (Code=Integer(CTRL_OPEN)) or (Code=Integer(CTRL_CLOSE)) Then   // skip NONE

          Begin
         {We'll switch capacitor this time, but then not again until inhibit released}
// Prevent further switching until inhibit released in 15 min
               XYZInhibit := TRUE;
               With CallBack^ Do
               Begin
                  intHour := Trunc(GetTimeHr);
                  XYZControlActionHandle := ControlQueuePush(intHour, GetTimeSec + 900.0
, XYZINHIBITRELEASE, 0, Fowner);
               End;
          End;

End;
```

Edit Function Implementation

This is the function that receives the string when the “userdata=(mystring)” property of the CapControl is processed. It loads up the Parser, which is a copy of the OpenDSS parser compiled into the DLL in this example. Then is calls the model’s local Edit function.

```
Procedure Edit(s:pAnsichar; Maxlen:Cardinal); Stdcall; // receive string from OpenDSS to
handle

Begin

  If ActiveModel <> Nil Then
    Begin
         ModelParser.CmdString := String(S);  // Load up Parser
         ActiveModel.Edit;     {Interpret string}
    End;

End;
```

The local Edit function in this example is implemented as follows:

```
procedure TXYZCapControlModel.Edit;
{-----------------------------------------------------------------------------------}

VAR
   ParamPointer:Integer;
   ParamName:String;
   Param:String;

begin
{This DLL has a version of the DSS Parser compiled into it directly because it
 was written on the same platform as the DSS. Otherwise, one should use the Callbacks.}

     ParamPointer := 0;
     ParamName := ModelParser.NextParam;
     Param := ModelParser.StrValue;
     WHILE Length(Param)>0 DO BEGIN
         IF Length(ParamName) = 0 THEN Begin
           If Comparetext(Param, 'help')=0 then ParamPointer := 2 Else Inc(ParamPointer);
         End
         ELSE ParamPointer := CommandList.GetCommand(ParamName);

         CASE ParamPointer OF
           // 0: DoSimpleMsg('Unknown parameter "'+ParamName+'" for Object "'+Name+'"');
            1: PF := ModelParser.DblValue;
            2: DoHelpCmd;     // whatever the option, do help
         ELSE
         END;

         ParamName := ModelParser.NextParam;
         Param := ModelParser.StrValue;
     END;

     RecalcElementData;

end;
```

This particular implementation of a CapControl user model requires only one property: the target PF. That is captured as CASE 1 above. This Edit procedure uses the OpenDSS Parser and the Command interpreter (CommandList) to decipher the string. It also has a Help command that is processed by the DoHelpCmd function (CASE 2).