OpenDSS Documentation

Storage Model Dynamics DLL (DynaDLL)

Sept, 5, 2012

The OpenDSS Storage element model supports two types of user-written DLLS: one general user-written model and one for Dynamics mode only. The latter is described in this document. It is for cases where the standard Storage model is adequate for the other solution modes but the default Thevenin equivalent model is inadequate for dynamics simulations.

An example of where this model is useful is a recent case where a DLL was written to model the electronic control of the storage element in great detail.

Public Data Structure

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:

```
{Struct to pass basic data to user-written DLLs}
   TStorageVars = Packed Record

        kWrating        :double;
        kWhRating       :Double;
        kWhStored       :Double;
        kWhReserve      :Double;
        ChargeEff       :Double;
        DisChargeEff    :Double;
        kVArating       :Double;
        kVStorageBase   :Double;
        kvarRequested   :Double;
        RThev           :Double;
        XThev           :Double;
        // Dynamics variables
        Vthev           :Complex;  {Thevenin equivalent voltage (complex) for dynamic
model}
        ZThev           :Complex;
        Vthevharm       :Double;  {Thevenin equivalent voltage mag for Harmonic model}
        Thetaharm       :Double;  {Thevenin equivalent voltage angle reference for
Harmonic model}
        VthevMag        :Double;    {Thevenin equivalent voltage for dynamic model}
        Theta           :Double;   {Power angle between voltage and current}
        w_grid          :Double;   {Grid frequency}
        TotalLosses     :Double;
        IdlingLosses    :Double;

                {32-bit integers}
        NumPhases,       {Number of phases}
        NumConductors,   {Total Number of conductors (wye-connected will have 4)}
        Conn           :Integer;   // 0 = wye; 1 = Delta

   End;
```

Public Data Variable Definitions

kWrating:double;

Rated power output for the Storage element, kW.

kWhRating :Double;

Rated energy storage capacity, kWh.

kWhStored :Double;

Present amount of energy in the storage element, kWh.

kWhReserve :Double;

Reserve level of storage element, kWh. The storage element would generally not be discharged below this level.

ChargeEff :Double;

Charging efficiency, pu.

DisChargeEff :Double;

Discharging efficiency, pu.

kVArating :Double;

Voltampere rating of inverter, kVA.

kVStorageBase:Double;

Base voltage for the Storage element, kV L-L or 1-phase kV.

kvarRequested:Double;

Reactive power output requested, kvar. Could be + or -.

RThev :Double;

Equivalent resistance used for simple Thevenin equivalent, ohms.

XThev :Double;

Equivalent reactance used for simple Thevenin equivalent, ohms.

// Dynamics variables

Vthev :Complex;

Thevenin equivalent voltage (complex) for default dynamic model.

ZThev :Complex;

Thevenin impedance used for default dynamics model.

Vthevharm :Double;

Thevenin equivalent voltage magnitudefor Harmonic mode model.

Thetaharm :Double;

Thevenin equivalent voltage angle reference for Harmonic model.

VthevMag:Double;

Thevenin equivalent voltage magnitude for dynamic model.

Theta :Double;

Power angle between voltage and current for dynamic model.

w_grid:Double;

Grid frequency, radians/s.

TotalLosses:Double;

Present value of total losses in Storage element.

IdlingLosses :Double;

Idling losses in Storage element

{The following are 32-bit integers}

NumPhases,

Number of phases in the Storage element terminal (only one terminal).

NumConductors,

Total Number of conductors in the Storage element terminal. 1-phase elements always have 2. If wye-connected, 3-phase, there will be 4.

Conn :Integer

Terminal connection. 0 = Wye (or L-N); 1 = Delta (or L-L)

DLL Function Interface

The declaration of the functions and procedures in the DLL interface for a user-written DLL is for dynamics simulation only. The name of the DLL is specified by the *DynaDLL=MyUserWrittenDLL* statement in the description of the Storage element. The OpenDSS will then load the DLL and attempt to use it. All the functions described here will have to be found before the OpenDSS will continue.

User-written DLLs should be capable of supporting more than one instance of a model. Each of several Storage elements in the OpenDSS circuit model could invoke the same DLL. The DLL is only loaded one time, but could be called by more than one Storage 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 StoreUserModel.Pas file in the ..\Source\PCelement folder on the source code sharing site.

Function New

(Var DynaData : TDynamicsRec;
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 instance that may be used to specifically select this instance at a later time.

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 Storage object that creates a DynaDLL 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:

DynaData: The TDynamicsRec structure is passed by reference. This contains the time variables for a typical dynamics simulation. This is defined as follows.

```
   TDynamicsRec = Packed Record

       h,     // Time step size in sec for dynamics
       t,     // sec from top of hour
       tstart,
       tstop:Double;
       IterationFlag:Integer;
{0=New Time Step; 1= Same Time Step as last iteration}
       SolutionMode :Integer;
       intHour :Integer;  // time, in hours as an integer
       dblHour :Double;   // time, in hours as a floating point
number including fractional part

   End;
```

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 procedure in it to obtain specific data 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. This 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 retuns the ID as confirmation. If the return value is 0, there has been an error.

Procedure Init

(V, I:pComplexArray);Stdcall;

This procedure is called when the OpenDSS is entering Dynamics mode from one of the power flow modes. Two pointers to complex number arrays are passed representing the present voltage, V, and current, I, at the Storage element terminals. The user-written DLL uses these values to perform whatever calculations are required to initialize the active model.

Procedure Calc

(V, I:pComplexArray); stdcall;

This procedure invokes the main electrical calculation algorithm of the model. This will nearly always be to compute the terminal currents, I, given the present estimate of the voltages, V, and internal state variables.

V and I are pointers to complex number arrays. There will be one complex value for each conductor of the Storage element terminal. If there is any question how many conductors are present, the **GetActiveElementTerminalInfo** call back function may be called or, for the Storage element, the values are available on the public data structure.

Procedure Integrate;

stdcall; // Integrates any state vars

This procedure is called by the OpenDSS in Dynamics mode when it is time to integrate the state variables if the model uses differential equations.

OpenDSS currently uses a two step predictor-corrector integration process. The IterationFlag variable of the TDynamicsRec structure indicates with step is currently being executed. If IterationFlag=0, the predictor step is being executed. If it is =1 the corrector step is being executed.

Most other models in the OpenDSS use a forward Euler for the predictor step and trapezoidal for the corrector step. The derivatives from the previous time step – necessary for the trapezoidal integration step – are captured during the predictor step.

Procedure Edit

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

The Edit procedure is called when the OpenDSS encounters the DynaData=(*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. The callbacks to the OpenDSS parser may be used to help interpret the string if an OpenDSS-like syntax is used.

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 expect 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.

MONITOR VARIABLES / STATE VARIABLES

The following API functions in the DLL allow the user to get and set selected variables from within the model as determined by the programmer implementing the DLL.

The values returned by these functions are automatically captured by **Monitor** elements assigned to Power Conversion (PC) elements and defined with **Mode=3**. The variables exposed by the programmer are appended to the default list of variables in the parent PC element (i.e., a Storage element). A Monitor will put the names of the variables in the header record and then capture the numeric values for each sample.

The user may return a number of double-precision floating point values for monitoring. These are frequently values of the state variables, but can be any number the programmer desires.

Function NumVars:

Integer;Stdcall; // Number of variables that can be returned for monitoring

This function returns the number of double-precision variables that can be accessed.

Procedure GetAllVars

(Vars:pDoubleArray);StdCall;

This Procedure is called by OpenDSS monitor elements and is expected to return the values of all the exposed model variables in an array of doubles. The OpenDSS will allocate the necessary space and pass a pointer to the buffer in the Vars argument.

Function GetVariable

(var i:Integer):double;StdCall; // returns a i-th variable value only

This function returns the value of the *i*-th variable. Note that *i* is passed by reference, That is, the user-written program should expect to receive a pointer to this index.

Procedure SetVariable

(var i:Integer;var value:Double); StdCall;

This procedure is provided to allow the user to set the value of a variable internal to the model. Both arguments are passed by reference. That is, the user-written program should expect to receive pointers to the values of these arguments.

The DLL is not obligated to set the internal variable to the value and should not permit it if it will break the model.

Procedure GetVarName

(var VarNum:Integer; VarName:pAnsiChar; maxlen:Cardinal); StdCall;

This procedure is provided to obtain the name assigned to a monitor or state variable. Monitor elements defined as Mode=3 will automatically query the names using this function.

VarNum is the index of the variable name requested and will be passed by reference (i.e., expect a pointer to an integer).

The OpenDSS will allocate space for *VarName*, a null-terminated Ansi character string, and pass the size of the allocation in the *maxlen* argument, which is an unsigned integer passed by value. The DLL implementation of this procedure will copy the string name of the requested variable into the buffer up to the limit given by *maxlen*.

Example

Code snippets are provided from an actual DynaDLL implementation 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 Implementataion

```
Function  New(Var DynaData:TDynamicsRec; Var CallBacks:TDSSCallBacks): Integer;
Stdcall;// Make a new instance
Begin
    ActiveModel := TDESS.Create(DynaData, CallBacks); // calls TDESS constructor
// keep track of instance by adding to a pointer list
// handle is simply the index to the list
    Result := ModelList.Add(ActiveModel)+1;
End;

… The TDESS constructor:


constructor TDESS.Create(var DynaData:TDynamicsRec; var CallBacks:TDSSCallBacks);
VAr
   PublicDataSize : Integer;
   S : AnsiString;
   DataPtr : Pointer;
   NameBuffer : Array[1..255] of AnsiChar;
   pNameBuffer : pAnsiChar;

begin

{Device Characteristics}


{Some constantsa}
         Tcpll := 0.04;
         Kcpll := -0.251;
         TcV   := 0.04;
         KcV   := 1.012;
         Tcp   := -0.06;
         Kcp   := -0.6;


        // …. etc.

         // Keep pointers to Public Data

         FCallBacks := @CallBacks;
         FDynaData  := @DynaData;

// Make sure CallBacks Pointer is valid and then get a pointer to the
// public data structure. When this gets called, the parent Storage
// element is the active circuit element.

         If Assigned(FCallBacks) Then
         Begin
               FCallBacks^.GetPublicDataPtr(DataPtr, PublicDataSize);
               DESSStorageVars := DataPtr;

// Uer OpenDSS message facility to send warning if the public data structure
// size is different than we think it should be.

               If PublicDataSize <> SizeOf(DESSStorageVars^) Then
               Begin
                     S := 'Warning: Mismatch in DESS Public Data Sizes';
                     FCallBacks^.MsgCallBack(PAnsichar(S), Length(S));
               End;
               // Get Element Name – might come in handy later
               pNameBuffer := @NameBuffer;
               FCallBacks^.GetActiveElementName(pNameBuffer, 255);
               FElementName := AnsiString(pNameBuffer);
         End;

        If DebugTrace Then InitTraceFile;
```

end;

Select Function Implementation

This is a key function that is used by OpenDSS to select a particular instance of objects that this DLL supports.

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

Calc Function Implementation

```
Procedure  Calc(V, I:pComplexArray); stdcall; // returns voltage or torque
{
 Perform calculations related to active model
}
Begin
  If ActiveModel <> Nil Then ActiveModel.CalcCurrents(V, I);
End;

// Here’s the implementation of CalcCurrents. Note variable change

procedure TDESS.CalcCurrents(V, Curr: pComplexArray);

Var
    Curr1   : Complex;
    Alpha   : Complex;
    Rotater : Complex;
    i       : Integer;

begin

     case DESSStorageVars^.Numphases of
        1: U := CSub(V^[1], V^[2]);  // Assume two conductors
     ELSE
           U := V^[1];           // Just 1st phase  for now
     end;

     Omega_Grid := DESSStorageVars^.w_grid;  // Get the present grid freq

     DoDESSActivePController;
     DoStorageDevice;
       […etc….]
     DoInverter;

     // Convert Ird and Irq to terminal currents
     Curr1 := RotateI;

     case DESSStorageVars^.Numphases of
        1: Begin
              Curr^[1] := Curr1;
              Curr^[2] := Cnegate(Curr1);
           End
     ELSE
       // assumes 3-phase  -- Shift the 3 currents 120 degrees in pos seq
        Alpha := cmplx(-0.5,-0.866025403);
        Rotater := Cmplx(1.0, 0.0);
        For i := 1 to 3 Do Begin
            Curr^[i] := Cmul(Curr1, Rotater);
            Rotater := Cmul(Rotater, Alpha);
        End;
     end;

// this is a pointer to the kWhStored value in the public data structure
// Updates the storage kWh

     FkWh_Stored^ := 0.5 * Csc * SQR(VC)  ;

     If DebugTrace Then WriteTraceRecord;
end;
```

Integration Function

The variables with a “d” are derivatives of state variables computed during the current calculations above (not shown). The main formula here is for trapezoidal integration. However, by shifting the last computed value into the “old”values (with “_n” suffix), the first pass through this essentially turns the formula into a forward Euler. This is the general process in OpenDSS. The integration takes place prior to the current calculations. (See SolveDynamic procedure in SolutionAlgs.Pas.)

```
procedure TDESS.Integrate;
Var
   dt2 :Double;
begin

    If FDynadata^.IterationFlag = 0 Then
    Begin
         // First iteration of a new time step (Predictor)
         // Save states and derivatives
         dIrd_n       := dIrd      ;
         dIrq_n       := dIrq      ;
…
         dIsc_n       := dIsc      ;
         dVc_n        := dVc      ;

          Ird_n        :=  Ird      ;
          Irq_n        :=  Irq      ;
…
          Isc_n        :=  Isc      ;
          Vc_n         :=  Vc      ;

    End;

    dt2 := FDynaData^.h/2.0;  // for trapezoidal rule

   {Integrate the state variables - trapezoidal formula}
      Ird        :=   Ird_n       + dt2 * ( dIrd       + dIrd_n       )  ;
      Irq        :=   Irq_n       + dt2 * ( dIrq       + dIrq_n       )  ;
…
      Isc        :=   Isc_n       + dt2 * ( dIsc       + dIsc_n       )  ;
      Vc         :=   Vc_n        + dt2 * ( dVc        + dVc_n        )  ;

end;
```