Adding User-Written DLL to Generator

For OpenDSS Version 9,   
March 19, 2022

The IndMach012a.DLL will be used to illustrate the procedure of adding a user-written DLL under the Generator model. The source code for this DLL may be downloaded from Sourceforge.net at the following URL:

https://sourceforge.net/p/electricdss/code/HEAD/tree/trunk/Version8/Source/IndMach012a

The API functions in the *Usermodel* interface of the Generator model in OpenDSS that must be provided are:

**Instantiation Functions**

New,

Delete,

Select,

**Initialization and Calculation Functions**

Init,

Calc,

Integrate,

Save,

Restore,

Edit,

UpdateModel,

**Management of State Variables and Other User-Defined Variable**

NumVars,

GetAllVars,

GetVariable,

SetVariable,

GetVarName;

The actual definition of the interface procedures and functions for IndMach012a.dll found in the file Mainunit.Pas are:

{Imports Generator Variables Structures and DSS structures from Dynamics}

{Note: everything is passed by reference (as a pointer), so it is possible to

change the values in the structures in the main program.

This is dangerous so be careful.}

Function **New**(Var GenVars:TGeneratorVars;

Var DynaData:TDynamicsRec;

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

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

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

Procedure **Init**(V, I:pComplexArray);Stdcall;

{Initialize model. Called when entering Dynamics mode.

V,I should contain results of most recent power flow solution.}

Procedure **Calc**(V, I:pComplexArray); stdcall;

{Main routine for performing the model calculations. For "usermodel", this

function basically computes I given V. For "shaftmodel", uses V and I

to calculate Pshaft, speed, etc. in dynamic data structures}

Procedure **Integrate**; stdcall; // Integrates any state vars

{Called to integrate state variables. User model is responsible for its own

integration. Check IterationFlag to determine if this

is a predictor or corrector step }

Procedure **Edit**(s:pAnsichar; Maxlen:Cardinal); Stdcall;

{called when DSS encounters user-supplied data string. This module

is reponsible for interpreting whatever format this user-written modeli

is designed for.}

Procedure **UpdateModel**; StdCall;

{This is called when DSS needs to update the data that is computed

from user-supplied data forms. }

Procedure **Save**; Stdcall;

{Save the model to a file (of the programmer's choice) so that the state

data, if any can be restored for a restart.}

Procedure **Restore**; Stdcall;

{Reverse of the Save command}

{The user may return a number of double-precision values for monitoring}

Function **NumVars**:Integer;Stdcall; // Returns number of variables that are defined

Procedure **GetAllVars**(Vars:pDoubleArray);StdCall;

{Called by DSS monitoring elements.

Returns values of all monitoring variables in

an array of doubles. The DSS will allocate "Vars" to the appropriate size.

This routine will use Vars as a pointer to the array.}

Function **GetVariable**(var i:Integer):double;StdCall; // returns the i-th variable value

Procedure **SetVariable**(var i:Integer;var value:Double); StdCall;

{OpenDSS allows users to set variables of user-written models directly.

Whatever variables that are exposed can be set if this routine handles it}

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

{Returns name of a specific variable as a pointer to an ANSI string.

Set VarName= a pointer to the first character in a null-terminated string

User-written DLLs in OpenDSS are loaded by the parent device model code rather than the main program. They are specified in the regular DSS script, in this case during the definition of a Generator class device. There could be a different API for each class of device that supports DLLs. The Storage model has a similar interface, but it is slightly different than the Generator element. The Storage model also has a separate API for Dynamics mode simulations.

This API has a Delphi Pascal flavor to it, but the DLL can be written in any language capable of producing a standard DLL. Note that *Function* types return a value while *Procedure* types do not. All arguments, except for the Maxlen argument for string types, are passed by reference (i.e., as a pointer). Thus, the interface can be implemented in a wide variety of languages, but programmers must exercise discipline not to overwrite memory. Poorly written code can crash the program, although OpenDSS has error trapping active and will generally successfully report where the error came from before it halts.

OpenDSS comes in both 32-bit (X86) and 64-bit (X64) versions. The DLL must match the version of OpenDSS being used. On most installations, both versions are installed. While most applications will prefer 64-bit code these days, some instances of software that will be used to drive OpenDSS such as Microsoft Office might be installed as 32-bit. Thus, it is recommended that users develop the DLL using a compiler capable of producing both X86 and X64 executables.

A user-written mode DLL imports variables and structures from the Generator model and other OpenDSS structures from the program’s Dynamics module. The connection to the data structures in the parent model is made through the device’s public data pointer obtained from a callback function, GetPublicDataPtrCallBack. This allows data to be moved across the DLL boundary directly through memory.

The current Generator class public data structure, TGeneratorVars, is:

{Generator public data/state variable structure}

TGeneratorVars = packed Record

Theta, {Direct-Axis voltage magnitude & angle}

Pshaft, {present Shaft Power}

Speed, { relative Speed, difference from Synchronous speed, w0}

w0, {actual speed = Speed + w0}

Hmass, {Per unit mass constant}

Mmass, {Mass constant actual values (Joule-sec/rad}

D, Dpu, {Actual and per unit damping factors}

kVArating,

kVGeneratorBase,

Xd, Xdp, Xdpp, {machine Reactances, ohms}

puXd, puXdp, puXdpp, {machine Reactances, per unit}

dTheta,

dSpeed, {Derivatives of Theta and Speed}

ThetaHistory,

SpeedHistory, {history variables for integration}

Pnominalperphase,

Qnominalperphase {Target P and Q for power flow solution, watts, vars}

: Double; { All Doubles }

{32-bit integers}

NumPhases, {Number of phases}

NumConductors, {Total Number of conductors (wye-connected will have 4)}

Conn :Integer; // 0 = wye; 1 = Delta

{ Revisons (additions) to structure ...

Later additions are appended to end of the structure so that

previously compiled DLLs do not break

}

VthevMag : Double; {Thevinen equivalent voltage for dynamic model}

VThevHarm : Double; {Thevinen equivalent voltage mag reference for Harmonic model}

ThetaHarm : Double; {Thevinen equivalent voltage angle reference for Harmonic model}

VTarget : Double; // Target voltage for generator with voltage control

Zthev : Complex;

XRdp : Double; // Assumed X/R for Xd'

End;

## OpenDSS Code That Loads the DLL

The actual code for loading a Generator-class Usermodel DLL is shown below. It uses the standard Windows LoadLibrary function to load the DLL by name. The DLL is expected to be installed in the same folder where OpenDSS.exe is installed. After opening the library, OpenDSS attempts to find all the expected exported functions from the DLL. They all must exist for the process to proceed. The addresses in the DLL are resolved and assigned to Procedure/Function variables defined in the TGenUserModel class. This is how the OpenDSS Generator model knows how to call the functions in the user-written model.

FHandle := LoadLibrary(PChar(Value)); // Default LoadLibrary and PChar must agree in expected type

IF FHandle = 0 Then

Begin // Try again with full path name

FHandle := LoadLibrary(PChar(DSSDirectory + Value));

End;

If FHandle = 0 Then

DoSimpleMsg('Generator User Model ' + Value + ' Not Loaded. DSS Directory = '+DSSDirectory, 570)

Else

Begin

FName := Value;

// Now set up all the procedure variables

FuncError := False;

@Fnew := CheckFuncError(GetProcAddress(FHandle, 'New'), 'New');

If not FuncError Then @FSelect := CheckFuncError(GetProcAddress(FHandle, 'Select'), 'Select');

If not FuncError Then @FInit := CheckFuncError(GetProcAddress(FHandle, 'Init'), 'Init');

If not FuncError Then @FCalc := CheckFuncError(GetProcAddress(FHandle, 'Calc'), 'Calc');

If not FuncError Then @FIntegrate := CheckFuncError(GetProcAddress(FHandle, 'Integrate'), Integrate');

If not FuncError Then @FSave := CheckFuncError(GetProcAddress(FHandle, 'Save'), 'Save');

If not FuncError Then @FRestore := CheckFuncError(GetProcAddress(FHandle, 'Restore'), 'Restore');

If not FuncError Then @FEdit := CheckFuncError(GetProcAddress(FHandle, 'Edit'), 'Edit');

If not FuncError Then @FUpdateModel := CheckFuncError(GetProcAddress(FHandle, 'UpdateModel'),   
 UpdateModel');

If not FuncError Then @FDelete := CheckFuncError(GetProcAddress(FHandle, 'Delete'), 'Delete');

If not FuncError Then @FNumVars := CheckFuncError(GetProcAddress(FHandle, 'NumVars'), 'NumVars');

If not FuncError Then @FGetAllVars := CheckFuncError(GetProcAddress(FHandle, 'GetAllVars'),

'GetAllVars');

If not FuncError Then @FGetVariable := CheckFuncError(GetProcAddress(FHandle, 'GetVariable'),

'GetVariable');

If not FuncError Then @FSetVariable := CheckFuncError(GetProcAddress(FHandle, 'SetVariable'),

'SetVariable');

If not FuncError Then @FGetVarName := CheckFuncError(GetProcAddress(FHandle, 'GetVarName'),

'GetVarName');

If FuncError Then Begin

FreeLibrary(FHandle);

FID := 0;

FHandle := 0;

FName := '';

end

Else Begin

FID := FNew(FActiveGeneratorVars^, ActiveCircuit[ActiveActor].Solution.Dynavars, CallBackRoutines); // Create new instance of user model

End;;

End

The general usage of the API functions is as follows:

1. The *New* function creates of new instance of the custom storage model. Pointers to the required dynamics data and callback routines are provided. An ID is returned that is subsequently used to select this instance of the model.
2. The *Edit* procedure is called to set the values of the model parameters. Since there could be many parameters in the storage controller for dynamics, the values may be read in from a text file.
3. The *Select* procedure sets the current model active. The DLL should support multiple instances of the user model and have a mechanism for keeping track of the instances. This can be a simple array or some kind of a linked list. *Select* has an integer argument, ID, that is assigned when the New function is executed. It is stored in the instance of the user model that is connected to a Generator object.
4. The *Init* procedure is called to initialize the state variables of the dynamics model from the voltages and currents computed from the OpenDSS circuit solver for the initial steady-state condition.
5. The *Calc* procedure computes the current given the present values of the terminal voltages and the values of the state variables. This is called for each iteration of the solution process.
6. The *Integrate* function is called when the Dynamics simulation performs integration of the state variables. This will be called twice for each iteration: the programmer must implement both a predictor and a corrector step.
7. *UpdateModel* is called infrequently to synchronize model parameters with the base Storage element model in OpenDSS.

There are, of course, many more details for DLL developers to understand that the developers may obtain by inspecting the actual source code. The OpenDSS Dynamics mode algorithm is described in the next section.