OpenDSS CallBack Routines

Version 7.6.1, 31-Aug-2012

The OpenDSS creates a structure of Function/Procedure pointers that have been used to support the various user-written DLLs to execute selected functions within OpenDSS and to access certain types of data more quickly. It is expected that this structure will grow over time as more needs are identified. The definition of the structure may be found in **DSSCallbackStructDef.Pas** in the Source\Common folder. A compacted listing of the file is shown below

```
TYPE

 {NOTE: Maxlen argument is to better accommodate Fortran strings. VB also}
 { Caller must allocate space for pchar values }

pDSSCallBacks = ^TDSSCallBacks; {Pointer to callback structure}
TDSSCallBacks = Packed Record

MsgCallBack: Procedure (S:pAnsiChar; Maxlen:Cardinal);Stdcall;

{Routines for using OpenDSS Parser. This allows you to write models that accept
 syntax like other OpenDSS scripts.}
GetIntValue: Procedure(var i:Integer);Stdcall;
GetDblValue: Procedure(var x:Double); Stdcall;
GetStrValue: Procedure(S: pAnsiChar; maxlen:Cardinal); Stdcall;
LoadParser: Procedure(S: pAnsiChar; maxlen:Cardinal); Stdcall;
NextParam: Function (ParamName:pAnsiChar; Maxlen:Cardinal):Integer; Stdcall;

DoDSSCommand: Procedure(S: pAnsiChar; Maxlen:Cardinal); StdCall;
GetActiveElementBusNames: Procedure(Name1:pAnsiChar; Len1:Cardinal; Name2:pAnsiChar;
Len2:Cardinal); StdCall;
GetActiveElementVoltages: Procedure(Var NumVoltages:Integer; V:pComplexArray); StdCall;
GetActiveElementCurrents: Procedure(Var NumCurrents:Integer; Curr:pComplexArray); StdCall;
GetActiveElementLosses: Procedure(Var TotalLosses, LoadLosses, NoLoadLosses:Complex); StdCall;
GetActiveElementPower: Procedure(Terminal:Integer; Var TotalPower:Complex); StdCall;
GetActiveElementNumCust: Procedure(Var NumCust, TotalCust:Integer); StdCall;
GetActiveElementNodeRef: Procedure(Maxsize:Integer; NodeReferenceArray:pIntegerArray); StdCall
GetActiveElementBusRef: Function(Terminal:Integer)Integer; StdCall;
GetActiveElementTerminalInfo: Procedure(Var NumTerminals, NumConds, NumPhases:Integer); StdCall;
GetPtrToSystemVarray: Procedure(var V:Pointer; var iNumNodes:Integer); StdCall;
GetActiveElementIndex: Function():Integer; StdCall;
IsActiveElementEnabled: Function():Boolean; StdCall;
IsBusCoordinateDefined: Function(BusRef:Integer) :Boolean; StdCall;
GetBusCoordinate: Procedure(BusRef:Integer; Var X, Y:Double); StdCall;
GetBuskVBase: Function(BusRef:Integer):Double; StdCall;
GetBusDistFromMeter: Function(BusRef:Integer) :Double; StdCall;

GetDynamicsStruct:Procedure(var pDynamicsStruct:Pointer); StdCall;
GetStepSize: Function():Double; StdCall;
GetTimeSec: Function():Double; StdCall;
GetTimeHr: Function():Double; StdCall;

GetPublicDataPtr: Procedure(var pPublicData:Pointer; Var PublicDataBytes:Integer); StdCall;
GetActiveElementName: Function(FullName:pAnsiChar; MaxNameLen:Cardinal) :Integer; StdCall;
GetActiveElementPtr: Function():Pointer; StdCall;
ControlQueuePush: Function(Const Hour:Integer; Const Sec:Double; Const Code, ProxyHdl:Integer;
Owner:Pointer):Integer; StdCall;
GetResultStr: Procedure(S:pAnsiChar; Maxlen:Cardinal); StdCall;

End;
```

All routines use the **stdcall** calling convention. This is the common calling convention of the Windows API and commonly used for DLLs. It passes all arguments right-to-left on the stack and the called routine cleans up the stack upon return. Note that most functions internal to OpenDSS use the Delphi default **register** calling convention, which passes arguments left-to-right, with the first 3 arguments passed in CPU registers.

A *packed* record means that the pointers to the procedures and functions are packed tightly in memory. The type of the structure is named **TDSSCallBacks**.

Future revisions of OpenDSS may require additions to this set of functions. Our plan is to add function/procedure pointers to the bottom of the structure so that user-written software will continue to operate.

The usual process in user-written software is to assign a pointer to this structure. For example, the user-written DLL will implement a *New* function for creating a new instance of an object modeled by the DLL. In Pascal, this might be declared as follows:

```
Function New( Var DynaData : TDynamicsRec; Var CallBacks : TDSSCallBacks) : Integer;  Stdcall
```

The TDSSCallBacks structure is passed as an argument to the New function by reference. That is, a pointer to the structure is pushed onto the stack. The user implementation would typically keep a copy of the pointer for use by other functions in the API. Functions/procedures are accessed using the name of a field in the structure followed by the arguments just like a conventional function call. For example, to use the MsgCallback procedure, the code might look something like this:

```
Var  S : AnsiString;
…
        S := 'Warning: Mismatch in DESS Public Data Sizes';
        FMyCallBacksPointer^.MsgCallBack(PAnsichar(S), Length(S));
```

This assumes that the module has saved a pointer to the callbacks structure named FMyCallBacksPointer in persistent memory somewhere. The AnsiString variable S is typecast to a PAnsiChar, which is a pointer to the first character in the string. The second argument is the length. If you are writing in a language that automatically pushes the length onto the stack, you would not need to explicitly supply this. Be sure to understand how your compiler handles calling conventions for strings. OpenDSS doesn’t always need this for interpreting the string, but it may be required to make the stack arguments align correctly. See next section.

Strings

All strings in this version are passed as pointers to null-terminated Ansi strings. This is referred to as a **pAnsiChar** type. To accommodate languages such as Fortran that will push an unsigned integer (**Cardinal**) representing the length of the string onto the stack, functions expecting strings accept two arguments: a pointer to the string and a Cardinal variable representing the length. Typically, the OpenDSS assumes the length represents either the actual length of the string or the amount of memory allocated for the string.

Other Special Types

Arguments with a **var** qualifiers are passed on the stack by reference. Variables passed by reference are likely to be altered. Otherwise, arguments are passed by value and any changes are discarded upon return. Note that passing a pointer variable by value, which is done frequently for array types, is essentially the same as passing the array by reference. Dereferencing the variable yields the first element in the array. The pointer is not altered in the called routine, but the values it points to could be changed. A **const** qualifier also passes by reference, but the Delphi Pascal compiler will prevent code in the called function from changing the value. We included the const qualifier in the definition to let you know that the OpenDSS will not be changing the memory referenced by the variable even though the variable is passed by reference.

Complex: A packed structure of two doubles representing the real and imaginary values of a complex number.

Pointer: An untyped pointer to a block of memory. Used to return pointers to selected structures.

pComplexArray: Pointer to an array of Complex numbers.

pIntegerArray: Pointer to an array of Integer numbers. **Integer** is expected to be a 32-bit integer.

Function Definitions

MsgCallBack:

Procedure (S:pAnsiChar; Maxlen:Cardinal);Stdcall;

This procedure provides access to the internal *SimpleMsg* function in the OpenDSS so that you don't have to provide your own mechanism for messaging. This will provide a mechanism for returning messages on the Result property of the COM interface as well as popping up a message window when forms are allowed.

Routines for using OpenDSS Parser:

The next 5 routines provide access to the OpenDSS command parser. This allows you to write models that accept syntax in OpenDSS scripting format.

LoadParser:

Procedure(S: pAnsiChar; maxlen:Cardinal); Stdcall;

This procedure is used to load the OpenDSS parser with a string. Define a null-terminated string of Ansi characters, S. Then pass a pointer to the first character in S and the length of the string.

NextParam:

Function (ParamName:pAnsiChar; Maxlen:Cardinal):Integer; Stdcall;

This function causes the OpenDSS parser to advance to the next token (parameter) on the script string that was previously loaded into the parser. If the syntax

```
ParamName = Token
```

is encountered, the ParamName string is returned in the first argument. You have to allocate space to receive this string and tell the OpenDSS how much space you have allocated in the Maxlen argument so that it doesn't overwrite the buffer.

The result of the function call is the length of the token found, which you can use to allocate space for receiving the token through the GetStrValue function or simply to test whether or not a token was found.

Note that in OpenDSS, ParamName does not have to be specified, so test for a null string does not mean that there is no token.

GetIntValue:

Procedure(var i:Integer);Stdcall;

Returns the value of the present token as an Integer type in the variable i.

GetDblValue:

Procedure(var x:Double); Stdcall;

Returns the value of the present token as a Double type in the variable x.

GetStrValue:

Procedure(S: pAnsiChar; Maxlen:Cardinal); Stdcall;

Returns the present token as an Ansi string. You have to allocate space for S including the null terminator and set Maxlen to the size of the allocated space so that the OpenDSS parser does not overwrite memory.

End of Routines for using OpenDSS Parser:

DoDSSCommand:

Procedure(S: pAnsiChar; Maxlen:Cardinal); StdCall;

This procedure sends a string to the OpenDSS main command processor. The effect is the same as if the string were entered through any other means (COM interface, script window, or file).

One effective use of this procedure is to use the “Select” command to set the active circuit element for use by the routines defined below. For example,

```
MyCallBacks.DoDSSCommand(pAnsiChar(‘Select Line.Line1’), 255);
```

Another useful application of this procedure is to query values of various properties with the “?” command. The value is returned in the global result string that is retrieved with the **GetResultStr** procedure (see below). This is the same result string that can be retrieved through the COM interface (DSSobj.Text.Result).

```
MyCallBacks.DoDSSCommand(pAnsiChar(‘? Line.Line1.Length’), 255);
MyCallBacks.GetResultStr(MyResultStrBuffer, MyResultBufferSize);
```

GetActiveElementBusNames:

Procedure(Name1:pAnsiChar; Len1:Cardinal; Name2:pAnsiChar; Len2:Cardinal); StdCall;

Returns the first two buses to which the active circuit element is connected. This is mostly used for LINE objects. Will not give all bus connections for TRANSFORMER elements with more than 2 windings.

GetActiveElementVoltages:

Procedure(Var NumVoltages:Integer; V:pComplexArray); StdCall;

This procedure returns an array of complex numbers for all the voltages at each conductor of each terminal of the active circuit element. Make sure the buffer V is big enough to handle the expected array. On calling pass the size of the allocation (number of voltages) in the NumVoltages variable. On return it will be set to the actual number returned. If the actual number is greater than the size of the buffer, the OpenDSS will return only enough voltages to fill the buffer to prevent buffer overrun.

GetActiveElementCurrents:
Procedure(Var NumCurrents:Integer; Curr:pComplexArray); StdCall;

This procedure returns an array of complex numbers for all the current going INTO each conductor of each terminal of the active circuit element. Make sure the buffer Curr is big enough to handle the expected array. On calling pass the size of the allocation (number of currents) in the NumCurrents variable. On return it will be set to the actual number returned. If the actual number is greater than the size of the buffer, the OpenDSS will return only enough currents to fill the buffer to prevent buffer overrun.

GetActiveElementLosses:
Procedure(Var TotalLosses, LoadLosses, NoLoadLosses:Complex); StdCall;

This procedure returns three complex numbers representing the active and reactive power losses for the active circuit element. The terminology is based on transformer losses. However, elements such as the REACTOR can also have no-load losses for certain models. For LINE objects, the NoLoadLosses are currently zero because the model does not have shunt conductance. Units: watts, vars

GetActiveElementPower:
Procedure(Terminal:Integer; Var TotalPower:Complex); StdCall;

Returns the total complex power (P + jQ) going INTO the specified terminal.

GetActiveElementNumCust:
Procedure(Var NumCust, TotalCust:Integer); StdCall;

Returns the number of customers connected to the active element and the total number of customers downline from the active circuit element. Requires an ENERGYMETER object at the head of the feeder to work properly.

GetActiveElementNodeRef:
Procedure(Maxsize:Integer; NodeReferenceArray:pIntegerArray); StdCall

Returns the internal node reference numbers for each conductor of each terminal in an integer array. You must allocate the space for the array before calling. Pass Maxsize to indicate the size of the array buffer. The node reference for a conductor is the index into the system voltage array for the node to which the conductor is connected (see **GetPtrToSystemVarray** below).

GetActiveElementBusRef:
Function(Terminal:Integer):Integer; StdCall;

This function returns the bus reference index (BusRef) for the specified terminal of the active circuit element. The BusRef value is used to retrieve bus coordinates, kV base, and distance from the meter.

GetActiveElementTerminalInfo:
Procedure(Var NumTerminals, NumConds, NumPhases:Integer); StdCall;

This procedure returns basic information about the terminals of the active circuit element (number of terminals, number of conductors per terminal, and number of phases).

GetPtrToSystemVarray:
Procedure(var V:Pointer; var iNumNodes:Integer); StdCall;

Returns a pointer to the internal system voltage array, V, from the actual OpenDSS system admittance matrix solution array. The array can be indexed by the NodeRef value (see **GetActiveElementNodeRef**). V is an array of complex numbers representing Node-to-Ground voltages. iNumNodes is an integer set to the number of nodes in the system equations when the procedure is called. This value can change and the memory pointed to by V can be reallocated. To be safe, execute this procedure call each time you need to access the voltages.

GetActiveElementIndex:

Function():Integer; StdCall;

Returns an index into the device class collection for the active circuit element. For example, if the element is a LINE, it is the index into the LINE class of elements. Presently, this is used mainly to check to see if this is >0 which indicates that the pointers to the element have been assigned. For future use, the index will be used as a faster way to index to a particular element than searching by name.

IsActiveElementEnabled:

Function():Boolean; StdCall;

This function is used to check if the active circuit element is actually enabled in the present circuit.

IsBusCoordinateDefined:

Function(BusRef:Integer) :Boolean; StdCall;

This Boolean function returns true if the coordinates for the referenced bus are defined.

GetBusCoordinate:

Procedure(BusRef:Integer; Var X, Y:Double); StdCall;

Returns the (X, Y) coordinates for the bus referenced by the BusRef variable. The variable may be retrieved from the **GetActiveElementBusRef** procedure.

GetBuskVBase:

Function(BusRef:Integer):Double; StdCall;

Returns the voltage base, in kV, for the bus referenced by the BusRef variable. Note that if this value is zero, it means the voltage base has not been defined.

GetBusDistFromMeter:

Function(BusRef:Integer) :Double; StdCall;

Returns the distance, in meters, from the referenced bus to the nearest upline ENERGYMETER object. Used for reporting bus location and for creating voltage profile plots, etc.

GetDynamicsStruct:

Procedure(var pDynamicsStruct:Pointer); StdCall;

A pointer to the DynaVars structure is typically supplied for the New function in a user-written DLL. However, it is sometimes convenient to have another means to get to the structure.

Returns a pointer to the DynaVars structure commonly used for dynamics simulations. It is also used for other solution modes in which time varies. The structure is currently defined as:

```
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;

    End;
```

Note that the time, *t*, in this record refers to seconds from an hour. This it is generally used for dynamics simulations of only a few seconds at a small time step of perhaps a millisecond. You can also get this value from **GetTimeSec** below. To get a Double value representing time in hours, use the **GetTimeHr** function below. Regardless of solution mode, the time step, *h*, is correct (in seconds). The *IterationFlag* variable is only used in Dynamics mode. *SolutionMode* is an integer constant representing the present solution mode. The constants are defined in the Dynamics unit. The values are currently:

```
{Solution modes}
  SNAPSHOT = 0;
  DAILYMODE = 1;
  YEARLYMODE = 2;
  MONTECARLO1 = 3;
  LOADDURATION1 = 4;
  PEAKDAY = 5;
  DUTYCYCLE = 6;
  DIRECT = 7;
  MONTEFAULT = 8;  // Monte Carlo Fault Study
  FAULTSTUDY = 9;
  MONTECARLO2 = 10;
  MONTECARLO3 = 11;
  LOADDURATION2 = 12;
  AUTOADDFLAG = 13;
  DYNAMICMODE = 14;
  HARMONICMODE = 15;
  GENERALTIME = 16;
```

GetStepSize:

Function():Double; StdCall;

Returns time step size, *h*, from the DynaVars structure.

GetTimeSec:

Function():Double; StdCall;

Returns time, *t*, in seconds starting hour from the DynaVars structure.

GetTimeHr:

Function():Double; StdCall;

Returns time as a double value in units of hours.

GetPublicDataPtr:

Procedure(var pPublicData:Pointer; var PublicDataBytes:Integer); StdCall;

Returns a pointer to the public data structure of the element supporting the DLL. The procedure returns the size of the data structure in bytes for checking for a different size than the DLL was designed for. The public data structure is a fixed size structure whose contents vary with class of device and the DLL programmer will have to know how it is defined.

This pointer is provided for fast access to internal variables in time critical simulations. However, there is no checking and memory can be easily overwritten if the programmer is not careful with pointers.

GetActiveElementName:

Function(FullName:pAnsiChar; MaxNameLen:Cardinal) :Integer; StdCall;

Returns the full name of the active circuit element such as “Line.Line1” in a null-terminated Ansi string. You must allocate a buffer to receive the string and pass the size of the buffer in the MaxNameLen variable so that OpenDSS does not accidentally overwrite the buffer.

GetActiveElementPtr:

Function():Pointer; StdCall;

Returns a pointer to the active circuit element. This is used primarily for the **ControlQueuePush** function below.

ControlQueuePush: Function(Const Hour:Integer; Const Sec:Double; Const Code, ProxyHdl:Integer; Owner:Pointer):Integer; StdCall;

This function is used to push messages onto the control queue for discrete controls with delayed actions (e.g., regulator and capacitor controls). The control queue sorts the messages in time order and pops them off in order when the time for action has been reached in the simulation.

Time for the delayed operation is specified in *Hour* (Integer) and *Sec* (Double), both of which may be computed from the result of the **GetTimeHr** function call above.

Code is an integer code that is meaningful to the control element that will receive the message. *ProxyHdl* is used for dispatching custom control proxies that use the COM interface and will generally be 0 for user-written DLLs that use the call back routines.

Owner is usually a pointer to the active circuit element retrieved by the **GetActiveElementPtr** function. Must be a Control type that has a DoPendingAction function. When a message is popped off the queue, it is sent to the DoPendingAction function of the element referenced by this pointer.

Returns an integer handle to the control queue message.

GetResultStr:

Procedure(S:pAnsiChar; Maxlen:Cardinal); StdCall;

Returns the global result string (result of last command) as an Ansi character string in the buffer that S points to. You must allot space for S and pass the length of that buffer in the Maxlen argument so OpenDSS does not overwrite memory when the result string is lengthy.