# User Defined Model Development Guide

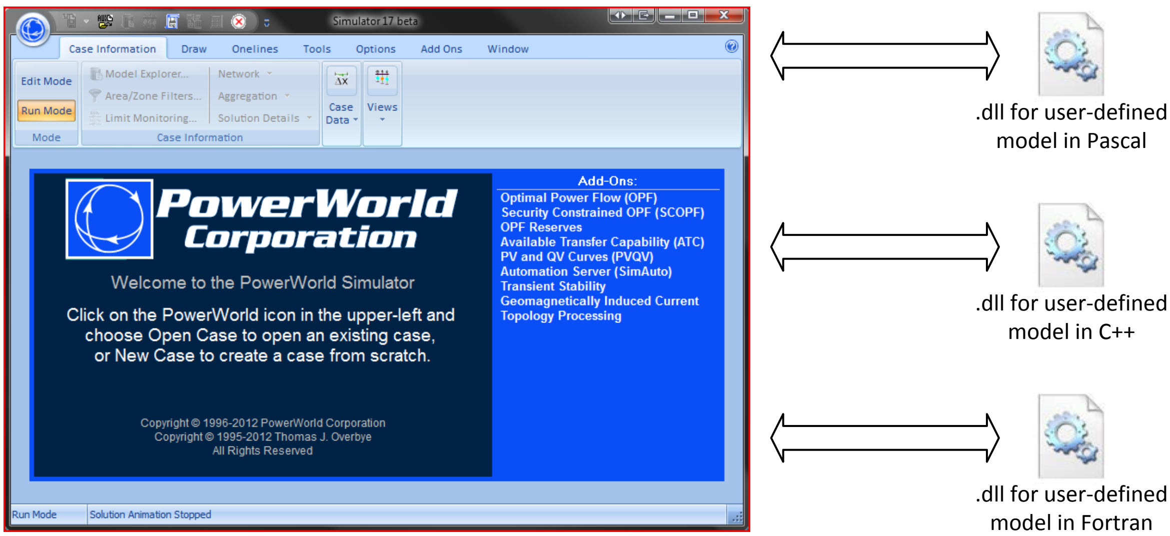
Last Updated: March 14, 2013

PowerWorld Corporation
2001 South First St
Champaign, IL 61820
(217) 384-6330
http://www.powerworld.com
info@powerworld.com

# Overview

The purpose of this document is to describe the user defined modeling interface in PowerWorld Simulator to developers. This document should facilitate a deeper understanding of the interactions of user defined models (UDMs) with Simulator. The basics of UDMs from a user's perspective are covered in Simulator's standard help documentation. Additionally, a tutorial and templates containing sample code are available.

User defined models provide an alternative to built-in models. The user can load standalone alone *.dll files and assign them to components in the power system case. These DLL files contain a library of functions that is completely separate but can be accessed from PowerWorld Simulator.




On the Simulator side, each user defined model DLL is represented by a **UserDefinedModel** object. This object is not automatically linked to any particular transient stability objects (generators, etc.). There are no instances of it until you insert them.

Each DLL corresponds to one user defined model type. Simulator manages all memory and keeps track of all of the instances of each type. When a new instance of a model is created, the DLL initializes it, and Simulator maintains all of the values of its parameters and states, etc. The DLL is given access to the memory where that information is stored (using pointers) so the DLL can access it within its functions.

The functions, their input arguments, and their return values are described in Section 6. The tutorial and sample models provide a starting point to begin creating user-defined models. The programming languages that have been tested are Pascal, C++ and Fortran. There are some limitations in Fortran because it lacks an object-oriented style; Pascal and C++ might prove more useful, especially when making complicated models such as CLOD (Complex Load Model).

Everything discussed in this document is geared towards a 32-bit platform.

## 1. Developer's Responsibility

All of the variables in user defined modeling are passed as pointers. This allows the DLLs to manipulate the data and operate as intended with PowerWorld Simulator. However, code in the DLL can potentially impact other parts of PowerWorld Simulator by inadvertently overwriting memory locations in use. This may lead to undesired operation of Power World Simulator. It is the responsibility of model developer to ensure that the DLL does not initiate such unwanted operations. The sample models are intended to provide an appropriate reference.

## 2. Model Class Types

The presently supported model classes and their corresponding names in Simulator are given in the table below.

| | Simulator Name |
|---|---|
| Machine Models | **UserDefinedMachineModel** |
| Exciter Models | **UserDefinedExciter** |
| Governor Models | **UserDefinedGovernor** |
| Stabilizer Models | **UserDefinedStabilizer** |
| Load Characteristic Models | **UserDefinedLoadModel** |
| Multi-terminal DC Converter Models | **UserDefinedMTDCConverter** |
| Multi-terminal DC Line Models | **UserDefinedMultiTerminalDC** |

## 3. Automatic Loading of DLLs from Directories

The user simply drops all user defined models that are to be used in a specific directory and tells Simulator where that directory is. Once a directory is selected to monitor, Simulator will automatically try to read in all of the DLL files contained in that directory as user defined models. Simulator will watch for changes in the directory and automatically add or remove the corresponding Simulator models accordingly. To aid those who are developing the user defined models, since it is not always possible to move the DLLs to a directory during debugging, multiple paths may be specified which are accessed in the specified order.

## 4. Signal Selection

Each class of supported model now has a hard-coded list of signals that are passed into and out of the model. This automatic handling of signal selection is done to make development easier for the user defined model – the signals which are necessary and common for the model class are automatically included. This is to avoid requiring the user to specify the signals which essentially are the same for all models of the same model class, i.e. machine models, governor models, etc.

If additional input fields from Simulator are required, they can be specified in the "Algebraics" array inside the TTxMyModelData structure. The user defined model tells Simulator the size of this array, and Simulator allocates memory for it. The signalSelection function specifies the field name, bus loc, and digits corresponding to the values to be passed in the Algebraics array. If an object other than the local object is to be used for a particular field, the "digits" field specifies which extra object to use, corresponding to "Num" in OtherObjectClass and OtherObjectDescription. For example, a stabilizer may use a voltage signal from another bus.

Simulator does not need to know all that is stored in the Algebraics array. After the end of the fields specified by signalSelection, the user can store Custom Algebraics that are not used internally by Simulator. Simulator does still have access to these variables for plotting. All computations with Custom Algebraics, if any, are on the DLL side. Some models such as load models may require custom variables of this type.

## 5. Hard-coded Available Signals by Index for Each Model Class

Certain signals are always automatically made available to each model based on its class. The values of the signals are located inside the HardCodedSignals array of the TTxMyModelData structure, using the indices given below. Indexing begins at zero.

### Exciter Models

```
HARDCODE_EXCITER_Vref                = 0;
HARDCODE_EXCITER_InitFieldVoltage    = 1;
HARDCODE_EXCITER_FieldCurrent        = 2;
HARDCODE_EXCITER_GenVcomp            = 3;
HARDCODE_EXCITER_GenSpeedDeviationPU = 4;
HARDCODE_EXCITER_BusVoltMagPU        = 5;
HARDCODE_EXCITER_StabilizerSignal    = 6;
HARDCODE_EXCITER_OELActive           = 7;
HARDCODE_EXCITER_OELSignal           = 8;
HARDCODE_EXCITER_UELActive           = 9;
HARDCODE_EXCITER_UELSignal           = 10;
```

| [Index] Signal | Description |
|---|---|
| [0] HARDCODE_EXCITER_Vref | Voltage reference for the exciter. Value should be set by the DLL during initialization and is an input afterward. |
| [1] HARDCODE_EXCITER_InitFieldVoltage | Initial value of machine field voltage $E_{fd}$. Input only. |
| [2] HARDCODE_EXCITER_FieldCurrent | Present value of machine field current $I_{fd}$. Input only. |
| [3] HARDCODE_EXCITER_GenVcomp | Compensated terminal voltage of the machine. Input only. |
| [4] HARDCODE_EXCITER_GenSpeedDeviationPU | Generator speed deviation $\Delta\omega$. Input only. |
| [5] HARDCODE_EXCITER_BusVoltMagPU | Generator terminal voltage magnitude. Input only. |
| [6] HARDCODE_EXCITER_StabilizerSignal | Input signal from stabilizer Vs. Input only. |
| [7] HARDCODE_EXCITER_OELActive | Flag for active over excitation limiter (OEL), 1 indicates active. |
| [8] HARDCODE_EXCITER_OELSignal | OEL signal, if active. Input only. |
| [9] HARDCODE_EXCITER_UELActive | Flag for active under excitation limiter (UEL), 1 indicates active. |
| [10] HARDCODE_EXCITER_UELSignal | UEL signal, if active. Input only. |

## Governor Models

```
HARDCODE_GOV_Pref                  = 0;
HARDCODE_GOV_InitPmech             = 1;
HARDCODE_GOV_GenSpeedDeviationPU = 2;
HARDCODE_GOV_GenPElecPU            = 3;
HARDCODE_GOV_GenMVABase            = 4;
HARDCODE_GOV_GovResponseLimits     = 5;
HARDCODE_GOV_StabStatePitch        = 6;
```

| [Index] Signal | Description |
|---|---|
| [0] HARDCODE_GOV_Pref | Power reference $P_{ref}$. Value should be set by the DLL during initialization and is an input afterward. |
| [1] HARDCODE_GOV_InitPmech | Initial mechanical power $P_{mech}$. Input only. |
| [2] HARDCODE_GOV_GenSpeedDeviationPU | Generator speed deviation Δω. Input only. |
| [3] HARDCODE_GOV_GenPElecPU | Electrical power $P_{elec}$. Input only. |
| [4] HARDCODE_GOV_GenMVABase | Generator MVA base. Input only. |
| [5] HARDCODE_GOV_GovResponseLimits | Governor response limits. A byte representing the “GE Baseload_flag” parameter, where 0 means “normal” (valves act normally and can open or close), 1 means “close only” response (valves can close but not open), and 2 means “fixed” response (valve is stuck at present position). Input only. |
| [6] HARDCODE_GOV_StabStatePitch | Pitch input from “stabilizer” pitch model. Input only. Applicable only for wind models. |

## Stabilizer Models

```
HARDCODE_STAB_GenSpeedDeviationPU     = 0;
HARDCODE_STAB_BusFreqDeviationPU      = 1;
HARDCODE_STAB_GenPElecPU              = 2;
HARDCODE_STAB_GenPAccelPU             = 3;
HARDCODE_STAB_BusVoltMagPU            = 4;
HARDCODE_STAB_GenVcomp                = 5;
```

| [Index] Signal | Description |
|---|---|
| [0] HARDCODE_STAB_GenSpeedDeviationPU | Generator speed deviation Δω. Input only. |
| [1] HARDCODE_STAB_BusFreqDeviationPU | Bus frequency deviation Δω. Input only. |
| [2] HARDCODE_STAB_GenPElecPU | Electrical power $P_{elec}$. Input only. |
| [3] HARDCODE_STAB_GenPAccelPU | Accelerating power $P_{accel}$. Input only. |
| [4] HARDCODE_STAB_BusVoltMagPU | Generator terminal voltage magnitude. Input only. |
| [5] HARDCODE_STAB_GenVcomp | Compensated terminal voltage of the machine. Input only. |

## Machine Models

```
HARDCODE_MACHINE_TSGenFieldV = 0;
HARDCODE_MACHINE_TSPmech     = 1;
```

```
HARDCODE_MACHINE_InitVreal   = 2;
HARDCODE_MACHINE_InitVimag   = 3;
HARDCODE_MACHINE_InitIreal   = 4;
HARDCODE_MACHINE_InitIimag   = 5;
HARDCODE_MACHINE_TSstateId   = 6;
HARDCODE_MACHINE_TSstateIq   = 7;
```

| [Index] Signal | Description |
|---|---|
| [0] HARDCODE_MACHINE_TSGenFieldV | Field voltage Efd signal from exciter. Value should be set by the DLL during initialization and is an input afterward. |
| [1] HARDCODE_MACHINE_TSPmech | Mechanical power Pmech signal from governor. Value should be set by the DLL during initialization and is an input afterward. |
| [2] HARDCODE_MACHINE_InitVreal | Real part of the initial terminal voltage. Input only. |
| [3] HARDCODE_MACHINE_InitVimag | Imaginary part of the initial terminal voltage. Input only. |
| [4] HARDCODE_MACHINE_InitIreal | Real part of the initial terminal current. Input only. |
| [5] HARDCODE_MACHINE_InitIimag | Imaginary part of the initial terminal current. Input only. |
| [6] HARDCODE_MACHINE_TSstateId | Machine d-axis current $I_d$. Value should be set during machine initialization and is an input afterward. Then, this value is maintained by Simulator using the Thevenin or Norton equivalent parameters from the DLL. |
| [7] HARDCODE_MACHINE_TSstateIq | Machine q-axis current $I_q$. Value should be set during machine initialization and is an input afterward. Then, this value is maintained by Simulator using the Thevenin or Norton equivalent parameters from the DLL. |

## Load Characteristic Models

```
HARDCODE_LOAD_DeviceVPU      =  0;
HARDCODE_LOAD_DeviceAngleRad =  1;
HARDCODE_LOAD_DeltaFreqPU    =  2;
HARDCODE_LOAD_DeviceStatus   =  3;
HARDCODE_LOAD_LoadScalar     =  4;
```

| [Index] Signal | Description |
|---|---|
| [0] HARDCODE_LOAD_DeviceVPU | Load bus voltage magnitude. Input only. |
| [1] HARDCODE_LOAD_DeviceAngleRad | Load bus voltage angle. Input only. |
| [2] HARDCODE_LOAD_DeltaFreqPU | Load bus frequency deviation from nominal $\Delta\omega$. Input only. |
| [3] HARDCODE_LOAD_DeviceStatus | A boolean indicating whether the load is in service. Input only. |
| [4] HARDCODE_LOAD_LoadScalar | A scalar for scaling the load. All loads that derive from this load should be multiplied by this scalar. |

| | |
|---|---|
| | This is initially 1, but load relays may cause it to be reduced. |

## Multi-Terminal DC Converter Models

```
HARDCODE_MTDCConv_IRef         =  0;
HARDCODE_MTDCConv_InitIOrd     =  1;
HARDCODE_MTDCConv_InitCosAngle =  2;
HARDCODE_MTDCConv_Idc          =  3;
HARDCODE_MTDCConv_Vdc          =  4;
HARDCODE_MTDCConv_Vac          =  5;
```

| [Index] Signal | Description |
|---|---|
| [0] HARDCODE_MTDCConv_IRef | Present value of the current reference ID_Ref. Value should be set by the DLL during initialization and is an input afterward. |
| [1] HARDCODE_ MTDCConv _InitIOrd | Initial current order. Input only. |
| [2] HARDCODE_ MTDCConv _InitCosAngle | Initial cosine of the control angle. Here, this signal is input only. Its value should be maintained by the DLL in the MTDCConverterCosControlAngle function. |
| [3] HARDCODE_ MTDCConv _Idc | DC current in Amps. Input only. |
| [4] HARDCODE_ MTDCConv _Vdc | DC voltage in kV. Input only. |
| [5] HARDCODE_ MTDCConv _Vac | AC voltage in pu. Input only. |

## Multi-Terminal DC Line Models

Multi-terminal DC lines will receive the following hardcoded signals for **each converter model**.

```
HARDCODE_MTDC_Iref     =  0;
HARDCODE_MTDC_Idc      =  0;
HARDCODE_MTDC_Vdc      =  1;
HARDCODE_MTDC_Vac      =  2;
HARDCODE_MTDC_IdcSense =  3;
HARDCODE_MTDC_VdcSense =  4;
HARDCODE_MTDC_VacSense =  5;
```

| [Index] Signal | Description |
|---|---|
| [0] HARDCODE_MTDC_IRef | Present value of the current reference ID_Ref from the converter. |
| [1] HARDCODE_ MTDC _Idc | MTDC line section current in Amps from network solution. |
| [1] HARDCODE_ MTDC _Vdc | MTDC voltage at the DC bus from network solution. |
| [2] HARDCODE_ MTDC _Vac | MTDC voltage at the AC bus from networks olution. |
| [3] HARDCODE_ MTDC _IdcSense | Sensed DC current in Amps from the converter. |
| [4] HARDCODE_ MTDC _VdcSense | Sensed DC voltage in kV from the converter. |
| [5] HARDCODE_ MTDC _VacSense | Sensed AC voltage in pu from the converter. |

## 6. Exported Functions for Each Model Class

A list of function names that must be made available in the export directory of the DLL for each model class is given below. Functions with names in italic are optional. Detailed descriptions of each function are given in the tables that follow. Data type compatibility is discussed in Section 8. The functions in the export directory of the DLL file are all called using the stdcall calling convention which is a variation on the Pascal calling convention in which the callee is responsible for cleaning up the stack, but the parameters are pushed onto the stack in right-to-left order. Registers EAX, ECX, and EDX are designated for use within the function. Return values are stored in the EAX register.

Note that the function calls (including names and parameter types) exported from this DLL must exactly match those being expected in Simulator (as listed below).

### All - General

DLLVersion
modelClassName
allParamCounts
parameterName
stateName
getDefaultParameterValue
*OtherObjectClass*
*OtherObjectDescription*
*getStringParamDefaultValue*
*signalSelection*

| | |
|---|---|
| **DLLVersion**<br><br>An integer to support versioning in the future. Currently, use "1." | |
| parameters | N/A |
| result | Integer |
| **modelClassName**<br><br>Simulator calls this function twice, once to get the length ("result") in characters of the model class name, and once to retrieve the model class name, i.e. "UserDefinedExciter," in the buffer which Simulator allocates. The purpose of this function is for Simulator to recognize the type of transient stability model contained in the DLL. This should be one of the supported classes. | |
| parameters | (StrSize:PInteger; StrBuf : PChar; dummy : Integer) |
| result | Integer |
| **allParamCounts**<br><br>Fills the TTxParamCounts structure in Simulator to tell Simulator how much memory to allocate. | |
| parameters | (var numbersOfEverything : TTxParamCounts; TimeStepSeconds : double) |
| result | N/A |
| **parameterName** | |

<table>
<tr><td colspan="2">Simulator calls this function twice for each parameter and works the same way as modelClassName.</td></tr>
<tr><td>parameters</td><td>(paramNum : PInteger; StrSize : PInteger; StrBuf : PChar; dummy : Integer)</td></tr>
<tr><td>result</td><td>Integer</td></tr>
<tr><td colspan="2">stateName<br><br>Works the same way as modelClassName.</td></tr>
<tr><td>parameters</td><td>(paramNum : PInteger; StrSize : PInteger; StrBuf : PChar; dummy : Integer)</td></tr>
<tr><td>result</td><td>Integer</td></tr>
<tr><td colspan="2">getDefaultParameterValues<br><br>Simulator retrieves the default parameter values inside a TTxMyModelData structure.</td></tr>
<tr><td>parameters</td><td>(paramsAndStates : PTxMyModelData)</td></tr>
<tr><td>results</td><td>N/A</td></tr>
<tr><td colspan="2">OtherObjectClass<br><br>The PowerWorld class of each “other object” to be used. This function must be written if the model uses “other objects.” This must match the object name in Simulator, i.e. “Bus.” Works the same way as modelClassName. “Num” gives the index of the other object in the list.</td></tr>
<tr><td>parameters</td><td>(Num : PInteger; StrSize : PInteger; StrBuf : PChar; dummy : Integer)</td></tr>
<tr><td>results</td><td>Integer</td></tr>
<tr><td colspan="2">OtherObjectDescription<br><br>A user-specified description of each “other object” to be used, i.e. “Signal Bus,” used for the GUI. This function should be written if the model uses “other objects.”</td></tr>
<tr><td>parameters</td><td>(Num : PInteger; StrSize : PInteger; StrBuf : PChar; dummy : Integer)</td></tr>
<tr><td>results</td><td>Integer</td></tr>
<tr><td colspan="2">getStringParamDefaultValue<br><br>Default values for string parameters, if any.</td></tr>
<tr><td>parameters</td><td>(Num : PInteger; StrSize : PInteger; StrBuf : PChar; dummy : Integer)</td></tr>
<tr><td>results</td><td>Integer</td></tr>
<tr><td colspan="2">signalSelection<br><br>Names of fields in ALG vector at position Num. Only fields that Simulator knows about should appear. This includes fields corresponding to “other objects,” where the format is “FieldName:BusLoc:Digits,” where Digits specifies the other object (Num) in otherObjectClass and otherObjectDescription. Custom algebraics should only appear at the end of the ALG vector, and are not listed here.</td></tr>
<tr><td>parameters</td><td>(Num : PInteger; StrSize : PInteger; StrBuf : PChar; dummy : Integer)</td></tr>
<tr><td>results</td><td>Integer</td></tr>
</table>

## All - Numerical Integration

initializeYourself
calculateFofX
PropogateIgnoredStateAndInput
*SubIntervalPower2Exponent*
*getNonWindUpLimits*
*TimeStepEnd*
*TimeStepEndAction*

**initializeYourself**

Initialization of the dynamic model. By assuming f(x) is zero at steady-state, the initial values of the model states are set inside the TTxMyModelData structure, pointed to by PTxMyModelData. The TTxMyModelData structure shares relevant network input fields with the DLL and allows the DLL to set the values of the calculated fixed input fields needed by Simulator. Relevant system options are also shared. See description of the TTxMyModelData and TTxSystemOptions structures.

| | |
|---|---|
| parameters | (paramsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions) |
| results | N/A |

**calculateFofX**

These are the differential equations of the model, xdot = f(x), which get called every time step. The actual numerical integration of these equations is handled in Simulator.

| | |
|---|---|
| parameters | (paramsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; nonWindUpLimits : PTxNonWindUpLimits; dotX : PDouble) |
| results | N/A |

**PropogateIgnoredStateAndInput**

This function handles ignored states. That is, if choices for certain parameter values cause a state to be “ignored,” this function must make sure the inputs to the ignored state are correctly propagated through to the next state. ParamsAndStates.IgnoreStates is used to propogate the values and should be set in the initialization function based on the parameters. An example of this is the User_IEEEST model.

| | |
|---|---|
| parameters | (paramsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions) |
| results | N/A |

**SubIntervalPower2Exponent**

This is an optional function that tells Simulator the exponent to use when determining the number of subintervals for integrating the model. The actual number of subintervals will be calculated as 2^exponent, so if you want 8 subintervals, this function should return 3 (2^3=8).

| | |
|---|---|
| parameters | N/A |
| results | Integer |

**getNonWindUpLimits**

This function tells Simulator the index of states which have non-windup limits and the values of

| | |
|---|---|
| those limits by setting LimitStates, minLimits, and maxLimits inside the TTxNonWindUpLimits structure. "Result" specifies how many states have nonwindup limits. States are indexed starting at zero. | |
| parameters | (paramsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; nonWindUpLimits : PTxNonWindUpLimits) |
| results | N/A |
| **TimeStepEnd**<br><br>This function can perform specific checks at the end of a timestep and returns true if an action should actually occur at the end of the timestep. The User_CLOD model uses this to check whether to perform an undervoltage trip. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; index : Integer; MaxPossibleEventIndex : PInteger; EventTime : PDouble; ExtraObjectIndex : PInteger) |
| results | Boolean |
| **TimeStepEndAction**<br><br>This function returns a string containing the name of the action for Simulator to perform corresponding to the same "index" in TimeStepEnd, a pipe character \|, and a custom log message. The action should match PowerWorld's syntax for event descriptions, i.e. keyword "OPEN" will trip a load. Like all string functions, this is called twice. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; index : integer; StrSize: PInteger; StrBuf: PChar; dummy : Integer) |
| results | Boolean |

## Exciter Models

### ExciterEfieldOut

| | |
|---|---|
| **ExciterEfieldOut**<br><br>This function returns the final value of Efield from the exciter, taking into account any limits. This value is the field voltage of the machine model, $E_{FD}$. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions) |
| results | Double |

## Governor Models

### GovernorPmechOut

| | |
|---|---|
| **GovernorPmechOut**<br><br>This function returns $P_{mech}$ out of the governor. This value is the mechanical power input for the machine model. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : |

| | |
|---|---|
| | PTxSystemOptions) |
| results | Double |

## Stabilizer Models

StabilizerVsOut
*StabilizerPitchOut*

| | |
|---|---|
| **StabilizerVsOut**<br><br>This function returns $V_S$ out of the Stabilizer, which is passed into the exciter. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions) |
| results | Double |
| **StabilizerPitchOut**<br><br>If the "stabilizer" is a wind turbine pitch control model, this function returns its pitch, to be used by the wind turbine "governor" model. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions) |
| results | Double |

## Machine Models

MachineSpeedDeviationOut
MachineTheveninImpedance
MachineTheveninVoltage
MachineFieldCurrent
MachineElectricalTorque
*MachineNortonCurrent*
*MachineHighVReactiveCurrentLim*
*MachineLowVActiveCurrentPoints*
*MachineCompensatingImpedance*

| | |
|---|---|
| **MachineSpeedDeviationOut**<br><br>This function returns the machine speed deviation from synchronous, which is normally also a state. This is passed into the governor model. It is also be used with Generic Limit Monitors to implement basic protection functionality for low frequency, high frequency, or excessive change. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions) |
| results | Double |
| **MachineTheveninImpedance**<br><br>This function returns the equivalent Thevenin impedance of the machine (R + jX), which is passed back to the network. For the GENCLS model, this is simply ($R_a + jX_d'$). | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : |

<table>
<tr><td></td><td>PTxSystemOptions; theR : PDouble; theX : PDouble)</td></tr>
<tr><td>results</td><td>Double</td></tr>
<tr><td colspan="2">MachineTheveninVoltage<br><br>This function returns the equivalent Thevenin voltage of the machine in the form $(V_d+jV_q)e^{j(\delta-\pi/2)}$, which is passed back to the network.</td></tr>
<tr><td>parameters</td><td>(ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; Delta : PDouble; Vd : PDouble; Vq : PDouble)</td></tr>
<tr><td>results</td><td>Double</td></tr>
<tr><td colspan="2">MachineFieldCurrent<br><br>This function returns the field current of the machine, which feeds into the exciter model as $I_{FD}$. This may also be checked by other models such as over excitation limiters (OELs) and under excitation limiters (UELs).</td></tr>
<tr><td>parameters</td><td>(ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions)</td></tr>
<tr><td>results</td><td>Double</td></tr>
<tr><td colspan="2">MachineElectricalTorque<br><br>This function returns the electrical torque delivered by the machine.</td></tr>
<tr><td>parameters</td><td>(ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions)</td></tr>
<tr><td>results</td><td>Double</td></tr>
<tr><td colspan="2">MachineNortonCurrent<br><br>This function returns the equivalent Norton current of the machine, which is passed back to the network. This function can be written instead of the Thevenin equivalent voltage.</td></tr>
<tr><td>parameters</td><td>(ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; IReal : PDouble; IImag : PDouble)</td></tr>
<tr><td>results</td><td>Double</td></tr>
<tr><td colspan="2">MachineHighVReactiveCurrentLim<br><br>Returns the high voltage limit for high voltage reactive current management, if any. If this voltage is exceeded at the bus, the functionality adjusts the reactive power injection to clamp the voltage. If this voltage is exceeded during initialization, the limit is assumed to be incorrect and is ignored.</td></tr>
<tr><td>parameters</td><td>(ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions)</td></tr>
<tr><td>results</td><td>Double</td></tr>
<tr><td colspan="2">MachineLowVActiveCurrentPoints<br><br>For low voltage active current management. Returns the breakpoints, if any. When the bus voltage is above Lvpt1, no low voltage logic is used. When the bus voltage is below Lvpnt0, the active current is zero. Between Lvpt1 and Lvpt0, the active current is linearly ramped down. This should only occur during a fault.</td></tr>
<tr><td>parameters</td><td>(ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; Lvpnt1 : PDouble; Lvpnt0 : PDouble)</td></tr>
</table>

| | |
|---|---|
| results | N/A |
| **MachineCompensatingImpedance**<br><br>This function returns the compensating resistance and reactance for the machine, if any. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; Rcomp : PDouble; Xcomp : PDouble) |
| results | N/A |

## Load Characteristic Models

LoadNortonAdmittance
LoadNortonCurrent
LoadNortonCurrentAlgebraicDerivative
LoadInitializeAlgebraic

| | |
|---|---|
| **LoadNortonAdmittance**<br><br>Returns the equivalent Norton admittance of the load. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; theG : PDouble; theB : PDouble) |
| results | N/A |
| **LoadNortonCurrent**<br><br>Returns the equivalent Norton current of the load. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; IReal : PDouble; IImag : PDouble) |
| results | N/A |
| **LoadNortonCurrentAlgebraicDerivative**<br><br>Derivative of the equivalent Norton current of the load with respect to rectangular voltage. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; IReal_dVreal : PDouble; IReal_dVimag : PDouble; IImag_dVreal : PDouble; IImag_dVimag : PDouble) |
| results | N/A |
| **LoadInitializeAlgebraic**<br><br>Initializes the algebraic variables for the load, including the P and Q used. Custom algebraic variables in the Algebraics vector may be initialized here. Returns true if successful. | |
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; INPUT_PUTol, SteadyStateP, SteadyStateQ, SteadyStateV : Double; InitLoadP, InitLoadQ : PDouble) |
| results | Boolean |

## Multi-Terminal DC Converter Models

MTDCConverterCosControlAngle

*MTDCConverterIdcSense*

*MTDCConverterVdcSense*

*MTDCConverterVacSense*

*MTDCConverterCurrentLimitAndMargin*

| **MTDCConverterCosControlAngle**<br><br>Returns the output of the converter, the cosine of the control angle, either cos(α) or cos(β), depending on whether the converter is acting as a rectifier or inverter, respectively. | |
|---|---|
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions) |
| results | Double |

| **MTDCConverterIdcSense**<br><br>Returns the DC current which changes when the control angle changes. Other converters connected to the same DC network may need to use this current. | |
|---|---|
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions) |
| results | Double |

| **MTDCConverterVdcSense**<br><br>Returns the DC voltage at the converter terminal. | |
|---|---|
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions) |
| results | Double |

| **MTDCConverterVacSense**<br><br>Returns the AC voltage at the converter terminal. | |
|---|---|
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions) |
| results | Double |

| **MTDCConverterCurrentLimitAndMargin**<br><br>Sets the current limit on the current order (Iord) and margin for the limit. The margin | |
|---|---|
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; IdRefLim, fid_Margin : PDouble) |
| results | N/A |

### Multi-Terminal DC Line Models

MultiTerminalDCGetIDRef

*NetworkSolutionEnd*

| **MultiTerminalDCGetIDRef**<br><br>This function returns the reference current IDRef. | |
|---|---|
| parameters | (ParamsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions; index : integer) |
| results | Double |
| **NetworkSolutionEnd**<br><br>Called at the end of the time step to perform any final actions. | |
| parameters | (paramsAndStates : PTxMyModelData; SystemOptions : PTxSystemOptions) |
| results | N/A |

## 7. Memory Sharing Data Structures

Data sharing between user defined transient stability model DLLs and Simulator is accomplished using the following structures on the DLL side. The Simulator side performs all memory allocation management.

### TTxMyModelData Record

```
 TTxMyModelData = record
    FloatParams   : PDouble;
    IntParams     : PInteger;
    StrParams     : PPChar;
    HardCodedSignals : PDouble;
    States        : PDouble;
    IgnoreStates : PBoolean;
    Algebraics    : PDouble;
  end;
  PTxMyModelData = ^TTxMyModelData;
```

### TTxSystemOptions Record

```
  TTxSystemOptions = record
    IgnoreLimitChecking    : boolean;
    TimeStepSeconds        : double;
    SimulationTimeSeconds : double;
    WBase                  : double;
    SBase                  : double;
    PUSolutionTolerance    : double;
    MinVoltSLoad           : double;
    MinVoltILoad           : double;
  end;
  PTxSystemOptions = ^TTxSystemOptions;
```

### TTxNonWindUpLimits Record

```
TTxNonWindUpLimits = record
  LimitStates  : PInteger;
  minLimits    : PDouble;
  maxLimits    : PDouble;
  activeLimits : PByte;
end;
PTxNonWindUpLimits = ^TTxNonWindupLimits;
```

### TTxParamCounts Record

```
TTxParamCounts = record
  nFloatParams   : Integer;
  nIntParams     : Integer;
  nStrParams     : Integer;
  nStates        : Integer;
  nAlgebraics    : Integer;
  nNonWindUpLimits : Integer;
end;
PTxParamCounts = ^TTxParamCounts;
```

| **TTxMyModelData**<br><br>A record containing all state, parameter, and signal data associated with each instance of a user defined model. | |
|---|---|
| FloatParams : PDouble | Pointer to array of double parameters |
| IntParams : PInteger | Pointer to array of integer parameters |
| StrParams : PPChar | Pointer to array of string parameters |
| HardCodedSignals : PDouble | Pointer to double array of hard-coded signals from PW. These are always the same for all models of each class (i.e., all stabilizers, governors, etc.). Simulator always shares these signals with the DLL. If additional signals are needed from Simulator, they must be defined using the Algebraics array and the signalSelection function. |
| States : PDouble | Pointer to double array of state variables x |
| IgnoreStates : PBoolean | Pointer to a boolean array indicating whether each state is to be ignored |
| Algebraics : PDouble | Pointer to a double array containing all signals other than the hardcoded signals.<br><br>The signalSelection function can define and then the Algebraics array can access any fields that are available in Simulator. The signalSelection function lists the object/fields to be accessed, and the Algebraics array is where the actual values are located.<br><br>Additionally, the Algebraics array may be used by the DLL to maintain its own "custom" algebraic variables. Custom algebraics must appear in the array AFTER the variables defined by |

| | |
|---|---|
| | signalSelection. An example of a model that uses custom algebraics is the User_CLOD model. |
| **TTxSystemOptions**<br><br>A record containing system options that may be relevant to the user defined model during the transient stability simulation. These are available to the DLL from Simulator. | |
| IgnoreLimitChecking : boolean | Set to true if limits should be ignored |
| TimeStepSeconds : double | The time step in seconds |
| SimulationTimeSeconds : double | The present time in seconds in the transient stability simulation. This is useful for models that use timers. |
| WBase : double | The base frequency in rad/sec |
| SBase : double | The three-phase power base |
| PUSolutionTolerance : double | |
| MinVoltSLoad : double | The minimum allowable voltage for constant power load |
| MinVoltSLoad : double | The minimum allowable voltage for constant current load |
| **TTxNonWindUpLimits**<br><br>A record specifying the states which have non-windup limits, what the limit values are, and which limits are presently active for each state. | |
| LimitStates : PInteger | A pointer to an integer array specifying the states by number which have non-windup limits. |
| minLimits : PDouble | A pointer to a double array listing the minimum values of the limits for the states in LimitStates |
| maxLimits : PDouble | A pointer to a double array listing the maximum values of the limits for the states in LimitStates |
| activeLimits : PByte | A pointer to a byte array which contains information on which limits are active. For each limit in LimitStates, a value of 0 means not active, 1 means active at the high limit, and 2 means active at the low limit. |
| **TTxParamCounts**<br><br>A record used to hold and access the counts of each of array. This prevents us from requiring many different “getNumberOf” functions in the DLL that need to be called by Simulator in order to allocate memory. It is convenient to define these numbers as constants in the DLL. | |
| nFloatParams : Integer | Number of double parameters |
| nIntParams : Integer | Number of integer parameters |
| nStrParams : Integer | Number of string parameters |
| nStates : Integer | Number of dynamic states |
| nAlgebraics : Integer | Number of algebraic variables in the Algebraics array. This number MUST include any custom algebraics in addition to the algebraics defined by signalSelection. |
| nNonWindUpLimits : Integer | Number of states with non-windup limits |

## Extra Objects

In addition to having access to all of an object's own fields, each user defined model also has the ability to specify "extra objects," where fields for other objects can serve as inputs or outputs to the model.

Values corresponding to the extra objects are stored in the algebraic vector. The corresponding object and field identifiers are specified in the signalSelection function.

### DLL Side

The DLL does not know which particular object is selected, but it knows what field type it requires, and it knows the index in the Algebraics array where it expects to find the value obtained from Simulator. The functions OtherObjectClass and OtherObjectDescription must be written to specify what field is required.

### Simulator Side

There is a dialog for specifying the extra objects on the Simulator side. For example, all the DLL knows is that it requires a "voltage" at a "signal bus." Then, the user can choose the object, i.e. "Bus 3" on the Simulator side.

# 8. Compatibility with Other Programming Languages

DLLs created in the programming languages Pascal, C++, and Fortran have been debugged and tested for compatibility with PowerWorld Simulator. This section details the important differences of these languages for PowerWorld UDM implementation.

### Data Structures and Variable Passing

Implementation of the required variables and data structures is very similar in all three programming languages. Example data structures are illustrated in Figure 1 and Figure 2. A structure with pointers to arrays and an array are shown, respectively. The implementation of these is slightly different in Fortran. Fortran also behaves somewhat differently when these variables are being passed to and from functions. While Pascal and C++ allow both pass by value and pass by reference, Fortran always uses pass by reference. To maintain compatibility with all three languages, some variables such as ParaNum, StateNum, StrSize, and dummy are intentionally passed as pointers to their respective locations, even though it is not necessary in Pascal and C++. The variable "dummy" appears in the function definitions in Pascal and C++, but not in Fortran. It represents a hidden input argument which is inserted and expected automatically on the Fortran side whenever a character array is being exchanged. Again, to maintain compatibility, "dummy" must appear appropriately in Pascal and C++, but does not show up in Fortran code.

There is no limit to the number of characters each string parameter (i.e., parameter name) can have in Pascal and C++. However, a 30 character limit has been set in the templates created in Fortran, which can be increased by altering a parameter in the Fortran script.

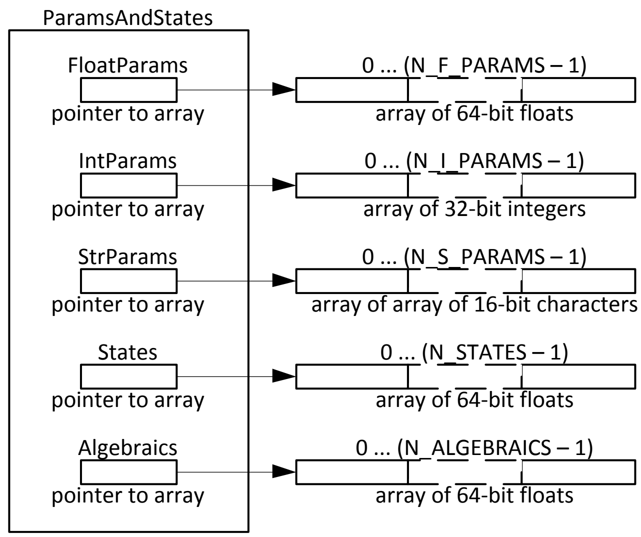



**Figure 1: Structure with Pointers to Arrays**

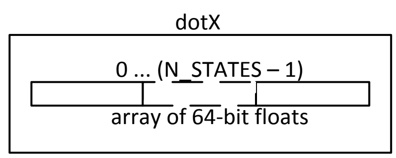



**Figure 2: Array**

## Data Type Compatibility

In mixed-language programming, particular attention needs to be given toward data type compatibility between programming languages. There might be limitations, but the commonly used data types are usually available in any programming language.

**Table 1. Comparison between data types across languages**

| Bytes [Bits] | Pascal (Embarcadero® Delphi® XE Version 15.0) | C++ (Microsoft® Visual Studio 2010) | Fortran (Microsoft® Visual Studio 2010 with Silverfrost FTN95 plug-in) |
|---|---|---|---|
| 1 [8] | ShortInt | __int8 | integer(kind = 1) |
| 2 [16] | SmallInt | __int16 | integer(kind = 2) |
| 4 [32] | Integer | int | integer(kind = 3) |
| 1 [8] | Byte | unsigned __int8 | |
| 2 [16] | Word | unsigned __int16 | |
| 4 [32] | Cardinal | unsigned int | |
| 1 [8] | Boolean, ByteBool | bool | logical(kind = 1) |
| 2 [16] | WordBool | | logical(kind = 2) |
| 4 [32] | LongBool | | logical(kind = 3) |
| 4 [32] | Single | float | real(kind = 1) |
| 8 [64] | Double | double | real(kind = 2) |
| 10 [80] | Extended, Real (32-bit sys.) | | real(kind = 3) |
| 1 [8] | AnsiChar | char | character |

| | | | |
|---|---|---|---|
| 2 [16] | Char, WideChar | wchar_t | |

Table 1 is not an exhaustive collection of data types, but a guide for those most commonly used. There are possibly other aliases, which can be found within documentation of each language [1], [2], [3]. The point to take note of is that, for cross-language compatibility, the type and the size of data types must match.

## 9. Tutorial and Example DLL Files

A tutorial and example DLL project files are available for all three languages.

## 10. References


[1]http://docwiki.embarcadero.com/RADStudio/en/Delphi_Data_Types

[2]http://msdn.microsoft.com/en-us/library/s3f49ktz(v=vs.100).aspx

[3]http://www.silverfrost.com/ftn95-help/mixlan/basicdatatypes.aspx