

# DLL programming in EMTP

DLL programming in EMTP.....	1
1 Introduction .....	2
2 Standard path names.....	2
3 Interfacing method .....	2
3.1 Network equations .....	3
3.1.1 Main system of equations.....	3
3.1.2 Example.....	4
4 The request and participation setup .....	8
4.1 The sequences of Core requests .....	9
4.2 Data Input Requests .....	9
4.3 Frequency scan requests .....	9
4.4 Steady-state requests.....	9
4.5 Time-domain requests .....	10
5 DLL structure .....	12
5.1 Main structure .....	12
5.2 Building rules .....	13
5.3 DLL function definitions .....	16
5.3.1 Compiling and linking .....	18
5.3.2 Debugging .....	19
5.4 DLL Locations.....	20
6 Example 1 .....	20
6.1 Model .....	20
6.2 Folder contents .....	22
6.3 Debugging .....	22
6.4 Masking for DLL data specification.....	22
7 Example 2 .....	22

Jean Mahseredjian, 5/24/2020 3:33:00 PM

# 1 Introduction

The DLL (Dynamic Link Library) function is designed to allow EMTP users to develop advanced program model modules and interface them directly and intimately with the EMTP engine. It is for advanced users and offers advanced capabilities.

Users interested in programming language based model development should use this feature to develop and maintain models. Such models can be placed in a given location, attached to a device with graphical user interface features in EMTPWorks and maintained in a library as any other built-in EMTP model.

To build DLLs it is required to use compilers. It is allowed to write EMTP DLLs in almost any language.

It is assumed that the reader is familiar with programming languages and compilers.

The DLLs created by users are not supported by the EMTP support team. The support team will not study DLL codes to correct user errors.

## 2 Standard path names

In the following text some path names are installation dependent. Named values are used below to designate installation dependent path names:

1. **ApplicationDir**: designates the application folder (path)
  - a. On a 64 bit Windows system the default value is C:\Program Files (x86)\EMTPWorks.
  - b. The following script command lines can be run from a JavaScript window or code in EMTPWorks to determine a given application path:  
`writeln(getAppDir); //will output the application path in the Script Console window`
2. **ApplicationDataDir**: designates the application data folder (path)
  - a. On a Windows 10, 64 bit system, the typical default value is:  
`C:\Users\ThisUser\AppData\Roaming\EMTP\C_Program_Files_x86_EMTPWorks`
  - b. The following script command line can be run from a JavaScript window or code in EMTPWorks to determine application path:  
`writeln(getAppDataDir); //will output the application data path in the Script Console window`

## 3 Interfacing method

DLL programming is based on various standards supported by compilers. Such standards allow creating a function compiled by the user with specific access methods defined by the calling program. In this case the calling program is EMTP (computational engine, EMTP-engine). The DLL file name normally has the extensions `.dll`.

The following rules must be followed:

1. The DLL file must be placed in a valid location to become automatically located by the EMTP-engine.
2. The DLL file name can be any name, the extension `.dll` is assumed.
3. The EMTP-engine does not need to know in advance the name and the contents of the DLL-file.
4. The EMTP-engine does not need to link with the DLL-file.
5. The DLL-file must simply contain a set of callable methods (functions) recognized in the EMTP-engine. If a mandatory interfacing function is missing, it will result into an error message and stop the loading of the DLL-file.
6. EMTP locates the DLL-file, opens it and loads it. There is no mandatory static linking procedure.
7. The DLL-file can be set to be completely independent, but must use some code modules provided in the EMTP installation folder for specific definitions of objects and data related to EMTP.
8. Any number of user-programmed DLLs can be used in a simulation case and each DLL can be used more than once.
9. If a user makes an error in the programming of a DLL it may crash the entire application. This is the negative side of open-architecture codes. In such case it may become necessary to kill the application manually using the Windows Task Manager. All code modules related to EMTP must be killed: `emptworks.exe` (in some cases) and `emptopt.exe`.

Normally the EMTP-engine calls the DLL directly, but it is preferable and easier to program using a Fortran-95 or Fortran-2003 buffer. There are two recommended approaches from top-down in calling sequence:

1. EMTP-engine
2. DLL code written in Fortran-95 and using only Fortran-95 functions and modules.

In the second approach the DLL code can call any other code written in any language:

1. EMTP-engine
2. DLL code written in Fortran-95 (Fortran layer)
3. Any set of code modules written in any language and called from the Fortran layer above.

It is also possible to skip the Fortran layer.

## 3.1 Network equations

### 3.1.1 Main system of equations

Before building DLLs it is important to understand how the DLL can interact with the computational engine of EMTP. This is achieved by allowing the DLL function to access the main system of equations in the same way as the developer of EMTP.

The main system of equations in EMTP is given by:

$$\begin{bmatrix} \mathbf{Y}_n & \mathbf{V}_c \\ \mathbf{V}_r & \mathbf{V}_d \end{bmatrix} \begin{bmatrix} \mathbf{v}_n \\ \mathbf{i}_v \end{bmatrix} = \begin{bmatrix} \mathbf{i}_n \\ \mathbf{v}_b \end{bmatrix} \quad (1)$$

Bold characters are used to denote matrices and vectors. In this system  $\mathbf{Y}_n$  is the nodal admittance matrix. The submatrices  $\mathbf{V}_c$ ,  $\mathbf{V}_r$  and  $\mathbf{V}_d$  are used to include non-nodal type equations, such as branch relations. These are called voltage-defined equations, but can also use current coefficients. The vector of unknown voltages is named  $\mathbf{v}_n$  and unknown currents are given by  $\mathbf{i}_v$ . On the right-hand side of equation (1),  $\mathbf{i}_n$  contains nodal current injections and  $\mathbf{v}_b$  is for determined quantities related to voltage-defined equations.

Equation (1) is similar to writing a general set of equations:

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad (2)$$

where the vector of unknowns is called  $\mathbf{x}$ . Another definition used in the EMTP code is given by the notion of augmented vectors and matrices:

$$\mathbf{Y}_{aug} \mathbf{V}_{aug} = \mathbf{I}_{aug} \quad (3)$$

In the upper part of equation (1) the submatrices  $\mathbf{Y}_n$  and  $\mathbf{V}_c$  are used to state the sum of currents exiting the circuit nodes and the vector  $\mathbf{i}_n$  is for expressing the sum of currents entering each node.

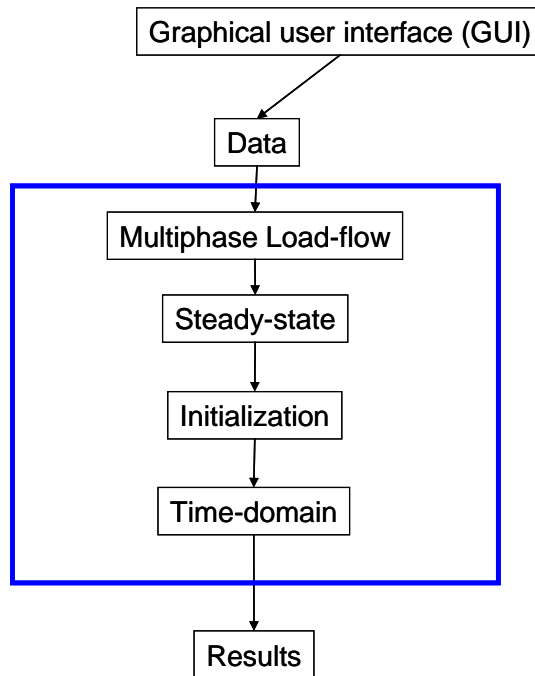
Through the DLL process the user is given access to the creation of  $\mathbf{Y}_n$  and to voltage-defined equations, also called voltage rows for inserting model equations. Once the model creation equations are understood, the procedures for sending the equations to the EMTP-engine are sufficiently simple and repetitive to allow users to build complex models.

The main system of equations (1) is used in both steady-state and time-domain solutions. In the steady-state solution all quantities are complex, whereas in the time-domain solution only real numbers are used.

A separate system is used for the Load-Flow solution, but this is transparent to the DLLs since the standard device equations submitted for the steady-state solution are automatically converted in EMTP for the Load-Flow solution. In this version of the DLL there is no access to the Load-Flow constraint devices (LF devices) as such.

The complete sequence of solver steps is shown in Figure 1. The Load-Flow and steady-state steps are optional. If there is no steady-state solution only manual initialization is available. EMTP is capable of conducting automatic initialization for all devices when the steady-state solution step is available. Any DLL device can provide its steady-state equations, as well as initialization procedures.

In addition to the steady-state solution, EMTP is capable of performing frequency scans. This is similar to performing several steady-state solutions.



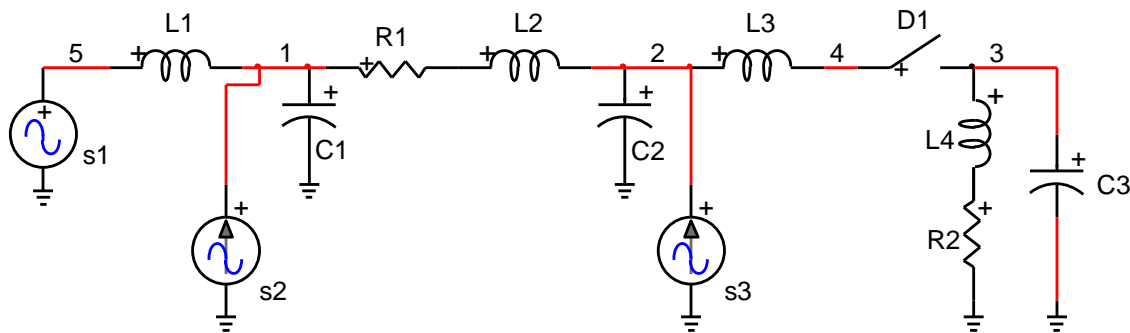
**Figure 1 Main computation modules (steps) in EMTP**

### 3.1.2 Example

The example presented in this section is used to help to understand of the programming of equation (1) for a complete circuit. It allows explaining how the various model equations are inserted into the main system of equations. The procedures are similar to those used in DLLs for specific models or circuits. The studied example is shown in Figure 2. The node numbering is arbitrary.

#### 3.1.2.1 Steady-state

It is assumed that the independent voltage source and the independent current sources are sinusoidal functions. The switch D1 is initially open and thus open in the steady-state solution.



**Figure 2 Sample simple network for demonstrating the building of equations**

For the steady-state solution, equation (1) of this circuit is given by:

$$\begin{array}{cccccc|cc}
 y_{11} & y_{12} & 0 & 0 & y_{15} & 0 & 0 & V_1 \\
 y_{21} & y_{22} & 0 & y_{24} & 0 & 0 & 0 & V_2 \\
 0 & 0 & y_{33} & 0 & 0 & 0 & -1 & V_3 \\
 0 & y_{42} & 0 & y_{44} & 0 & 0 & 1 & V_4 \\
 y_{51} & 0 & 0 & 0 & y_{55} & 1 & 0 & V_5 \\
 \hline
 0 & 0 & 0 & 0 & 1 & 0 & 0 & I_{s1} \\
 0 & 0 & 0 & 0 & 0 & 0 & 1 & I_{D1}
 \end{array} = \begin{array}{c} I_{s2} \\ I_{s3} \\ 0 \\ 0 \\ 0 \\ V_{s1} \\ 0 \end{array} \quad (4)$$

Since this is a steady-state solution, capital letters are used for the unknown vector elements in  $\mathbf{x}$  and the known components in  $\mathbf{b}$ . All quantities are complex numbers (phasors). The ground node is at zero volt and does not participate in these equations. The elements of  $\mathbf{Y}_n$  at the given frequency ( $s = j\omega$ ) are found from:

$$y_{11} = \frac{1}{sL1} + \frac{1}{R1+sL2} + sC1$$

$$y_{12} = -\frac{1}{R1+sL2} = y_{21}$$

$$y_{22} = \frac{1}{sL3} + \frac{1}{R1+sL2} + sC2$$

$$y_{24} = -\frac{1}{sL3} = y_{42}$$

$$y_{33} = \frac{1}{R4+sL4} + sC3$$

$$y_{44} = \frac{1}{sL3}$$

$$y_{55} = \frac{1}{sL1}$$

$$y_{51} = -\frac{1}{sL1} = y_{15}$$

Each passive device is using its own admittance value that is added into the matrix. In the more general case a user-defined model can have an admittance matrix that must be correctly positioned (added) into  $\mathbf{Y}_n$ .

The supplementary equations are added after building  $\mathbf{Y}_n$  by correctly accounting for the connectivity of the devices. The last two lines are voltage defined lines, one for the voltage source and the other one for the ideal switch. In the case of the ideal voltage source the equation of the source is given by:

$$V_5 - 0 = V_{s1} \quad (5)$$

This equation contributes the coefficient 1 at the line 6 and column 5 (6,5) in the left-hand side matrix of equation (4) and the source voltage phasor  $V_{s1}$  to the right-hand side vector. Since the voltage source current must be accounted for in the nodal equations, it is also needed to add the coefficient 1 at the location (5,6) to express the current exciting node 5 and entering the voltage source. This is actually the voltage source current  $I_{s1}$ .

The last line in this system is the switch equation. Since the switch is initially open its current is zero, thus:

$$1 \cdot I_{D1} = 0 \quad (6)$$

The switch is connected between the nodes 4 and 3. Its current  $I_{D1}$  is leaving node 4 and entering node 3. On the left-hand side of equation (4) it is needed to account for the sum of exiting currents at each node, this contributes -1 to (3,7) and +1 to (4,7). The seventh element is the current  $I_{D1}$  coefficient.

The active ideal current sources are contributing to the sum of currents entering nodes 1 and 2 respectively and appear with positive signs at the positions 1 and 2 in the right-hand side vector.

The solution of equation (4) provides all unknown phasors for the given solution frequency. If a given source is active at a different frequency or not active in the steady-state solution, then it is simply killed by short-circuiting (ideal voltage source) or by open-circuiting (ideal current source).

This approach is similar to the one used in the time-domain solution, only now, it is needed to discretize inductances and capacitors using a numerical integration method. This process changes the inductance into an equivalent model named the companion model or the Norton equivalent model. In EMTP there are currently two integration methods: trapezoidal and Backward-Euler. The standard trapezoidal method may be changed into the halved time-step Backward-Euler method to account for discontinuities and eliminate numerical oscillations (instabilities).

### Time-domain

For a given ordinary differential equation:

$$\frac{dx}{dt} = f(x, t) \quad (7)$$

the trapezoidal integration method with a time-step  $\Delta t$  is given by:

$$x_t = x_{t-\Delta t} + \frac{\Delta t}{2} [f_t + f_{t-\Delta t}] \quad (8)$$

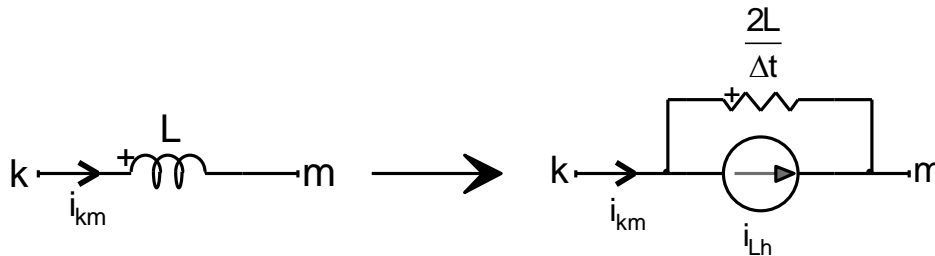
For the case of an inductance connected between two arbitrary nodes k and m:

$$\frac{di_{km}}{dt} = \frac{v_{km}}{L} \quad (9)$$

and according to equation (8):

$$i_{km_t} = \frac{\Delta t}{2L} v_{km_t} + \frac{\Delta t}{2L} v_{km_{t-\Delta t}} + i_{km_{t-\Delta t}} = \frac{\Delta t}{2L} v_{km_t} + i_{Lh} \quad (10)$$

which results into the diagram of Figure 3.



**Figure 3 Numerical discretization of an inductance**

In a similar approach for the capacitor:

$$\frac{dv_{km}}{dt} = \frac{i_{km}}{C} \quad (11)$$

$$i_{km_t} = \frac{2C}{\Delta t} v_{km_t} - \frac{2C}{\Delta t} v_{km_{t-\Delta t}} - i_{km_{t-\Delta t}} = \frac{2C}{\Delta t} v_{km_t} + i_{Ch} \quad (12)$$

Generally speaking the case of the RLC branch can be used as an illustrative example model. For a branch connected between any two nodes k and m, the time-domain equations of the composite branch are given by:

$$i_{km_t} = G_{RLC} v_{km_t} + i_{RLCh_t} \quad (13)$$

$$G_{RLC} = \frac{1}{R + \frac{2L}{\Delta t} + \frac{\Delta t}{2C}} \quad (14)$$

For history current computation steps at time-point  $t - \Delta t$  for a network solution at a following time-point  $t$ :

$$i_{km_{t-\Delta t}} = G_{RLC} v_{km_{t-\Delta t}} + i_{RLCh_{t-\Delta t}} \quad (15)$$

$$v_{L_{km_{t-\Delta t}}} = \frac{2L}{\Delta t} i_{km_{t-\Delta t}} - \frac{2L}{\Delta t} i_{km_{t-2\Delta t}} - v_{L_{km_{t-2\Delta t}}} \quad (16)$$

$$i_{RLCh_t} = G_{RLC} \left[ 2v_{L_{km_{t-\Delta t}}} - v_{km_{t-\Delta t}} + \left( \frac{2L}{\Delta t} + R - \frac{\Delta t}{2C} \right) i_{km_{t-\Delta t}} \right] \quad (17)$$

To initialize from steady-state conditions at  $t=0$  for the first solution at  $t = \Delta t$  it is necessary to use:

$$i_{RLCh_{\Delta t}} = G_{RLC} \left[ 2v_{L_{km_0}} - v_{km_0} + \left( \frac{2L}{\Delta t} + R - \frac{\Delta t}{2C} \right) i_{km_0} \right] \quad (18)$$

(19)

Variables at  $t=0$  are found from the real part of the cosine phasors used in the steady-state solution. If there are several steady-state solutions due to the presence of different source frequencies, then superposition is used and the related quantities must be combined using the real parts of the corresponding Fourier series.

Since EMTP also uses Backward-Euler integration, every model must account for this integration technique when requested by the solver. The solution of equation (7) when using Backward-Euler is given by:

$$x_t = x_{t-\Delta t} + \frac{\Delta t}{2} [f_t] \quad (20)$$

In the RLC branch model the equations for the time-domain solution using halved time-step Backward-Euler integration at time-point  $t$  are identical to (13) and (14), but the history current computation steps at time-point  $t - \Delta t/2$  for a network solution at a following time-point  $t$  are now given by:

$$i_{km_{t-\frac{\Delta t}{2}}} = G_{RLC} v_{km_{t-\frac{\Delta t}{2}}} + i_{RLCh_{t-\frac{\Delta t}{2}}} \quad (21)$$

$$v_{L_{km_{t-\frac{\Delta t}{2}}}} = \frac{2L}{\Delta t} i_{km_{t-\frac{\Delta t}{2}}} - \frac{2L}{\Delta t} i_{km_{t-\Delta t}} \quad (22)$$

$$i_{RLCh_t} = G_{RLC} \left[ v_{L_{km_{t-\frac{\Delta t}{2}}}} - v_{km_{t-\frac{\Delta t}{2}}} + \left( \frac{2L}{\Delta t} + R \right) i_{km_{t-\frac{\Delta t}{2}}} \right] \quad (23)$$

To initialize from steady-state conditions at  $t=0$  for the first solution at  $t = \Delta t/2$  it is necessary to use:

$$i_{RLCh_{\frac{\Delta t}{2}}} = G_{RLC} \left[ v_{L_{km_0}} - v_{km_0} + \left( \frac{2L}{\Delta t} + R \right) i_{km_0} \right] \quad (24)$$

When switching from trapezoidal to Backward-Euler it is necessary to compute  $v_L$  using the trapezoidal equation (16) at the switching time-point. Accordingly when switching from Backward-Euler to trapezoidal, equation (22) must be used first.

If all inductances and capacitors in Figure 2 are replaced by their Norton equivalents then the time-domain system of equations can be formulated as:

$$\begin{bmatrix}
 y_{11} & y_{12} & 0 & 0 & y_{15} & 0 & 0 \\
 y_{21} & y_{22} & 0 & y_{24} & 0 & 0 & 0 \\
 0 & 0 & y_{33} & 0 & 0 & 0 & -1 \\
 0 & y_{42} & 0 & y_{44} & 0 & 0 & 1 \\
 y_{51} & 0 & 0 & 0 & y_{55} & 1 & 0 \\
 \hline
 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 0 & 0 & -1 & 1 & 0 & 0 & 0
 \end{bmatrix}
 \begin{bmatrix}
 v_1 \\
 v_2 \\
 v_3 \\
 v_4 \\
 v_5 \\
 i_{s1} \\
 i_{D1}
 \end{bmatrix}
 =
 \begin{bmatrix}
 i_{s2} + i_{L1h} - i_{C1h} - i_{R1L2h} \\
 i_{s3} - i_{L3h} - i_{C2h} + i_{R1L2h} \\
 -i_{C3h} - i_{R2L4h} \\
 i_{L3h} \\
 -i_{L1h} \\
 v_{s1} \\
 0
 \end{bmatrix}
 \quad (25)$$

Now the history current sources (subscript h) are appearing in vector **b** and contributing to current injections in related nodes. Since history is available from the previous time-point solution, this equation is solved to find all unknown voltages and currents, all history terms are recalculated, the time-step is advanced and a new solution is found. The calculation of the admittance elements is the same as before, only now only real numbers are used. In the case of  $y_{11}$ , for example:

$$y_{11} = \frac{\Delta t}{2L1} + \frac{1}{R1 + \frac{2L2}{\Delta t}} + \frac{2C1}{\Delta t} \quad (26)$$

In equation (25) the ideal switch is now set to the closed position which gives the equation:

$$v_4 - v_3 = 0 \quad (27)$$

The contributed coefficients are -1 at the location (7,3) and +1 at the location (7,4).

#### 4 The request and participation setup

The EMTP computational engine is based on a request-participation setup where the Core Code is exchanging with device models. This concept is illustrated Figure 4. The DLL devices are viewed as any other device models except that their code is completely detached and loaded only on DLL usage.

The Core Code executes a certain number of steps for building and solving the main system of equations. At each step it sends a request to each device model for its participation into the solution process. The device codes are encapsulated in modules and exchange with the Core Code through specific methods. The participation is not mandatory. Some devices may not participate into some requests and still provide the correct solution.

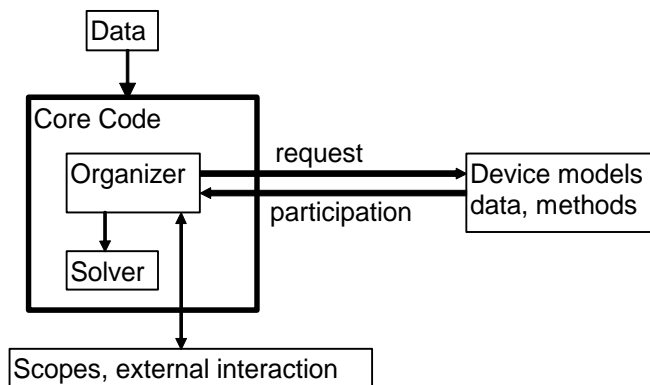


Figure 4 The request-participation design of EMTP



## 4.1 The sequences of Core requests

The Core requests follow a given sequence for all computation modules. This sequence is important since model calculations must be prepared and submitted in a given order. Some requests are not needed in the DLL type devices.

Each Core request has a code number transmitted by a variable.

In the following lines the matrix  $Y_{aug}$  will be designated as  $Yaug$ ,  $Y_n$  as  $Yn$ , the vector  $V_{aug}$  as  $Vaug$  and the vector  $I_{aug}$  as  $laug$ . It is recalled that  $Y_{aug}$  is actually the  $A$  matrix,  $V_{aug}$  is the vector of unknowns  $x$  and  $I_{aug}$  is the vector of known variables  $b$ . The same definitions are applicable to both steady-state and time-domain solutions.

## 4.2 Data Input Requests

The following list presets the sequence of requests during the data input procedures in the Core Code. All request signals are preceded by the keyword "req".

- 1 Read Load-Flow solution if available.
- 2 Initialize all matrices.
- 3 `req_initialize_new`: all devices must read data and perform initialization procedures.
- 4 `req_setup_controllables`: only if at least one controllable.
- 5 `req_setup_observables`: only if at least one observable.

## 4.3 Frequency scan requests

The system of equations is based on complex numbers. The following sequence of calls must be followed.

- 1 `req_put_nodes_in_Yaug`: Symbolic allocation of  $Yaug$ .
- 2 Initialize:  $laug=0$
- 3 `req_put_in_Yaug_ss`: Initialize the voltage-defined sections of  $Yaug$ .
- 4 Select solution frequency
  - 4.a Initialize:  $Yn=0$
  - 4.b `req_put_in_Yn_ss_freqscan`: Send elements into  $Yn$  at the given frequency.
  - 4.c `req_put_in_laug_freqscan`: Send elements into  $laug$  at the given frequency.
  - 4.d Save: at this stage the Core Code will dump results into the binary plot file.
  - 4.e Initialize:  $laug=0$
- 5 Go back to 4 or end if all frequencies are completed.
- 6 End

## 4.4 Steady-state requests

If the user does not request a steady-state solution or there are no active sources in the steady-state solution, then the request is:

`req_zero_initial_conditions`

If the steady-state is requested and can be calculated then the sequence of requests becomes:

- 1 Read Load-Flow solution if available.
- 2 Initialize all matrices.
- 3 `req_insert_source_w_for_ss`: Used to determine for which frequencies it is needed to perform the steady-state solutions.
- 4 `req_put_nodes_in_Yaug`: Symbolic allocation of  $Yaug$ .
- 5 Initialize:  $laug=0$
- 6 `req_put_in_Yaug_ss`: Initialize the voltage-defined sections of  $Yaug$ .
- 7 Select solution frequency
  - 7.a `req_put_in_Yn_ss`: Send elements into  $Yn$  at the given frequency.
  - 7.b `req_put_in_laug_ss`: Send elements into  $laug$  at the given frequency.
  - 7.c `req_superpose_ss_at_w`: Used in devices to prepare history terms from each frequency.
  - 7.d `req_print_ss`: Print the steady-state web for all devices.

- 7.e Initialize: laug=0
- 8 Go back to 7 or end if all frequencies are completed.

## 4.5 Time-domain requests

The steady-state solution is optional and can precede the time-domain solution for automatic initialization of all devices. If there is no steady-state solution then only zero initial conditions are used.

The integration techniques are:

1. TRAP\_EBA: when the selection is Trapezoidal and Backward-Euler
2. EBA: when the selection is Backward-Euler
3. TRAP: when the selection is Trapezoidal

The integration time-step is  $\Delta t$ . The first time-point solution is for  $t = \Delta t$  when TRAP and  $t = \Delta t/2$  when TRAP\_EBA. At  $t = 0$  the solution is found from the steady-state solution. If there is no steady-state then all variables are initialized to 0.

In the following list Core Code procedures are presented together with various requests to allow understanding procedure sequencing.

1. Initialize all matrices.
2. `req_put_nodes_in_Yaug`: optional, if not done in the steady-state solution
3. `req_index_controllables`: sends the index numbers of controlled signals, these indexes allow retrieving back control signal values during the time-domain simulation
4. `t=0 (simulation time)`
5. Initialize: laug=0
6. `req_load_observables_t0`: return all observable signals at t=0 for the next solution of Control System
7. Solve Control System equations at t=0
8. Set `SolMet.OnTimeMesh` for the first time-point solution. It is 0 when `SolMet.TRAP_EBA` is 1.
9. `req_insert_times`: allows transmitting to the core, start and stop times from devices, when known
10. Save scopes at t=0
11. Model history initialization
  - a. `req_EBA_init_at_t0`: when TRAP\_EBA or EBA alone, initialization procedures for the integration methods
  - b. `req_init_at_t0`: when TRAP only, initialization procedures for the integration method
12. `req_put_in_Yaug`: devices send equations into Yaug, voltage-defined section, augmented section
13. `req_put_in_Yn`: devices send individual admittance matrices into Yn
14. If nonlinear devices exist
  - a. Initialize Ynonlin (nonlinear nodes)
  - b. `req_put_nodes_in_Ynonlin`: identification of nonlinear device nodes
  - c. Initialize: Inonlin=0 (the current vector holding contributions from nonlinear elements)
  - d. `req_iter0`: perform first iteration in nonlinear devices. This is an initialization procedure that may help the convergence procedure. The iteration is based only on the steady-state solution.
15. `SimData.rebuild_for_sw=.TRUE.`
16. `SimData.rebuild_for_nonl=.TRUE.`
17. `SimData.rebuild_for_equation=.FALSE.`
18. **Advance to next time-point, time-domain loop**: use full or halved time-step (if TRAP or EBA step).
  - a. Exit time-domain loop if `SimData.tmax` is exceeded
  - b. `req_put_in_laug`: transmission of device contributions into laug
  - c. **Start switch loop** for simultaneous switching option (1 to `SolMet.max_switch_loop`, at least 1):
    - i. If `SimData.rebuild_for_sw` or `SimData.rebuild_for_equation` is true
      1. `req_update_topology`
      2. `SimData.rebuild_for_nonl=.TRUE.`
      3. The matrix Yaug must be refactored
    - ii. If there are no nonlinear devices:
      1. Solve:  $Yaug Vaug = laug$  ( $Ax = b$ )
    - iii. If there are nonlinear devices, start iterative loop for nonlinear devices:
      1. **Start iterative loop** (1 to `SolMet.MaxNumberIter`)

2.  $laugup = laug + Inonlin$
  3. If `SimData.rebuild_for_nonl` is true, refactor `Yaug`.
  4. Solve  $Yaug \text{ Vaug} = laugup$
  5. Exit iterative loop on `Vaug` convergence
  6. Reset `Yaug` back to its original version without nonlinear branches, using `Ynonlin`
  7. Reset: `Inonlin=0`
  8. `req_iter`
  9. Check convergence for all nonlinear devices and exit iterative loop if converged, or exit if the maximum number of iterations is exceeded
  10. Go back, iterative loop
  11. Reset: `Inonlin=0`
  12. `req_convergence_message`: Send convergence error if the maximum number of iterations was exceeded
- iv. `req_load_observables`: return all observable signals at the given time-point for the next solution of Control System
  - v. If first switch loop or `SolMet.resolve_controls`: Solve Control System equations
  - vi. If not Simultaneous switching procedure (`SolMet.no_simultaneous_switch` is true), exit this switch loop
  - vii. If Simultaneous switching procedure (`SolMet.no_simultaneous_switch` is false)
    1. `SimData.rebuild_for_sw` is set to false
    2. `SimData.rebuild_for_equation` is set to false
    3. `SimData.resolve_for_device` is set to false
    4. `req_update_status_at_t`: update device status, used for switches or nonlinear devices for updating operational segment positions
    5. If `SimData.rebuild_for_sw` is false and `SimData.resolve_for_device` is false, exit switch loop
- d. Go back, switch loop
- e. Save plot data
  - f. If there was No Simultaneous switching procedure (`SolMet.no_simultaneous_switch` is true)
    - i. `SimData.rebuild_for_sw` and `SimDat.rebuild_for_equation` are set to false.
    - ii. `req_update_status_at_t`: update device status, used for switches or nonlinear devices for updating operational segment positions
19. Perform the following steps depending on the solution stage (numerical integration method) for the next solution time-point. Preparing for the next time-point solution.
- a. When `SolMet.TRAP_EBA` is 1:
    - i. If `SimData.rebuild_for_sw` is 1, set `SolMet.discon_found` to 1. Other models can set `SolMet.discon_found` independently.
    - ii. Check for possible source discontinuities
    - iii. Determine if the next solution is on the time-mesh: set `SolMet.OnTimeMesh`
    - iv. Perform one of the following according to solution stage:
      1. `req_TRAPtoEBA_update_at_t`: computation of history terms when moving from TRAP to EBA.
      2. `req_EBAtoTRAP_update_at_t`: computation of history terms when moving from EBA to TRAP.
      3. `req_EBA_update_at_t`: computation of history terms when EBA
      4. `req_update_at_t`: computation of history terms when TRAP
  - b. When `SolMet.EBA` is 1: `req_EBA_update_at_t`
  - c. When `SolMet.TRAP_EBA` is 0 (`SolMet.TRAP` is 1): `req_update_at_t`
20. Reset: `laug=0`
21. Go back, time-domain simulation loop

When a statistical simulation is selected, devices must dump and reload memory between statistical shots. This is achieved through the requests:

`req_save_me`: save data into a binary file according to calling sequence

`req_load_me`: reload data from binary file according to calling sequence

More information on the various objects simulation control objects ([SolMet](#), [SimData...](#)) can be found in the file [mysimulation\\_data.f90](#) (folder: [ApplicationDir\EMTP\DLL\\_build\EMTP\\_Release](#)).

## 5 DLL structure

### 5.1 Main structure

A DLL can contain any code, but it must provide specific exported code sections to accommodate requests sent by the Core Code.

Generally speaking the DLL must contain its memory section, various internal functions and exported functions matching the requests:

- 1 Common memory section
22. Common services
23. Used external (provided) services or objects
24. Internal function definitions
25. Exported function definitions

Normally the Exported functions must reside in the DLL file declared in EMTPWorks. All other functions can reside in various files and even use other DLLs, but there is only one DLL code file identified in EMTPWorks. If the model DLL is using other DLLs then it must locate them through its own code.

In addition to the standard requests explained above, the DLL receives the following additional requests:

[req\\_post\\_initialize\\_new](#): allows the DLL to perform extra initialization actions not feasible in

[req\\_data\\_pointers](#): transmission of data pointers into the main program

[req\\_end](#): during this request each DLL can perform termination procedures, such as file closing, nullification of pointers and termination of other external procedures.

If Fortran-95 or Fortran-2003 is used to create the DLL, then a typical DLL file will have the following appearance:

```
MODULE FDLL_DATA
Usage of external modules
DLL memory, services
END MODULE

SUBROUTINE DLL_INITIALIZE_NEW(..)
.....
END SUBROUTINE

SUBROUTINE ...
.....
END SUBROUTINE
```

The subroutines that must reply to request procedures are callable from the Core Code. It means that they must be exported using a compiler specific keyword. In the case of the Intel Visual Fortran Compiler, the keyword is:

```
!DEC$ ATTRIBUTES DLLEXPORT:: DLL_INITIALIZE_NEW
```

In this case the subroutine `DLL_INITIALIZE_NEW` is made callable from the Core Code. The Core Code will locate the callable subroutines automatically and call them using their address. Some requests are mandatory and if the related function or subroutine is not available in the DLL file, it will result into an error message.

The function name related to a request name is simply the replacement of the characters “req” by “DLL”.

A set of modules used in EMTP (Core Code) that are usable also in a DLL model file are available in the folder “`ApplicationDir\EMTP\DLL_build`”. These are Fortran-95 modules.

## 5.2 Building rules

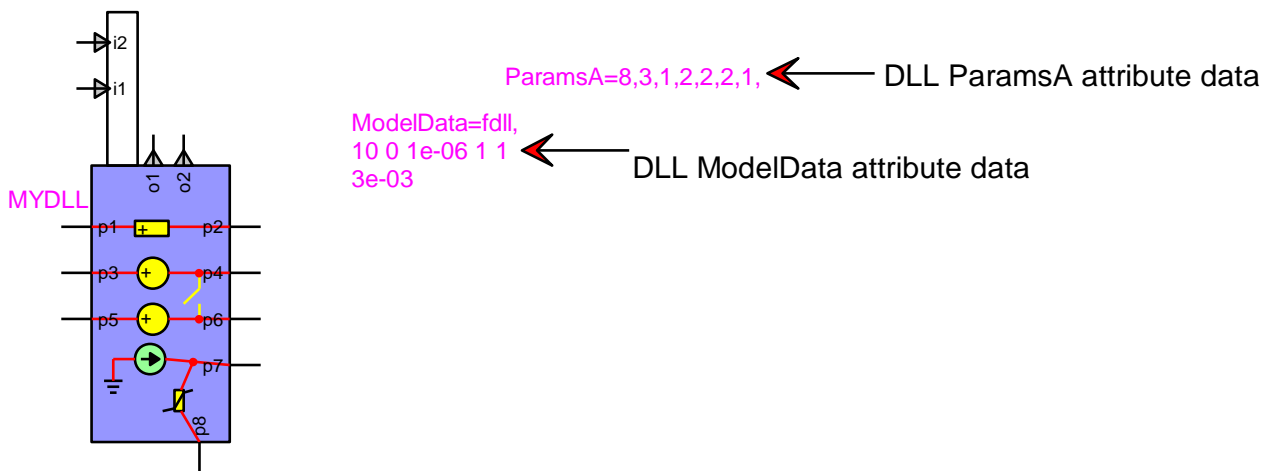
The example of Figure 5 will be used here to illustrate the rules for building a DLL. The following steps must be followed first:

1. Create the DLL symbol using the Symbol Editor or start from an existing device and modify.
  - a. To create from scratch use “File>Design Symbol” and save the part into a library. The Symbol Editor view for the device of Figure 5 is shown in Figure 6. The symbol is used for visualization only since it is meaningless for EMTP. In this example it is used to show the various devices modeled in the DLL for this particular example.
  - b. Use the Symbol Editor to attach pins to the device according to the DLL device contents (see “Fixing pins” item below).
  - c. All attributes must be set to default values shown in Table 1 using the right-click menu Attribute.
  - d. The LibType attribute will be based on the name used for saving into the library. It can be empty when placed into the design. This is more useful for updating from library procedures.
  - e. The Type name can be modified using “Options>Part Type>Make Unique Type”. The Type name is not necessary the Part name. It can be any valid name. In the example of Figure 5 it is named testDLL.
  - f. The Part name (Part attribute) must be set to DLL. This is a unique identifier.
    - i. It is also allowed to use other part names starting with the letters DLL, example DLLyyy. This offers the advantage that all such DLLs will be grouped and called sequentially from EMTP. The DLL developer can benefit from this flexibility to easily maintain relations and data exchanges between DLLs of the same type and/or to create grouped methods.
  - g. The DLL is given a unique name attribute, in this example it is named MYDLL.
2. Fixing pins:
  - a. The pins attached to the DLL device are given an ordinal (pin number). This is shown in the example of Figure 6: pins are numbered according to the order of appearance from top down.
  - b. To check Pin Ordinals after completing the Symbol Editor session: right-click on the pin and select “Pin Info”.
  - c. The pin numbers are important since they allow the DLL developer to establish connections with the contents of the DLL. The symbol is meaningless to EMTP computations, only the pin interface and data attributes count.
  - d. Only 1-phase pins are supported for the DLL devices. To enforce the “General Signal” Line Type on each pin, the DLL developer must set the pin Phase to: =1. The pin attributes can be set through the pin right-click menu Attributes or in the Symbol Editor. This is to avoid DLL users from changing any pin signal to 3-phase and thus causing corruption. It is however possible to connect to an existing 3-phase signal by using 3 pins on the DLL symbol and connecting each pin individually to its target phase.
  - e. Once the pin ordinals are fixed, the pin names are used only for visualization/documentation. In the example of Figure 5 the power pins are given the names p1 to p8 and the control pins are given the names o1 to o2 for output signals (observables) and i1 to i2 for inputs pins (controllables).

The power pins are numbered first (from 1 to 7 in the example of Figure 5 for p1 to p8), followed by control input pins (from 8 to 9 in the example of Figure 5 for i1 to i2) and finally control output-observable pins (from 10 to 12 in the example of Figure 5 for o1 to o2).
  - f. The power pins are set explicitly to the “Pin Function” Power in the Symbol Editor and the control pins are set to Input or Output.
3. ParamsA attribute: This attribute is used to provide specific information to EMTP on the DLL interfacing with the rest of the network. It is a comma separated string with the following data entered either manually or using a user-programmed mask:
  1. Npower\_signals: number of power (not control) signals, 8 in this example
  2. Nvoltage\_type: number of devices that will contribute to the voltage-defined section of Yaug, 3 in this example (2 independent voltage sources and 1 switch)
  3. Ncurrent\_sources: number of independent current sources, 1 in this case

4. Nnonlinear\_nodes: number of nonlinear nodes, it means number of nodes connected to nonlinear devices. There are 2 such nodes in this case.
5. Ncontrol\_signals: total number of control signals (input only), 2 in this case (for pins i1 and i2)
6. Nobserve\_signals: total number of observable (control output) signals, 2 in this case (for pins o1 and o2)
7. Relative\_path\_flag:
  - a. Use 0 to indicate direct access, the DLL file is specified with its full path name.
  - b. Use 1 to indicate relative path usage for the specified DLL file. Using 1 forces EMTP to search the design folder.
  - c. Use 2 to indicate that the specified DLL file must be searched in the location specified by the "DLL Options" device.
  - d. Use 3 to indicate that the specified DLL file must be searched in the Toolbox folder in ApplicationDir.  
This Toolbox folder contains the file Toolbox.ini which specifies searched DLL folders for Toolboxes. Example:  
[DLLs]  
MMC\DLL
  - e. The user may add supplementary search paths by modifying the Windows PATH environment variable.

4. ModelData attribute:
  - a. The first line of ModelData is used to identify the DLL file. The "dll" extension is assumed. Full path must be used when Relative\_path\_flag is 0 and relative (partial) path when 1.
  - b. The following lines are free. These lines can be used to transmit data defined on the screen to the actual DLL. EMTP simply reads these lines and sends them directly to the DLL code through the request `req_initialize_new`.



**Figure 5 User-defined DLL example, testDLL**

It is noticed that the parameters set in ParamsA are related only to equations inserted into equation (1). The device may have many other equations, but must provide only a given set for interfacing with EMTP. This means, for example, that a device may have many current sources, but none of them may be connected directly through a pin, in which case there will be no declared current sources ( $N_{current\_sources}=0$ ).

Table 1 Default Attributes		
Attribute	Value	Example
Exclude		
ExportedMask		
FormData		
FormData1		
LibType	any name, same as Type	
Mask.Dev		
Mask.Dev.Script		
ModelData		fdll, 10 0 1e-06 1 1 3e-03
ModelData1		
ModelDataError		
MPLLevel		
Name	any name	MYDLL
Name.Prefix	any prefix	
ParamsA		8,3,1,2,2,2,1,
ParamsB		
ParamsC		
Part	<b>DLL</b>	<b>DLL</b>
Part.List		
Script.Info.Dev		
Script.Mask.Dev		
Script.Open.Dev		
Sniffer		
Status.Script		
Value		
Value1		

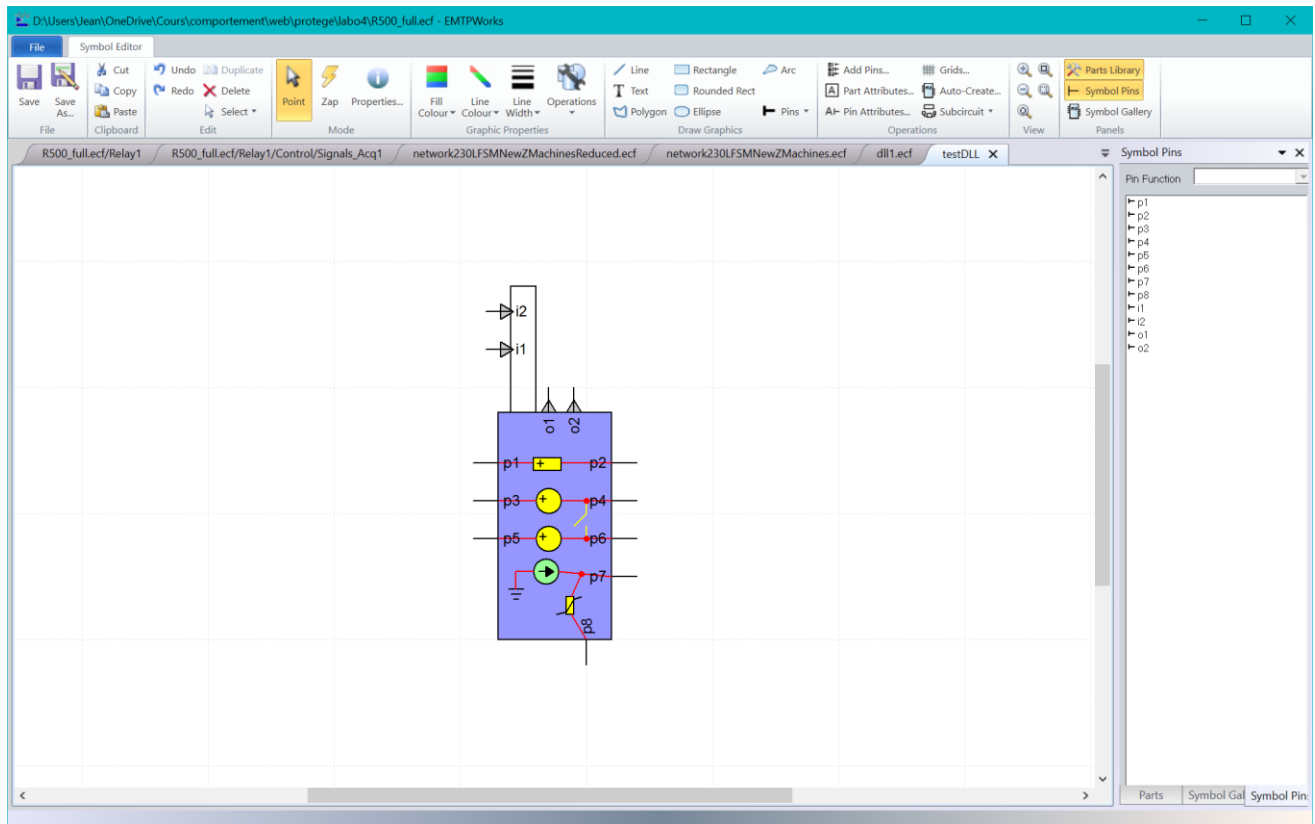


Figure 6 Symbol editor for testDLL

### 5.3 DLL function definitions

For each request transmitted to the DLL, the expected request is based on a function definition. The following is the list of available function definitions:

- 1 **DLL\_INITIALIZE\_NEW**(myname,idev,Data\_Section,DLL\_NAME)
  - 1.a mandatory function
  - 1.b myname: device name in EMTWorks, string
  - 1.c idev: device number in the list of all DLL devices, integer
  - 1.d Data\_Section: Array of strings, contents of ModelData after the DLL file identification
  - 1.e DLL\_NAME: the name of this DLL file name with full path
- 2 **DLL\_POST\_INITIALIZE\_NEW**(myname,idev,power\_signal\_nodes,n\_nodes)
  - 2.a mandatory function, called after DLL\_INITIALIZE\_NEW
  - 2.b myname: device name in EMTWorks, string
  - 2.c idev: device number in the list of all DLL devices, integer
  - 2.d power\_signal\_nodes: array of index numbers for all power signal nodes of the DLL device
  - 2.e n\_nodes: size of the array above
- 3 **DLL\_DATA\_POINTERS**(idev,Pointer\_simulation\_data)
  - 3.a optional function, always called first, before DLL\_POST\_INITIALIZE\_NEW.
  - 3.b idev: device number in the list of all DLL devices, integer
  - 3.c Pointer\_simulation\_data: Pointers to the Simulation Data objects
- 4 **DLL\_INSERT\_SOURCE\_W\_FOR\_SS**(idev)
  - 4.a optional
  - 4.b idev: device number in the list of all DLL devices, integer
- 5 **DLL\_PUT\_VOLTAGE\_ROW\_EQUATIONS**(idev,first\_voltage\_row)
  - 5.a optional, called for req\_put\_nodes\_in\_Yaug request
  - 5.b idev: device number in the list of all DLL devices, integer



- 5.c first\_voltage\_row: index of the first row in the list of voltage-defined equations
- 6 **DLL\_PUT\_IN\_YAUG\_SS**(idev)
  - 6.a optional
  - 6.b idev: device number in the list of all DLL devices, integer
- 7 **DLL\_PUT\_IN\_YN\_SS**(idev,w)
  - 7.a optional
  - 7.b idev: device number in the list of all DLL devices, integer
  - 7.c w: the computation frequency in rad/s, double precision real number
- 8 **DLL\_PUT\_IN\_IAUG\_SS**(idev,w)
  - 8.a optional
  - 8.b idev: device number in the list of all DLL devices, integer
  - 8.c w: the computation frequency in rad/s, double precision real number
- 9 **DLL\_PUT\_IN\_IAUG\_FREQSCAN**(idev,w)
  - 9.a optional
  - 9.b idev: device number in the list of all DLL devices, integer
  - 9.c w: the computation frequency in rad/s, double precision real number
- 10 **DLL\_PUT\_IN\_YN\_SS\_FREQSCAN**(idev,w)
  - 10.a optional
  - 10.b idev: device number in the list of all DLL devices, integer
  - 10.c w: the computation frequency in rad/s, double precision real number
- 11 **DLL\_SUPERPOSE\_SS\_AT\_W**(idev,w)
  - 11.a optional
  - 11.b idev: device number in the list of all DLL devices, integer
  - 11.c w: the computation frequency in rad/s, double precision real number
- 12 **DLL\_PRINT\_SS**(myname,idev,w,Current,Spower)
  - 12.a optional
  - 12.b Errors in this function may cause printed power mismatch
  - 12.c myname: device name in EMTPWorks, string
  - 12.d idev: device number in the list of all DLL devices, integer
  - 12.e w: the computation frequency in rad/s, double precision real number
  - 12.f Current: returned double precision complex vector of currents entering each pin of the DLL
  - 12.g Spower: returned double precision complex vector of S powers entering each pin of the DLL
- 13 **DLL\_INDEX\_CONTROLLABLES** (idev,control\_valindex)
  - 13.a Only when controlled input signals exist
  - 13.b idev: device number in the list of all DLL devices, integer
  - 13.c control\_valindex: integer array, indexes of controlled (input) signals
- 14 **DLL\_LOAD\_OBSERVABLES\_T0**(idev,Returned\_obs\_array)
  - 14.a only when observable signals exist (control output signals)
  - 14.b idev: device number in the list of all DLL devices, integer
  - 14.c Returned\_obs\_array: double precision real array of values sent as observables
- 15 **DLL\_ZERO\_INITIAL\_CONDITIONS**(idev)
  - 15.a optional
  - 15.b idev: device number in the list of all DLL devices, integer
- 16 **DLL\_INIT\_AT\_T0**(idev)
  - 16.a optional
  - 16.b idev: device number in the list of all DLL devices, integer
- 17 **DLL\_EBA\_INIT\_AT\_T0**(idev)
  - 17.a optional
  - 17.b idev: device number in the list of all DLL devices, integer
- 18 **DLL\_PUT\_IN\_YAUG**(idev)
  - 18.a optional
  - 18.b idev: device number in the list of all DLL devices, integer
- 19 **DLL\_PUT\_IN\_YN**(idev)
  - 19.a optional
  - 19.b idev: device number in the list of all DLL devices, integer
- 20 **DLL\_PUT\_NODES\_IN\_YNONLIN**(idev)

- 20.a optional
- 20.b idev: device number in the list of all DLL devices, integer
- 21 **DLL\_ITER0**(idev)
  - 21.a optional
  - 21.b idev: device number in the list of all DLL devices, integer
- 22 **DLL\_PUT\_IN\_IAUG**(idev)
  - 22.a mandatory
  - 22.b idev: device number in the list of all DLL devices, integer
- 23 **DLL\_UPDATE\_TOPOLOGY**(idev)
  - 23.a optional
  - 23.b idev: device number in the list of all DLL devices, integer
- 24 **DLL\_ITER**(idev,convergence\_flag)
  - 24.a optional
  - 24.b idev: device number in the list of all DLL devices, integer
  - 24.c convergence\_flag: return true when converged, return false when did not converge
- 25 **DLL\_CONVERGENCE\_MESSAGE**(idev)
  - 25.a optional
  - 25.b idev: device number in the list of all DLL devices, integer
  - 25.c Allows sending a convergence problem message to the EMTPWorks progress panel
- 26 **DLL\_LOAD\_OBSERVABLES**(idev,Returned\_obs\_array)
  - 26.a only when observable signals exist (control output signals)
  - 26.b idev: device number in the list of all DLL devices, integer
  - 26.c Returned\_obs\_array: double precision real array of values sent as observables
- 27 **DLL\_UPDATE\_STATUS\_AT\_T**(idev)
  - 27.a optional
  - 27.b idev: device number in the list of all DLL devices, integer
- 28 **DLL\_UPDATE\_AT\_T**(idev)
  - 28.a optional
  - 28.b idev: device number in the list of all DLL devices, integer
- 29 **DLL\_UPDATE\_AT\_T**(idev)
  - 29.a optional
  - 29.b idev: device number in the list of all DLL devices, integer
- 30 **DLL\_TRAPTOEBA\_UPDATE\_AT\_T**(idev)
  - 30.a optional
  - 30.b idev: device number in the list of all DLL devices, integer
- 31 **DLL\_EBA\_UPDATE\_AT\_T**(idev)
  - 31.a optional
  - 31.b idev: device number in the list of all DLL devices, integer
- 32 **DLL\_EBATOTRAP\_UPDATE\_AT\_T**(idev)
  - 32.a optional
  - 32.b idev: device number in the list of all DLL devices, integer
- 33 **DLL\_SAVE\_ME**(idev)
  - 33.a optional
  - 33.b idev: device number in the list of all DLL devices, integer
- 34 **DLL\_LOAD\_ME**(idev)
  - 34.a optional
  - 34.b idev: device number in the list of all DLL devices, integer
- 35 **DLL\_END**(idev)
  - 35.a optional
  - 35.b idev: device number in the list of all DLL devices, integer

### 5.3.1 Compiling and linking

Normally the DLL can be created with any compiler. There could be however some incompatibility issues when dealing with libraries and other DLL export related problems. It is also allowed to use any programming

language if the DLL programmer knows how to interface with calls from the Fortran-95 (Fortran-2003) code used in EMTP.

The compiler used in the creation of the provided examples and guaranteed to work with the current EMTP version is "Intel Visual Fortran Compiler" with "Microsoft Visual Studio".

Two basic examples are provided and can be studied for learning how to program EMTP DLLs. The examples are available in the EMTPWorks Examples folder named DLL. The Examples folder can be located using the EMTPWorks menu Examples>Folder.

Standard compiling and linking procedures are used.

If the DLL is set to respond to the request `DLL_DATA_POINTERS`, then it is needed to use an include statement for appropriate subroutine declaration in the DLL:

```
INCLUDE 'EMTP_Release/dll_data_pointers.f90'
```

In addition it is needed to include in the list of project files the file:  
`mysimulation_data.f90`

It contains the source code of the module: `simulation_data`. The request `DLL_DATA_POINTERS` is needed in almost all DLL design cases.

The Fortran-95 files available in `EMTP_Release` are:

1. **default\_precision.f90**: contains default precision selections used in EMTP-engine. Defines parameters such as `krealhp` for double-precision or maximum precision real numbers. It can be optionally used (USE statement) in the DLL for setting the definitions.
2. **dll\_data\_pointers.f90**: must be used when the DLL is set to respond to the request `DLL_DATA_POINTERS`. It is used through the INCLUDE statement presented above. It establishes pointers into the memory of `simulation_data` module.
3. **mysimulation\_data.f90**: contains the DLL version of the `simulation_data` module. It must be used to exchange simulation data with the EMTP-engine. In addition to data, this module provides various service methods (functions). An underscore is used to distinguish such methods. Documentation is available in the presented examples.
4. **sizelimits.f90**: Although EMTP is using full-dynamic memory, it is needed to establish some practical limits in simple arrays used for various data maintenance operations. This module provides the available limits in the EMTP-engine and can be used in the DLL for specifying memory limits of common data.
5. **variable.f90**: Allows to define various useful variables such as `pi` or `twopi` for the DLL usage. This module is for convenience only and the DLL may use its own definitions.

All of the above modules are read-only and should not be modified by the DLL developer.

### 5.3.2 Debugging

Standard DLL debugging procedures are acceptable. The following rules must be followed to debug a DLL:

1. The device using the DLL must select the debug version of the DLL.
2. It is necessary to indicate to the compiler the location of the executable file and to provide the appropriate program arguments. This is achieved through a compiler menu. In the case of "Intel Visual Fortran Compiler" with "Microsoft Visual Studio" (see Figure 7):
  - a. Command: is set to `ApplicationDir\EMTP\emtpopt.exe`
  - b. Command Arguments:  
`ApplicationDataDir\emtpstate.ini;;design_Netlist_file_name;1;`

Four semicolon ";" separated arguments must be specified in the Command Arguments field:

1. The first argument specifies the location of an initialization file for DLL search path settings. This is installation dependent (see `ApplicationDir` in the above Section 2). In this example:  
`C:\Users\p462909\AppData\Roaming\EMTP\emtpstate.ini`
2. The second argument is not used in this version.

3. The third optional argument specifies the design Netlist file name. The Netlist file can be generated using the command (menu) "EMTP>Generate EMTP Netlist". In this example:  
 C:\Users\p462909\Documents\EMTP\DLL\controlled\_source\test\_ICONV.net

4. The fourth argument indicates that the EMTPWorks Progress Panel must be activated.

An optional argument can be simply omitted by terminating it with a semicolon ";", for example:

C:\Users\p462909\AppData\Roaming\EMTP\empstate.ini;;;1;

In this case the Netlist file name is not used. If the Netlist file is not used, then EMTPWorks will open a File Selection panel when the debugging process is started.

It is not possible to debug with the internal code of "emtpopt.exe", but only with the code accessible in the DLL.

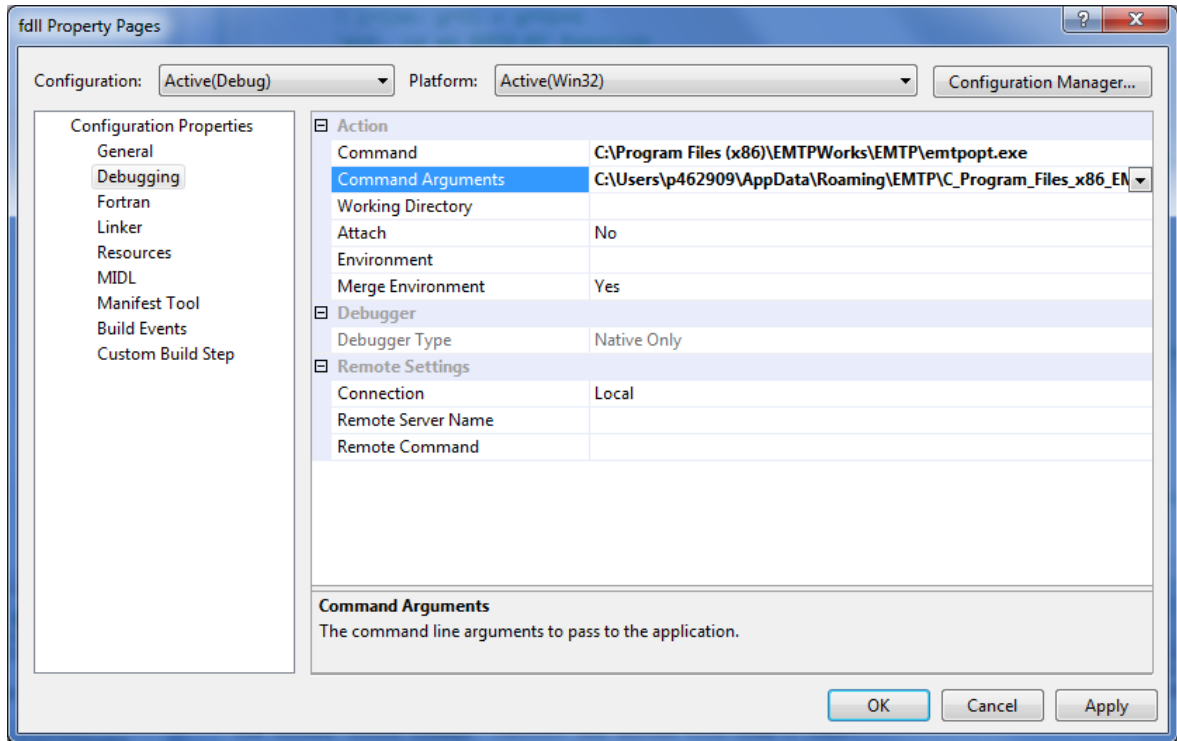


Figure 7 Setup of Command and Command Arguments for DLL debugging

## 5.4 DLL Locations

The DLL search locations are specified by a parameter in the ParamsA attribute (see Section 5.2).

## 6 Example 1

This example can be found in the EMTPWorks Example folder DLL/controlled\_source (see the menu Examples>Folder).

### 6.1 Model

The DLL device modeled in this example is shown in Figure 8. It has only 4 power pins. The pin ordinals are shown on the device. It is a voltage controlled current source. From the pin numbers:

$$I_{34} = g(V_1 - V_2) \quad (28)$$

This is a voltage-defined function given by:

$$I_{34} - gV_1 + gV_2 = 0 \quad (29)$$

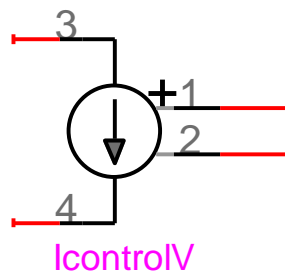
with  $g$  being the gain. The insertion of this equation in equation (1) is given by:

$$\left[ \begin{array}{cccc|c} \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \mathbf{Y}_n & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \hline -g & g & & & 1 \end{array} \right] \begin{bmatrix} \mathbf{v}_n \\ \vdots \\ I_{34} \end{bmatrix} = \begin{bmatrix} \mathbf{i}_n \\ \vdots \\ 0 \end{bmatrix} \quad (30)$$

The nodes 1, 2, 3 and 4 of the device are mapped into the actual network node numbers k, m, i and j respectively and result into:

$$-gV_k + gV_m + I_{ij} = 0 \quad (31)$$

The last row in equation (30) is entering the non-zero coefficients of this equation. The last column is used to maintain the correct sum of currents exiting the nodes i and j. A non-symmetric  $\mathbf{Y}_{aug}$  matrix is created by this example.



**Figure 8 DLL example: Voltage controlled current source**

The ParamsA attribute is given by:  
4,1,0,0,0,0,1,

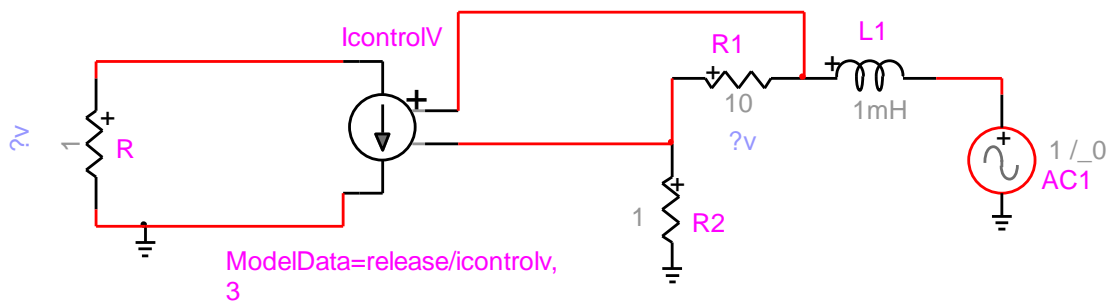
There are 4 power signals, 1 voltage type equation, 0 current sources, 0 nonlinear nodes, 0 control signals, 0 observable signals and Relative\_path\_flag is set to 1.

The ModelData contains the following lines:

```
release/icontrolV,  
3
```

It indicates that the DLL file is named "icontrolV" (icontrolV.dll) and is located in the folder "release" which is in the same folder as the design file "test\_iconv.ecf" shown in Figure 9.

The gain g is entered on the last data line. It is 3 in this example. This line is transmitted directly to the DLL code for internal decoding.



**Figure 9 Test case test\_iconv.ecf**

## 6.2 Folder contents

The folder contents for this example are:

- 1 Debug: folder related to the compiler output for the debug version of icontrolv.dll, automatically created
- 2 EMTP\_Release: this is a copy of the folder "ApplicationDir\EMTP\DLL\_build\EMTP\_Release". The DLL developer must always use the latest version of EMTP\_Release.
- 3 Release: folder related to the compiler output for the release version of icontrolv.dll, automatically created
- 4 test\_ICONV\_pj: project folder for test\_iconv.ecf, automatically created
- 5 fdll.sln: project file used by Microsoft Visual Studio (Intel Visual Fortran Compiler)
- 6 Icontrolv.f90: The Fortran-95 source code file used for building icontrolv.dll
- 7 test\_iconv.ecf: the design file
- 8 test\_iconv.net: the Netlist file, automatically generated when running a simulation or using the menu "EMTP>Generate EMTP Netlist".

This folder contains the complete example, but the only two files are actually needed to run the simulation: test\_iconv.ecf and the referenced DLL file icontrolv.dll. These files can be copied and used elsewhere as long as the reference to icontrolv.dll in test\_iconv.ecf is correctly established.

## 6.3 Debugging

To debug this DLL it is necessary to provide the Command and Command Arguments fields shown in Figure 7. It is also necessary to specify the debug version of the DLL (Debug/icontrolv) in the design file and generate its Netlist accordingly.

## 6.4 Masking for DLL data specification

In this example, the DLL based device is using a scripted mask for specifying the DLL file name and parameter. The following steps are used for adding a mask to Icontrolv:

1. Right-click on the device Icontrolv and select the Attributes menu
2. Select the Script.Open.Dev attribute and change it to script\_black\_box.dwj
3. Click on Done
4. Double-click on the device and add the following code lines in the mask
  - a. Initial values section:  
DLLname='release/icontrolv,'  
gain=3
  - b. Rules section:  
DLLname\_ = DLLname;  
gain\_ = 3
  - c. Variables to transmit section  
DLLname\_,gain\_

It is recalled that in a script\_black\_box mask, if a variable is terminated with the underscore character "\_" then its contents are transmitted directly into the ModelData attribute as a string without adding variable's name.

## 7 Example 2

This example can be found in the EMTPWorks Example folder DLL/fdll (see the menu Examples>Folder).

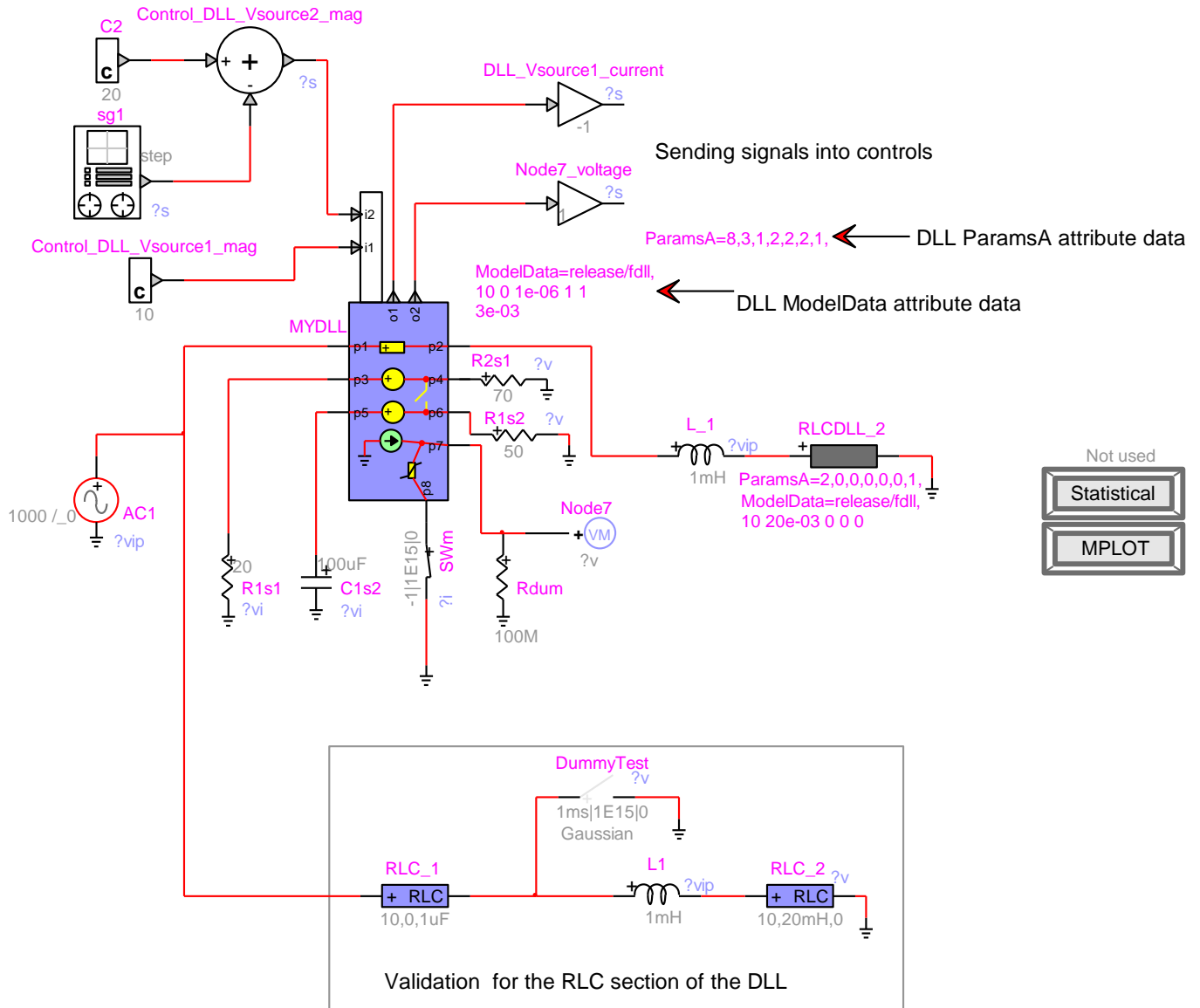
This example is more complex since it uses almost the entire range of options and data exchange features available in a DLL.

The DLL device is the one shown in Figure 5. The DLL file is **fdll.dll**. The test case is named **dll1.ecf** (see Figure 10). Notes:

1. This dummy test case setup is for demonstrating various DLL features.
2. The test case is self-explanatory and based on the source code file **fdll.f90**.

- To demonstrate that a single DLL can be used to develop different types of models, two versions (“a” and “b”) of the DLL are used. RLC DLL\_2 is the simple version “a” and MYDLL is the more complicated version “b”.
- The complicated version has voltage and current sources, a nonlinear function and connects to control signals.
- The definitions of ParamsA and ModelData are available in the module definition section in the file `fdll.f90`.
- Some model data is defined directly in `fdll.f90`. This is the case of the nonlinear device function.

### DLL demonstration example



**Figure 10 Test case dll1.ecf**

Piecewise linear segments are used in the programming of the nonlinear device function. The array of voltage points (all real values) is:

$$v = [-1 \ 0 \ 10 \ 15]$$

The array of current points (all real values) is:

$$i = [-10 \ 0 \ 1 \ 5]$$

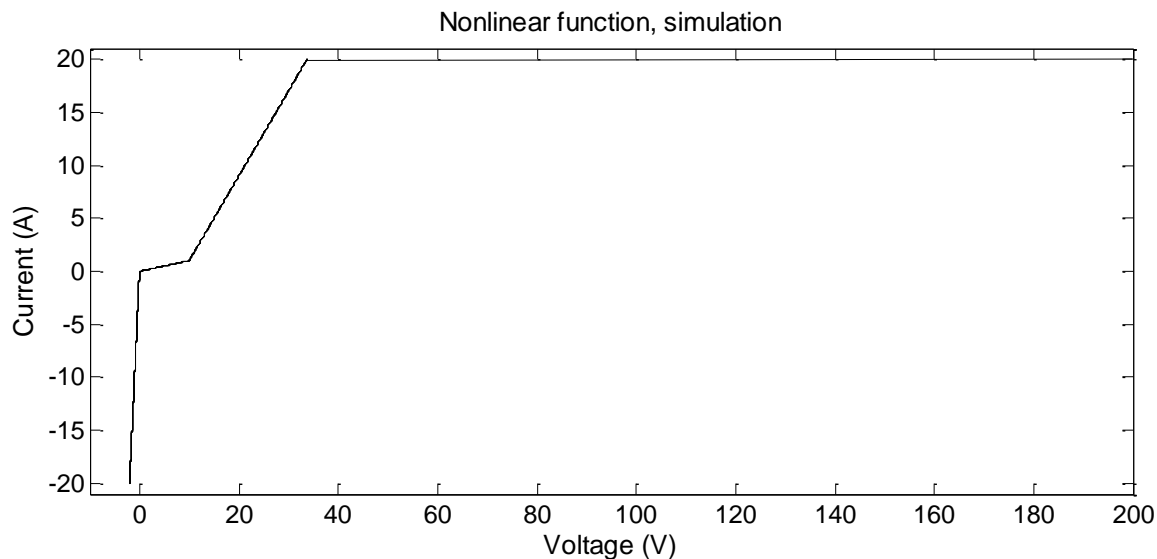
There is a total of 3 segments. The segments must be defined starting with the smallest voltage value and moving to the largest voltage value. EMTP normally accepts only monotonically increasing characteristics, but other types can be defined with some sophistication in the solution programming progress. At every solution time-point it is required to define the operating segment. This segment is found by comparing the time-point voltage solution to the segment starting voltage points. In this case the points are: -1, 0 and 10.

Each segment is given its line equation. In this case there are 3 equations since 3 segments:

1.  $i = 10v \quad 0 > v \geq -\infty$
2.  $i = 0.1v \quad 10 > v \geq 0$
3.  $i = 0.8v - 7 \quad \infty > v \geq 10$

The remaining work is to provide the correct operating segment to EMTP at each simulation time-point. Each segment is a Norton equivalent contributing to  $Y_n$  and to  $i_n$  in equation (1).

In the steady-state computations, the DLL developer may disconnect the nonlinear function completely or use it on its linear segment crossing zero. In this case it was chosen to initialize with segment 2. The actual simulation results are shown in Figure 11. Since segment 2 has an admittance of  $0.1\Omega$  and the current source has an amplitude of 20 A, the initial voltage value becomes 200. It is then moved back onto the correct characteristic segment at the first simulation time-point.



**Figure 11 Nonlinear function simulation**

A dummy statistical switch named DummyTest can be included to test the correct programming for the statistical requests DLL\_SAVE\_ME and DLL\_LOAD\_ME. The trick is based on requesting two statistical simulations (Statistical device) and saving both waveforms. Since these waveforms are not affected by the DummyTest switch position, they should be identical. If they are not identical or an error message occurs, then the saving and loading of some time-dependent variables in the DLL are missing or incorrectly positioned.