Creation VSM

Models of electronic components for Proteus.

Part I. Digital models.Introduction.

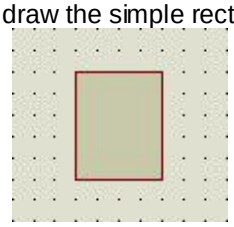
Proteus of firm Labcenter Electronics is the simulator of electronic circuits, based on Berkeley SPICE3F5 with the expansions for the simulation of digital and analog-digital diagrams. In the standard libraries Proteus the sufficiently broad band of the models of components is represented, but the need of creating its own models appears for some diagrams. In this article we will examine the process of their creation.
I am oriented to VSM API version 1.10, realized in Proteus 6.3, since in the later versions from the delivery is excluded VSM SDK. Version Proteus, on which I will explain - Proteus Professional OF '..SHCHSPY, since this is the last available to me version. You be careful - can arise (they will arise) some complexities in the work with the insufficiently treated nelitsenzionnymi versions.
Also, will be examined the creation only of digital VSM - models. Models PSPICE, analog, mixed, active, etc. will be described in the following articles.
Note. Before the reading of this article it is desirable to have an experience on the creation of diagrams in medium Proteus ISIS, and also Proteus themselves it is desirable to version not lower than 6.3, is better than Professional, is possible Lite. Demo- version will not approach because of the fact that in it the functions of the retention of the results of work are completely blocked. Is still necessary the experience of programming on C++, including knowledge of the objective- oriented programming.

Interaction Proteus ISIS VSM - by models.

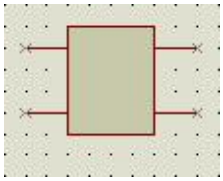
VSM- model is library..DLL, written in language C++. Is desirable to use compiler Microsoft Visual C++ 7.0 (it it is accessible for the free load from site Microsoft), or another compiler C++, for example Digital Mars. Delphi will not approach because of the impossibility of the call of the members of classes, compiled C++. I I will write version for Microsoft Visual C++, for other compilers the alteration will be not very complex.
Also, in order to map onto the screen of components and to interact with it, it is necessary to introduce the visual model of component into the library of components Proteus.
With the starting of the process of simulation ISIS it analyzes the parameters of visual model, it podgruzhayet corresponding..DLL, it initializes it, as a result of which must be created the object of the corresponding class, and then in the process of initialization and simulation the exchange of data with the methods of the created object occurs.

Visual model.

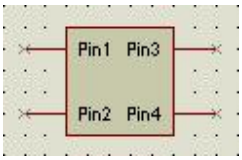
The visual model of electronic component is its schematic idea and description of its properties. It includes the figure of element ("housing" and of inscription), conclusions with the unique names, and also the "script", in which are described the properties of component.
the "housing" of element is created with the aid of 2d- elements (2D graphics line, 2D graphics Box, and so on). For example, let us draw the simple rectangle:



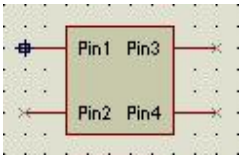
Let us add conclusions. This is done in the division "Device pin.". Let us take for example DEFAULT:



Let us name them Pin1- Pin4:



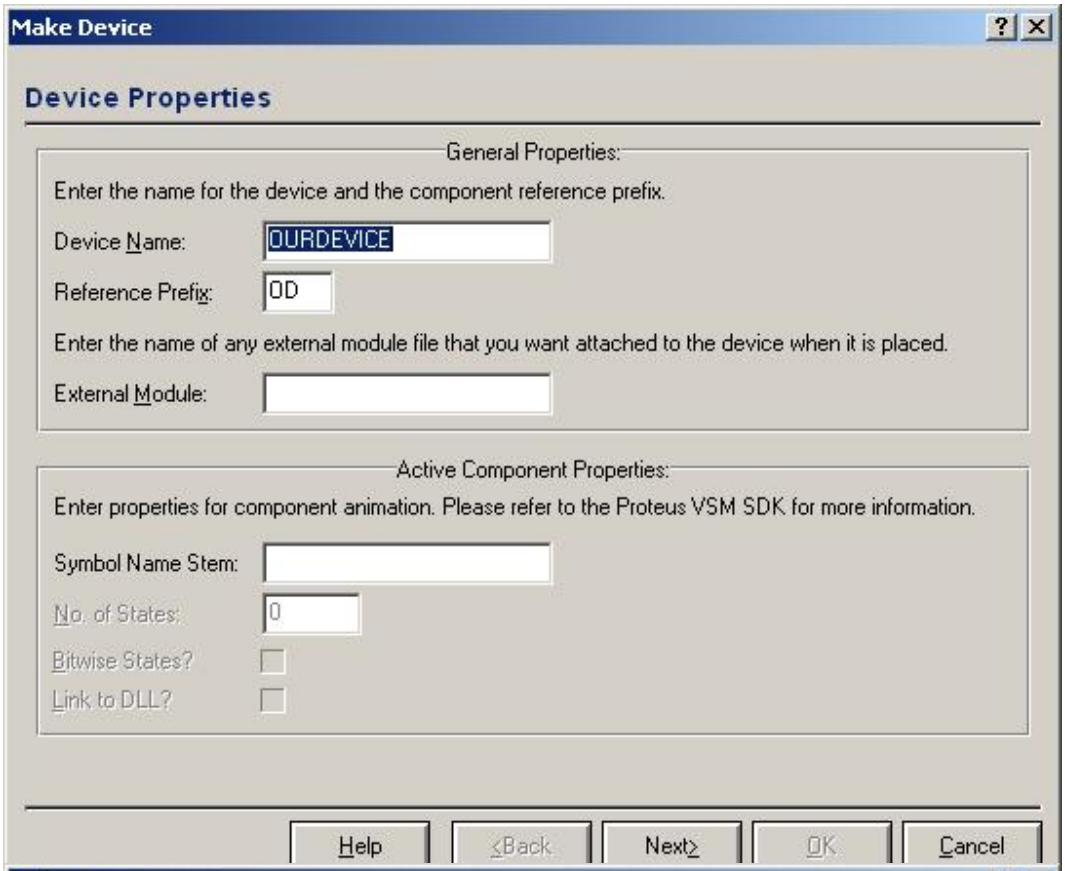
Let us establish Origin - point, with the aid of which the component will be equalized along the reference grid. Let us select in the division "Markers for component origin, etc" ORIGIN and will establish it to the external end of conclusion Pin1:

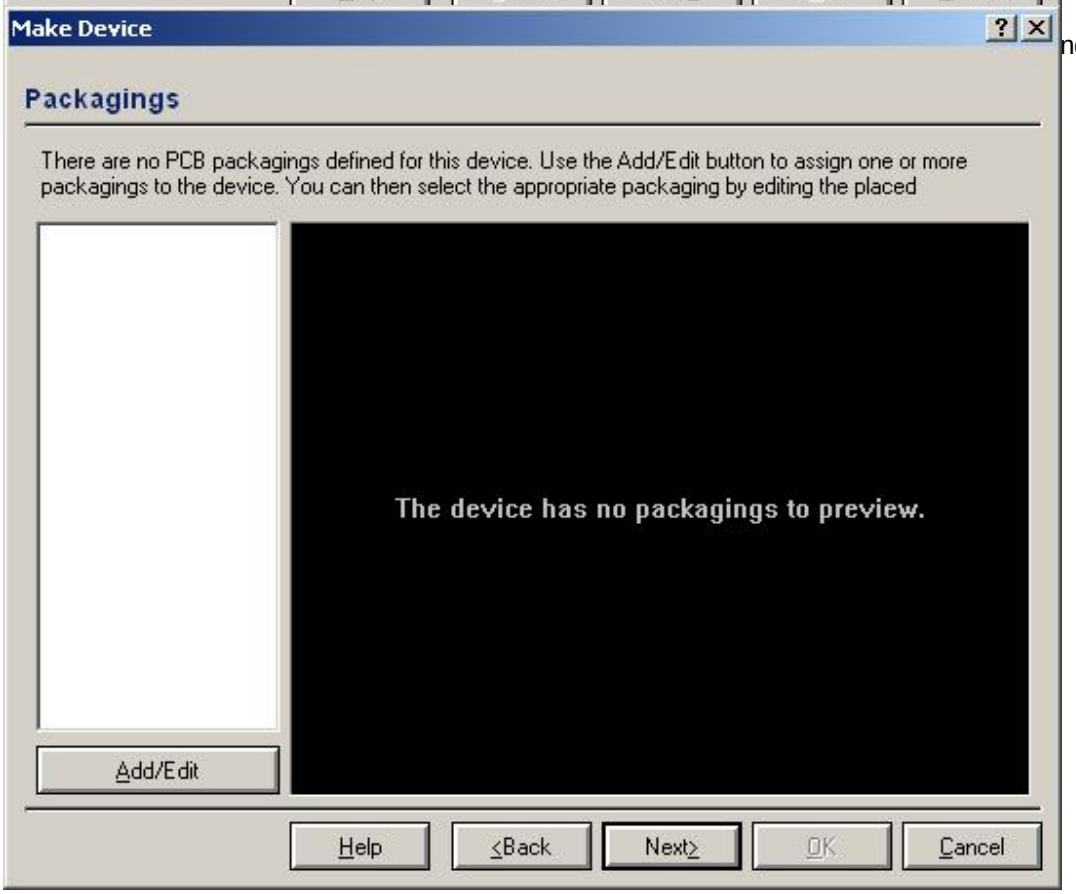


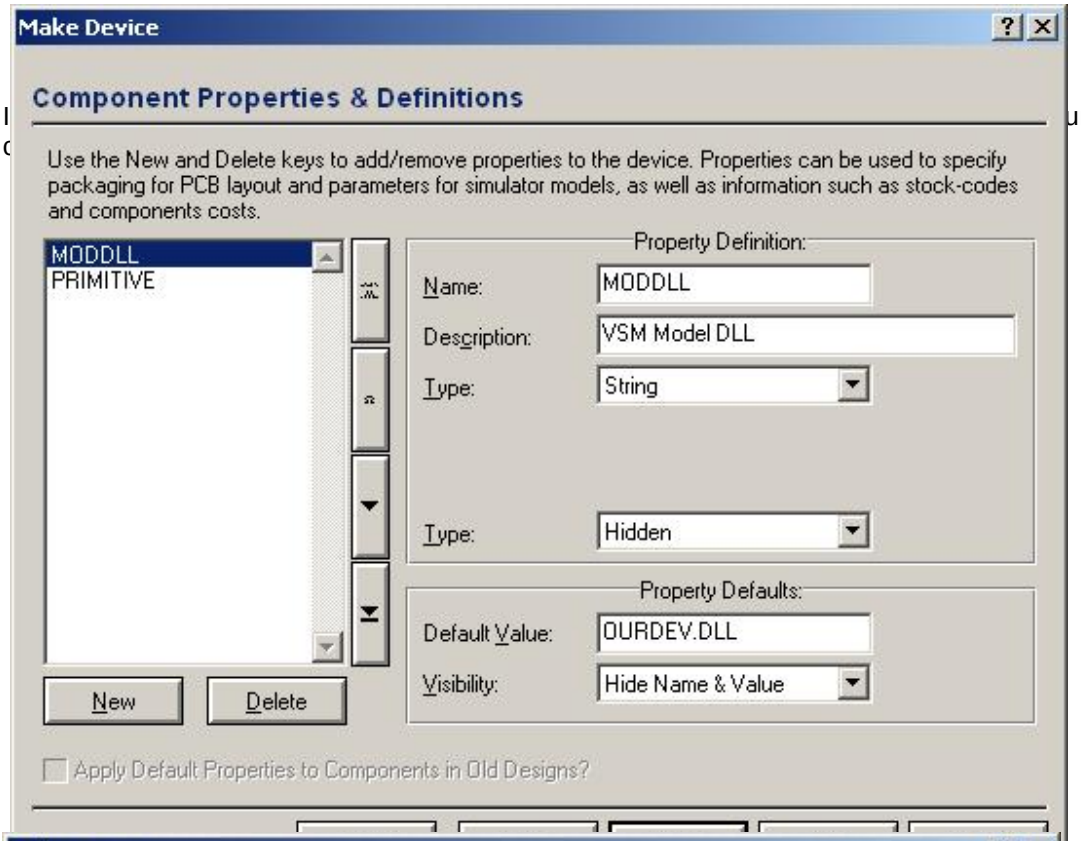
Now let us write script. For this let us establish (usually lower than the component)"Text script". In the opened window let us introduce the following:

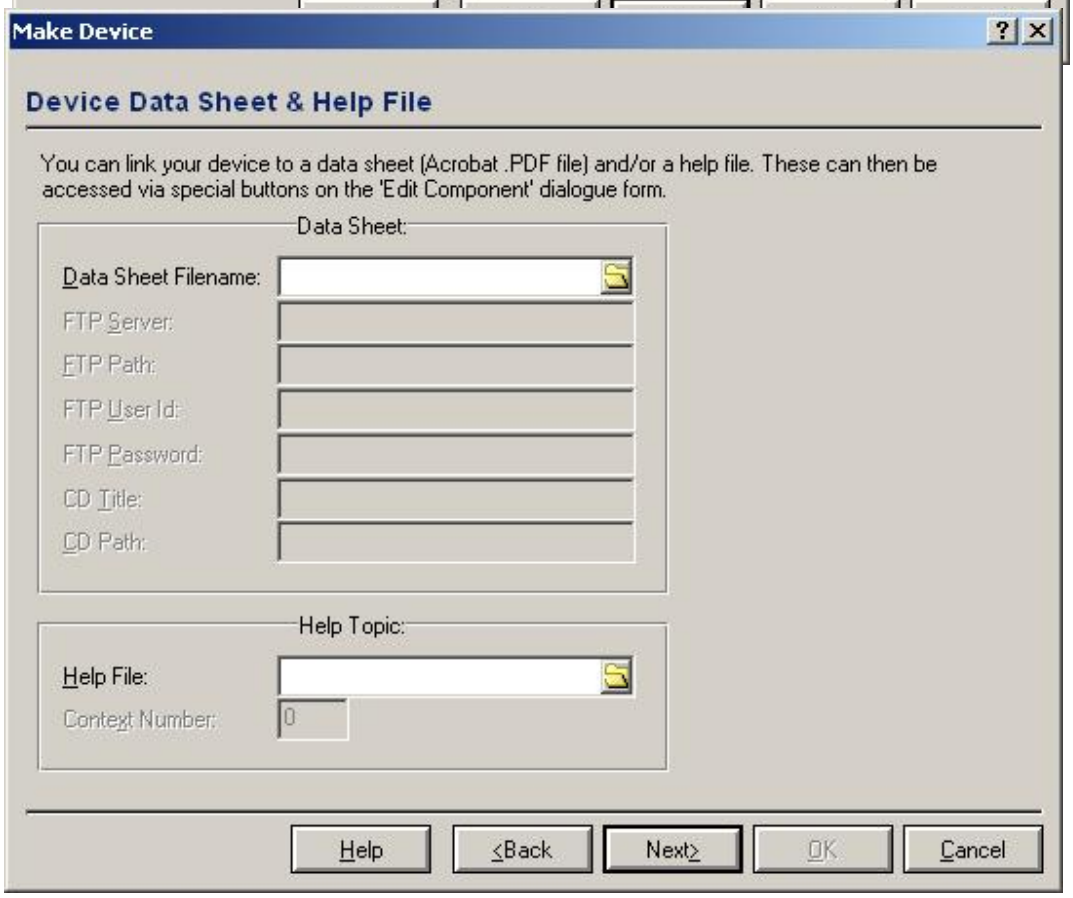
```
{* DEVICE} NAME=.OURDEVICE {PREFIX=.OD} {* PROPDEFS}
{MODDLL=".vsm Model DLL".yuidden STRING} {PRIMITIVE=".Primitive
Type".YuIDDEN STRING} {* COMPONENT} {MODDLL=.OURDEV.DLL}
{PRIMITIVE=.DIGITAL, to.ourdeviche}
```

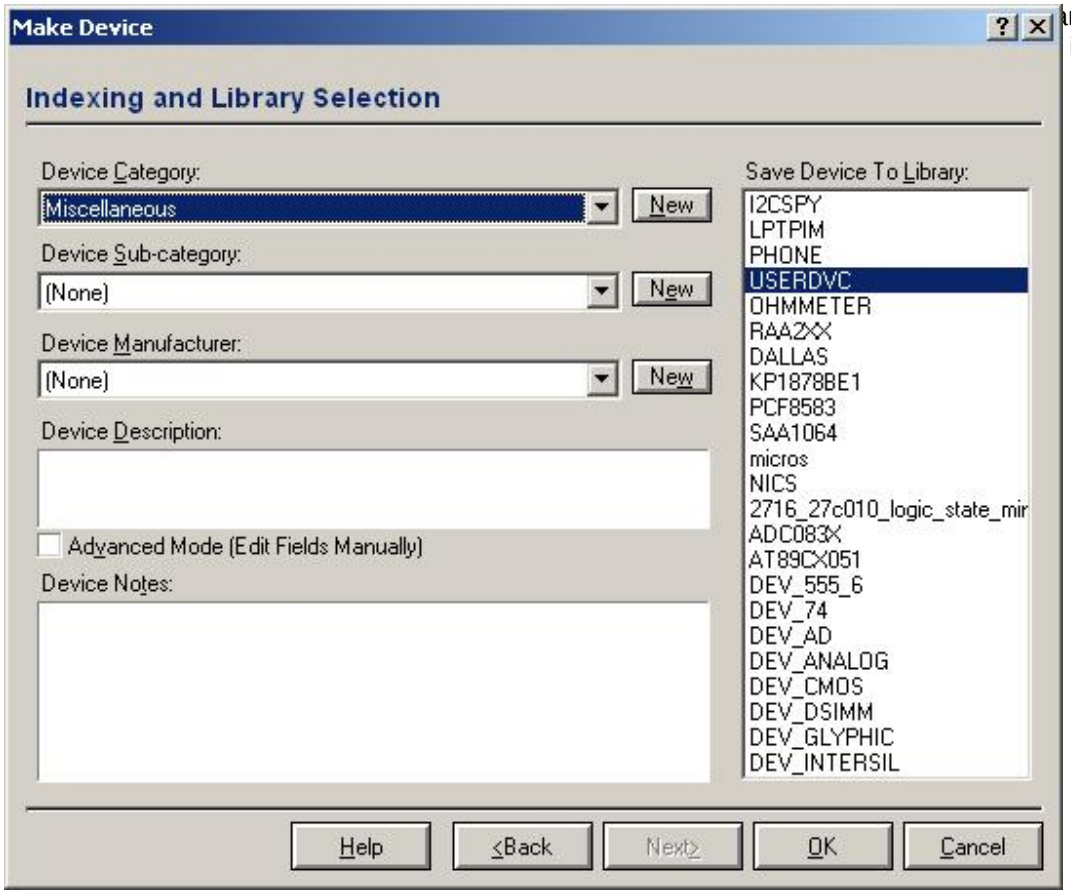
Now I will explain that this it indicates.
{*...} - name of division, further go the parameters.
{...} - concealed parameter, i.e., it not will be mapped into the properties of component.
In the division {* DEVICE} are indicated the name of component (NAME), its prefix (PREFIX) with the creation of the copy of the component (in our case this it will be OD1, OD2 and so forth).
In the division {* PROPDEFS} are transferred the properties of object. this MODDLL - indication that our model will be stored in..DLL, and PRIMITIVE - indication that in our of object there will be the name and type. Further in the division {* COMPONENT} these properties are filled up. In our case indicated, that the model will be stored in file OURDEV.DLL and by the following line PRIMITIVE=.DIGITAL, to.ourdeviche we show that this there will be digital model with name OURDEVICE.
There are many additional properties, but for creating the simple model to us that will be sufficient that we wrote.
It is now necessary to arrange the separate elements into single whole - visual model and to place it into the library. We separate all our components and we select menu Library - Make Device or we press on the button "Form tagged graphics/pads into device and place in library".


Make Device
Device Properties
General Properties:
Enter the name for the device and the component reference prefix.
Device Name:
OURDEVICE
Reference Prefix:
OD
Enter the name of any external module file that you want attached to the device when it is placed.
External Module:
Active Component Properties:
Enter properties for component animation. Please refer to the Proteus VSM SDK for more information.
Symbol Name Stem:
No. of States:
0
Bitwise States?
Link to DLL?
Help
<Back
Next>
OK
Cancel


Make Device
Packagings
There are no PCB packagings defined for this device. Use the Add/Edit button to assign one or more packagings to the device. You can then select the appropriate packaging by editing the placed
The device has no packagings to preview.
Add/Edit
Help
<Back
Next>
OK
Cancel

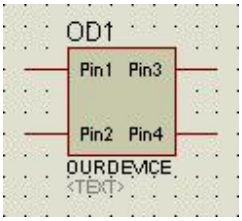

Make Device
Component Properties & Definitions
Use the New and Delete keys to add/remove properties to the device. Properties can be used to specify packaging for PCB layout and parameters for simulator models, as well as information such as stock-codes and components costs.
MODDLL
PRIMITIVE
Property Definition:
Name:
MODDLL
Description:
VSM Model DLL
Type:
String
Type:
Hidden
Property Defaults:
Default Value:
OURDEV.DLL
Visibility:
Hide Name & Value
New
Delete
Apply Default Properties to Components in Old Designs?


Make Device
Device Data Sheet & Help File
You can link your device to a data sheet (Acrobat .PDF file) and/or a help file. These can then be accessed via special buttons on the 'Edit Component' dialogue form.
Data Sheet:
Data Sheet Filename:
FTP Server:
FTP Path:
FTP User Id:
FTP Password:
CD Title:
CD Path:
Help Topic:
Help File:
Context Number:
0
Help
<Back
Next>
OK
Cancel



And in the last window we finally select Device Category - Miscellaneous, and library USERDVC. After pressure on button OK our component is placed into the library, and it is possible to establish it on the schematic diagram. So let us make.

This is what must come out as a result:



But here is misfortune - with the attempt to neglect the simulation of diagram the communication about the error jumps out:

```
SIMULATION LOGS
==============
...
FATAL: [ OD1 ] External model DLL "OURDEV.DLL" not
found. GLE=.0x00000002.
Simulation FAILED Due To Fatal Simulator Errors.
```

This means that ISIS could not find OURDEV.DLL - library, in which is located our model. The creation of this library we will study in the following chapter.

Creation Vsm- of model.

For the beginning let us create empty project..DLL - library Win32. Let us include in project the file of the titles of classes VSM SDK (these files they lie at catalog INCLUDE):

```
# include "Vsm.hpp"
```

Let us add two functions:

```
extern "C" IDSIMMODEL __ declspec (dllexport) *
createdsimmodel (CHAR * device, ILICENCESERVER * ils)
{
        ils->.authorize(.model_.key);
        Return NULL;
}
extern "C" VOID __ declspec (dllexport)
deletedsimmodel (IDSIMMODEL * model) {}
```

The first function transfers to our library the name of the device, for creating which ISIS is caused our of..DLL- ku. This is convenient to use, if our library realizes several different models. It must cause function ILIccENccESERVER::authorize of the server of licensing and report to it its key, in the case of success create object and return his address. In our case we return NULL, since the description of model is not yet prepared. In more detail about the keys of licensing I will describe more lately.

The second function must remove model from the memory. We it still do not have; therefore it is possible anything not to make.

Let us compile our model and will look, which will come out. By the way, it is possible to add way to our..DLL in window System - Set Paths...

Now simulator is scolded by other slightly communication:

```
ERROR: [ OD1 ] OURDEV.DLL failed to create DSIM
model for primitive type "OURDEVICE '.
```

Well it is correct, we create no object.

Now small observation. All digital VSM - models are inherited from the abstract class IDSIMMODEL. Let us create and we class DSOURDEVICE as heir IDSIMMODEL:

```
Class DSOURDEVICE: Public IDSIMMODEL
{
public:
        INT isdigital (CHAR * pinname);
        VOID setup (IINSTANCE * inst, IDSIMCKT * dsim);
        VOID runctrl (RUNMODES mode);
        VOID actuate (REALTIME time, ACTIVESTATE newstate);
        BOOL indicate (REALTIME time, ACTIVEDATA * data);
        VOID simulate (ABSTIME time, DSIMMODES mode);
        VOID callback (ABSTIME time, EVENTID eventid);
private:
        IINSTANCE * instance;
        IDSIMCKT * ckt;
        IDSIMPIN * Pin1;
        IDSIMPIN * Pin2;
        IDSIMPIN * Pin3;
        IDSIMPIN * Pin4;
};
```

In division public we overlap the abstract methods of class, while in division private we have to be stored references to different internal structures.

Then let us add the application of the methods:

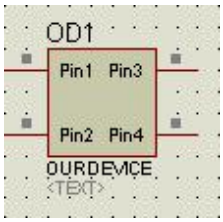
```
INT DSOURDEVIccE::isdigital (CHAR * pinname)
{
        return 1;
}
```

```
VOID DSOURDEVIccE::setup (IINSTANCE * instance, IDSIMCKT * dsimckt)
{
        inst = instance;
        ckt = dsimckt;
        Pin1 = inst->.getdsimpin(".Pin1", true);
        Pin2 = inst->.getdsimpin(".Pin2", true);
        Pin3 = inst->.getdsimpin(".Pin3", true);
        Pin4 = inst->.getdsimpin(".Pin4", true);
}
VOID DSOURDEVIccE::runctrl (RUNMODES mode)
{
}
VOID DSOURDEVIccE::actuate (REALTIME time, ACTIVESTATE newstate)
{
}
BOOL    DSOURDEVIccE::indicate    (REALTIME    time,    ACTIVEDATA    *    data)
{
        Return FALSE;
}
VOID DSOURDEVIccE::simulate (ABSTIME time,DSIMMODES mode)
{
}
VOID DSOURDEVIccE::callback (ABSTIME time, EVENTID eventid)
{
}
```

Also, let us change function createdsimmodel and deletedsimmodel, since the description of the model is already prepared:

```
extern "C" IDSIMMODEL __ declspec (dllexport) *
createdsimmodel     (CHAR     *     device,     ILICENCESERVER     *     ils)
{
        if (ils->.authorize(.model_.key)
                return new DSOURDEVICE;
        else
                Return NULL;
}
extern "C" VOID __ declspec (dllexport)
deletedsimmodel (IDSIMMODEL * model)
{
        delete (DSOURDEVICE *).model;
}
```

Let us compile model and will try to neglect simulation in ISIS. Hurray! It earned! One but - no results of work evidently:



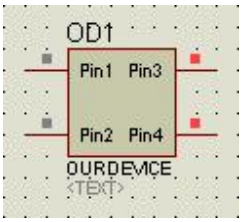
Addition of functionality.

First let us be determined, that Pin1 and Pin2 we have the entrances, and Pin3 and Pin4 - outputs.

Let us establish the initial state of outputs in "1". For this let us add the following lines into the end of function setup:

```
Pin3->.setstate(.SHI);
Pin4->.setstate(.SHI);
```

Perekompiliruyem and let us neglect simulation. It is evident that changed the initial state of our outputs - they were established in "1":



Let us make from our model a generator. For this let us add into the end of function setup the line

```
ckt->.setcallback(1000000000000, this, 0x25);
```

To these we assign, that in 1 second (time it is assigned in the picoseconds) after starting will be caused function callback with the code of event 0x25.

Also, let us add into function callback:

```
VOID DSOURDEVIccE::callback (ABSTIME time, EVENTID eventid)
{
        if (eventid == 0x25)
        {
                if (ishigh(.Pin3->.istate()))
                        Pin3->.setstate(.time, 1, SLO);
                Else
                        Pin3->.setstate(.time, 1, SHI);
                ckt->.setcallback(.time + 1000000000000, this, 0x25);
        }
}
```

I.e., if our event is caused, then we change the state of output to the opposite, and we start timer again for 1 second, after which again will be caused function callback.

Let us compile and will neglect simulation. So there is - model changes the state of leg Pin3 each second.

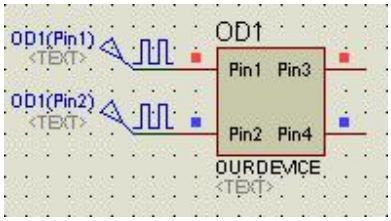
Now let us try to force our model to read state it is input and to reflect them to the outputs. Let us agree, which Pin3 will reflect state Pin1, and Pin4 - state Pin2.

For the beginning let us lead functions setup and callback into the initial form. Then let us add two lines into function simulate:

```
Pin3->.setstate(.time, 1, Pin1->.istate());
Pin4->.setstate(.time, 1, Pin2->.istate());
```

I.e., we read the value of state it is input Pin1 and Pin2, and through 1 value of clock frequency (1 picosecond) relative to current time (time) we establish the state of outputs. Certainly, in the reality it does not occur such rapid devices, so that it is necessary to assign the great significances of delays. It is better, if these values are assigned through the constant. It is still better, if they are be transferred as the parameters of model (against this later).

And let us fit to our entrances two digital generators. We see what - outputs reflect state it is input:



Now you learned to create digital models.

Application. The source texts to the article à http://www.callbus.ru/articles/models/part1/ourdev.zip