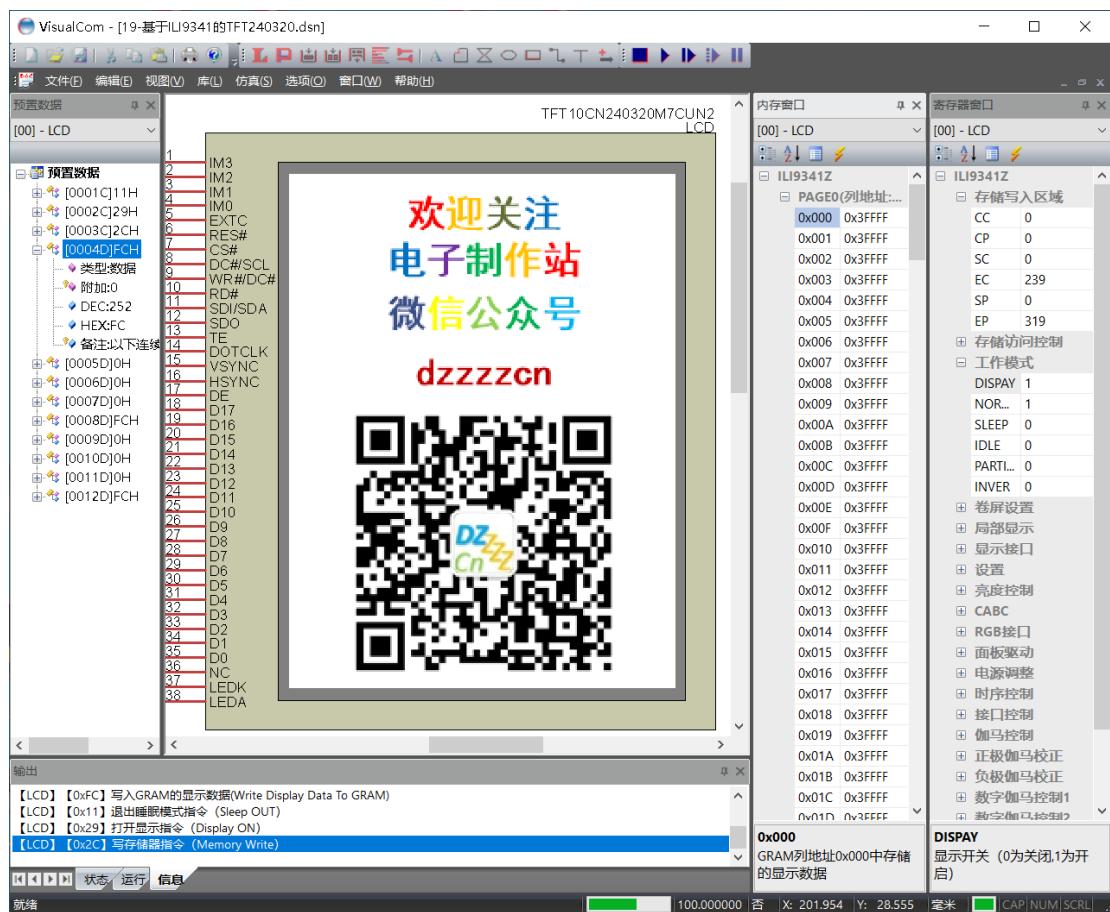


# VisualCom 软件平台

## 仿真模型开发手册 V2.1.0



第 1 章 概述.....	5
第 2 章 仿真模型架构.....	6
第 3 章 灯泡元件.....	8
3.1 最简洁的开发流程 .....	9
3.1.1 创建元件库（可选） .....	9
3.1.2 创建元件 .....	10
3.1.3 创建图形 .....	13
3.1.4 新建动态链接库工程 .....	19
3.1.5 仿真模型开发 .....	23
3.1.5 将 DLL 文件拷贝到 model 文件夹中.....	36
3.1.6 运行仿真 .....	36
3.2 优化灯泡仿真模型 .....	38
3.3 自定义灯泡的颜色 .....	42
3.4 将灯泡状态添加到寄存器窗口 .....	47
3.5 可控制的闪烁灯泡 .....	51
3.6 单击动作改变灯泡的状态 .....	57
3.7 按键动作改变灯泡的状态 .....	59
第 4 章 与远程模块连接.....	60
4.1 元件创建 .....	60

4.2 模型开发 .....	63
4.2.1 声明引脚（或总线） .....	64
4.2.2 设置引脚（或总线） .....	65
4.2.3 获取或设置引脚（或总线）信号 .....	66
4.3 与硬件模块联合仿真 .....	68
4.4 更多细节 .....	69
4.4.1 多个引脚数据读取 .....	69
4.4.2 输出数据 .....	69
4.4.3 更改引脚的电气类型 .....	70
4.4.4 主动读取数据 .....	71
4.4.5 获取引脚的分配状态 .....	72
4.4.6 设置数据电平延时 .....	72
4.4.7 错误编程 .....	73
4.4.8 调试信息 .....	74
第 5 章 API 函数 .....	76
5.1 IDSIMMODEL .....	76
5.1.1 获取仿真模型 .....	76
5.1.2 获取元件指针 .....	76
5.1.3 线程相关 .....	77

5.1.4 初始化相关 .....	78
5.1.5 解析预置数据 .....	79
5.1.6 动画 .....	80
5.1.7 绘制图形 .....	80
5.1.8 按键或鼠标交互 .....	81
5.1.9 其它 .....	81
5.2 ICOMPONENT .....	82
5.2.1 获取图形信息 .....	82
5.2.2 获得用户自定义属性信息 .....	85
5.2.3 获得系统信息 .....	91
5.2.4 获取或设置元件状态信息 .....	92
5.2.5 绘制图形 .....	94
5.2.6 设置图片与读取像素颜色 .....	104
5.2.7 控制声音播放 .....	105
5.2.8 字符串格式转换与比较 .....	106
5.2.9 与远程模块交互 .....	107
5.2.10 缓存控制 .....	115
第 6 章 常见问题 .....	117
版本历史 .....	119

# 第 1 章 概述

本文档详述使用 VisualCom 平台开发元件及相应仿真模型的具体过程，总体可划分为以下四部分：

其一，简单介绍仿真模型的总体框架，以帮助你更清晰地理解仿真模型是如何运行的。

简单地说，本章能够让你明白应该在哪个地方进行哪些编程。

其二，首先通过简单的灯泡元件初步认识仿真模型的开发。此部分使用最少步骤创建出最简单（也是有缺陷）但可以仿真的元件（0 为熄灭，1 为点亮）。然后进一步优化灯泡元件，包括根据用户输入的自定义属性修改灯泡的颜色、设计闪烁类型的灯泡、通过鼠标单击或按键控制灯泡状态以及其它更复杂的元件，以进一步深刻理解各种 API 函数的应用。

其三，详细阐述如何与远程硬件模块进行交互，包括获取/设置/判断引脚（总线）的电平数据等。

其四，仿真模型开发相关的 API 函数原型说明。

本文所述之“系统”即指 VisualCom 软件平台

本文涉及的 vsm.h 头文件位于 VisualCom 软件平台安装目录 vsm 文件夹

本文涉及的源代码位于 VisualCom 软件平台安装目录 src 文件夹

# 第 2 章 仿真模型架构

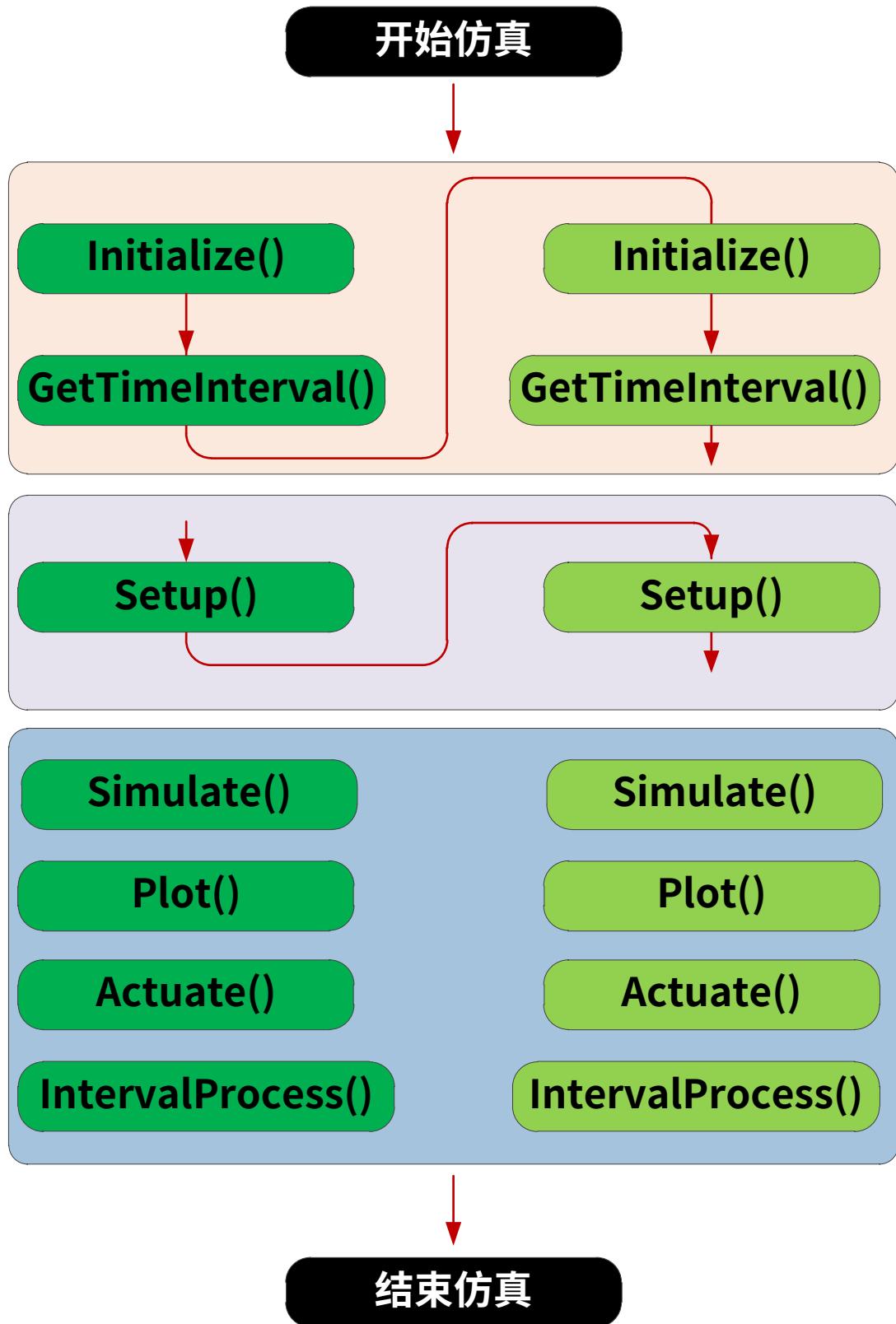
为最大限度简化仿真模型的开发，系统将所有 API 函数归类到 `IDSIMMODEL` 与 `ICOMPONENT` 结构体，前者是你在进行开发时需要实现的接口（即本章的重点），后者则是你可以进行操作（或使用的功能），具体的 API 函数说明见第 5 章，本章仅介绍系统调用仿真模型中 API 函数的流程及相应的功能，如下图所示（假设项目文件中存在两个具有仿真模型的元件），详细说明如下：

1、系统进入仿真状态后会依次尝试加载所有元件对应的仿真模型文件。如果仿真模型加载成功，首先会调用 `Initialize` 函数（其中第一个形参是代表当前元件的指针，仿真模型中所有可供使用的功能均来源于此指针，你应该声明一个本地变量并将其保存起来），紧接着，系统会调用 `GetTimeInterval` 函数获取仿真模型指定的时间间隔，以满足某些需要周期性更新状态的元件的需求，后续系统将启动一个线程（根据指定的时间间隔）循环调用 `IntervalProcess` 函数。

2、当系统调用了所有仿真模型的 `Initialize` 与 `GetTimeInterval` 函数后，接下来将依次调用所有仿真模型的 `Setup` 函数以提供第二次初始化的机会。

3、此时已经正式进入仿真状态，系统按照设置的时间（取决于系统的“选项”菜单下的“仿真”对话框）循环调用 `Simulate` 与 `Plot` 函数，前者主要用于解析元件的预置数据（你可以从中逐条取出预置数据进行处理）或从远程硬件模块采集的数据（如果已经与硬件模块连接，且元件模型对采集数据进行了处理，具体见第 4 章），后者则将图形绘制到屏幕上。如果需要按键或鼠标控制元件的状态，则可以实现 `Actuate` 函数，系统在按键或鼠标按下事件发生时将调用该函数（具体的编程开发可参考第 3 章的灯泡元件）。

4、系统在结束仿真时依次卸载所有元件对应的仿真模型。

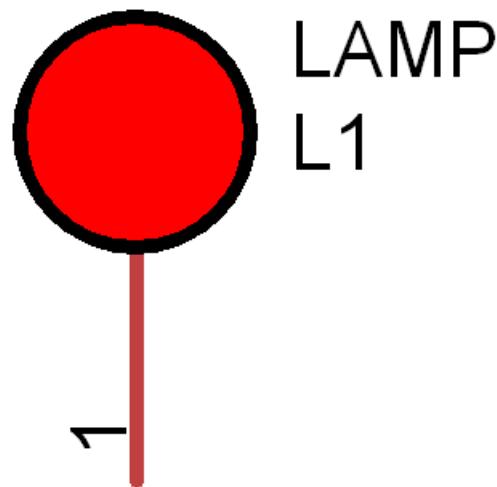


# 第3章 灯泡元件

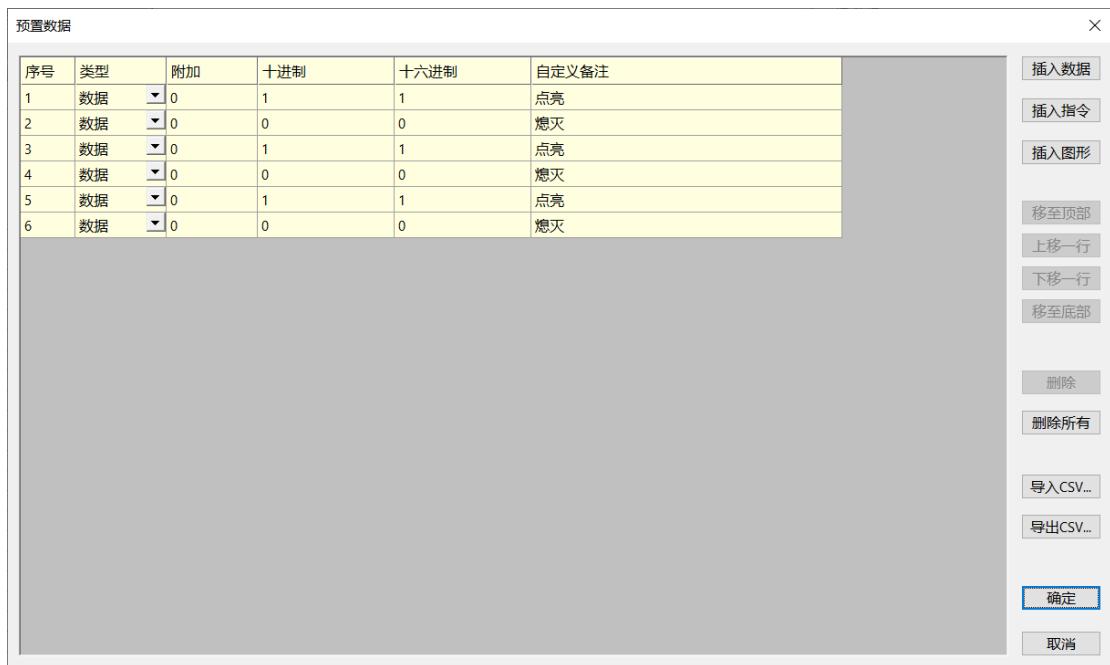
现在开始我们的第一个灯泡元件的制作，下图为可能会使用到的工具栏



我们需要创建的灯泡元件本身只有一个圆圈（**如果只是单机仿真，是否添加引脚并不重要，如果需要与远程硬件模块交互，则必须添加引脚且引脚编号必须唯一，而引脚名称则可选**），相应的状态如下图所示：



灯泡的行为也很简单，如果预置数据的最低位为 0 则熄灭，最低位为 1 则点亮，相应的预置数据如下图所示：

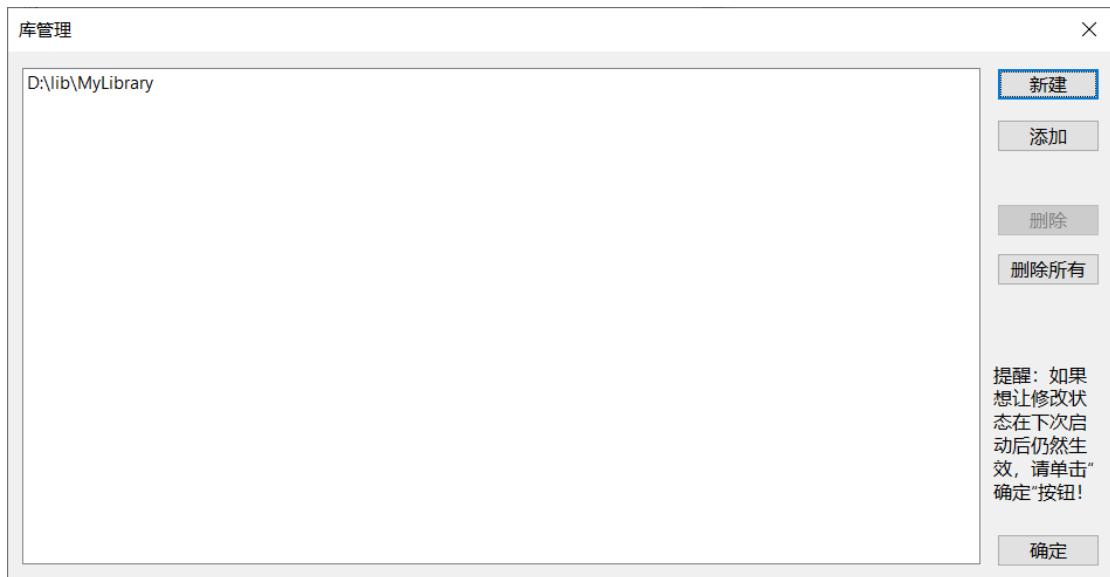


## 3.1 最简洁的开发流程

本节将使用最少的步骤创建一个勉强可用的灯泡，虽然缺陷不少，但至少可以让你快速了解仿真模型开发到底是怎么回事，赶紧开始吧~~

### 3.1.1 创建元件库（可选）

VisualCom 本身自带了几个元件库，如果你愿意，也可以新建一个元件库来保存自己的元件（**保存到系统自带元件库需要以“管理员身份”运行 VisualCom**），只需要单击工具栏上的  图标（或“库”菜单栏中的“库管理”项）即可弹出下图所示“库管理”对话框，从中新建一个元件库（此例为“MyLibrary.lib”）即可，此处不再赘述。

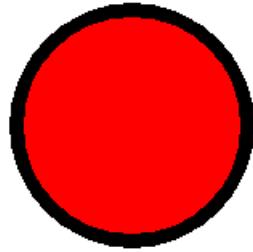


### 3.1.2 创建元件

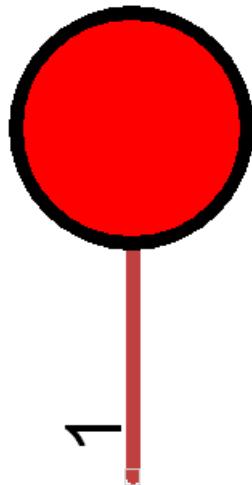
接下来开始进行元件原理图符号的绘制。单击工具栏上的“添加圆”按钮进入绘制圆圈状态，使用光标确定两个点即可完成圆圈的绘制，之后系统将自动弹出如下图所示“椭圆形选项”窗口，根据自己的需求更改相应的参数即可。



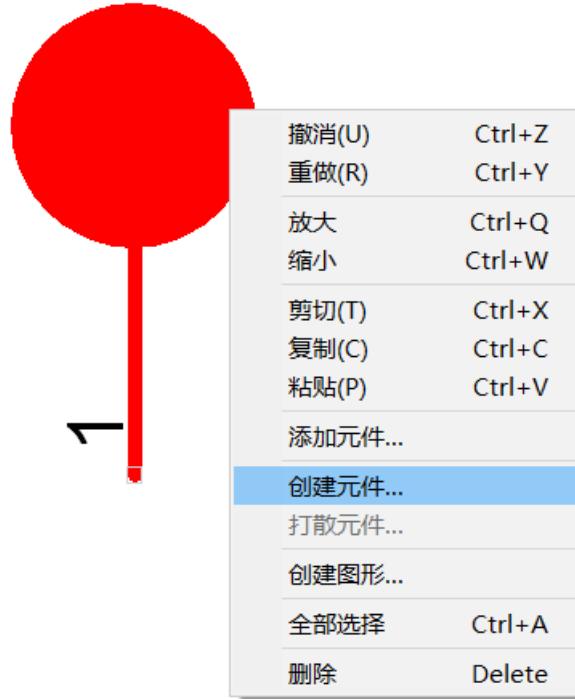
此处将线宽更改为“2”（默认为1），然后将“填充”项更改为“True”表示将该圆圈填充（默认不填充），并将颜色更改为红色即可，相应状态如下图所示。



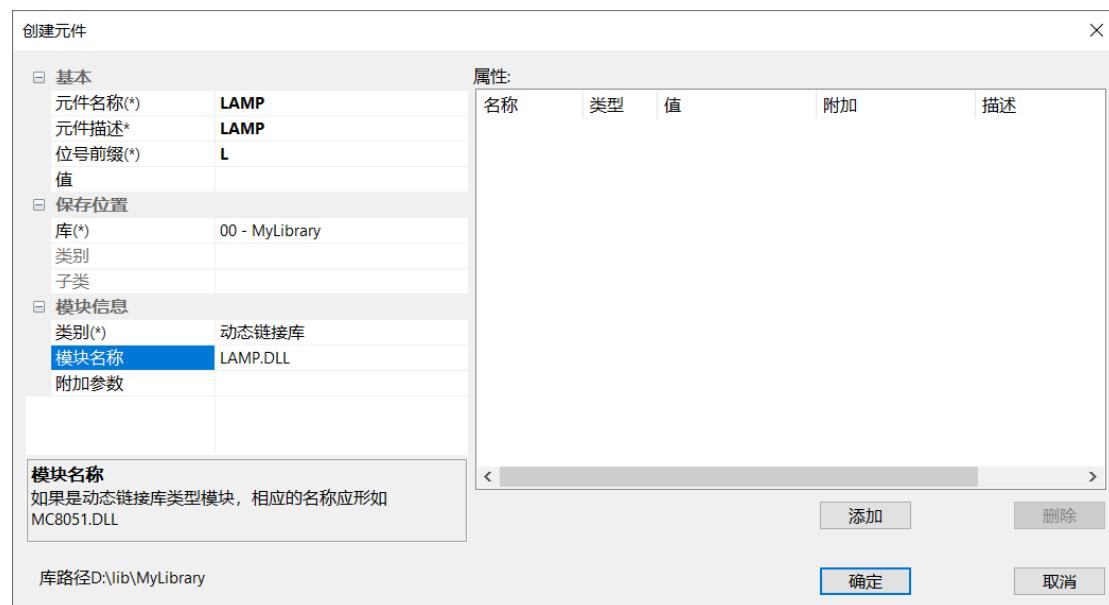
接下来添加一个引脚（可选）。单击工具栏上的“添加引脚”按钮，一个引脚将粘在光标上并随之移动，单击一次即可添加一个引脚，你也可以从弹出的“引脚 选项”对话框中更改参数。**引脚放置完毕后，右击即可退出添加引脚状态**，最后的状态如下图所示：



紧接着就是创建元件。使用光标拖动一个矩形框包围刚刚绘制的“圆”与“引脚”即可将其全部选中，右击后选择“创建元件”项，如下图所示。

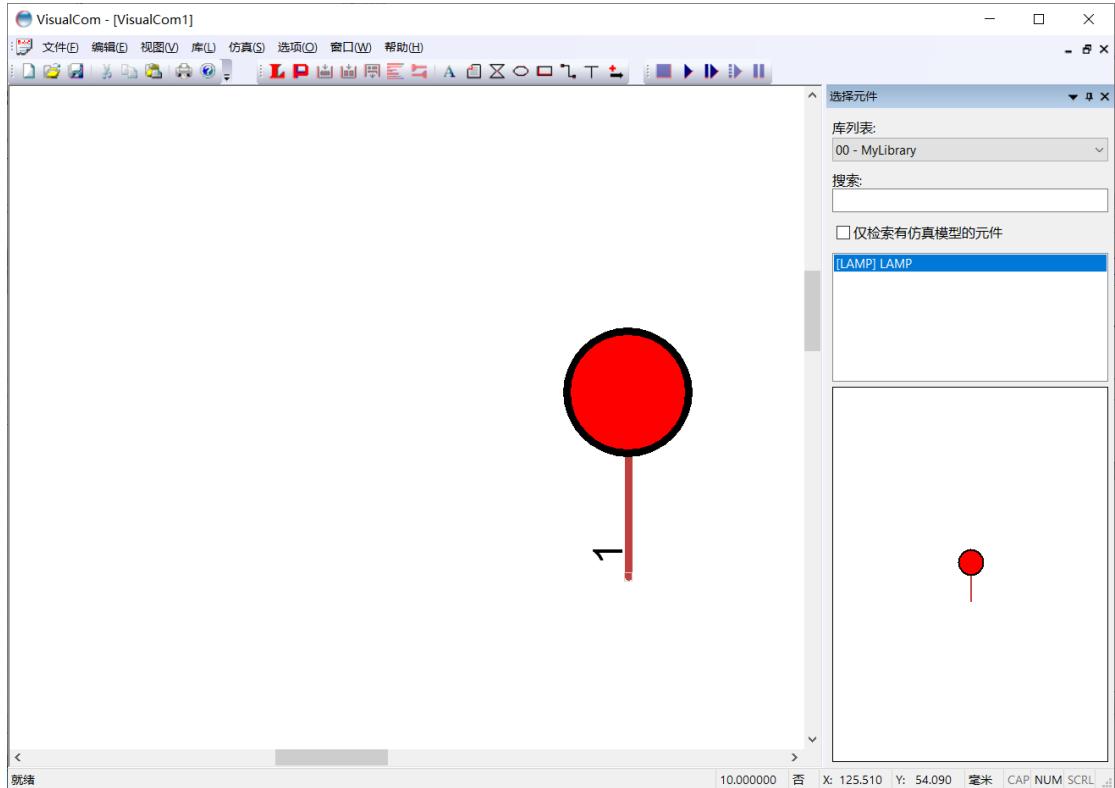


之后将弹出下图所示“创建元件”对话框，你可以自行输入“元件名称”、“元件描述”、“位号前缀”，此处选择的保存库为刚刚创建的元件库( MyLibrary )，“模块信息”类中的“模块名称”项非常重要，此处输入了“**LAMP.DLL**”，表示后续将创建文件名为“ LAMP.DLL ”的仿真模型文件，**请务必牢记！**



单击“确定”按钮，一个名为 LAMP 的元件就创建完毕了，你可以在“添加元件”对

话框中看到它，如下图所示：



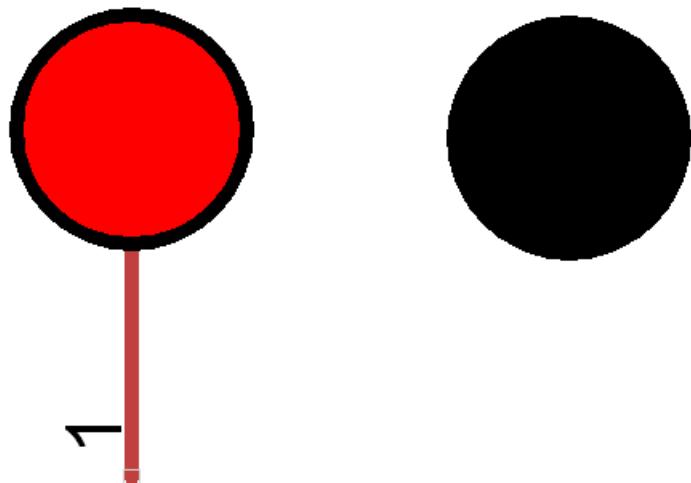
这里也提醒一下：**元件的大小应该绘制多大呢？**一般情况下以引脚大小为参考即可！这样就不会让多个创建的元件尺寸差别太大。但是也有人想，我就是想做一个大元件，这样仿真的效果好看！从实用的角度来讲没有太大的必要，因为主窗口的视图是可以缩放的，如果视图放大到极致还是觉得太小，你可以在选项窗口中修改“允许元件缩放”项，然后拖动四周的调整点即可（需要精确点击），而绘制到元件的图形是以元件为参考，所以显示的图形也会等比例缩放。

如果你觉得元件缩放后的尺寸更好，想将其放到库中，可以吗？不可以**直接**保存，但你可以将其打散。在元件已经缩放的状态下，圆形与矩形对象的参数在打散后可以保存（如果元件没有缩放，所有对象的参数在打散后都可以保存），然后再创建元件并保存即可

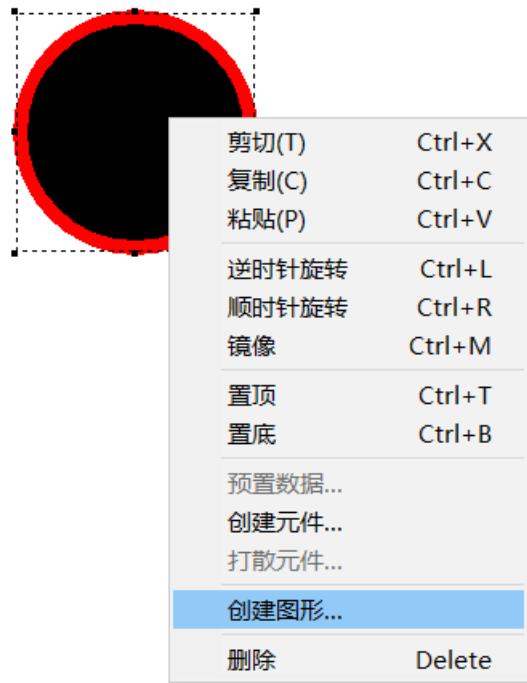
### 3.1.3 创建图形

灯泡元件的状态到底应该如何改变呢？我们的思路是这样：**灯泡默认是红色的，此时可**

以认为是点亮的，如果不需要点亮，只需要用一个黑色圆圈（与元件圆圈部分相同尺寸）贴上去覆盖原来的红色圆圈即可。所以我们需要一个黑色圆圈，只需要复制刚刚创建的红色圆圈，然后更改其填充颜色为黑色即可（实际上，颜色的设置并非必须，因为你在开发模型中随时更改，后述），相应状态如下图所示：



那么这个新创建的黑色圆圈怎么样才能贴上去呢？你必须得与刚刚创建的元件相**关联**！也就是为元件创建图形。选中刚刚复制的黑色圆圈，右击后选择“创建图形”项，如下图所示。



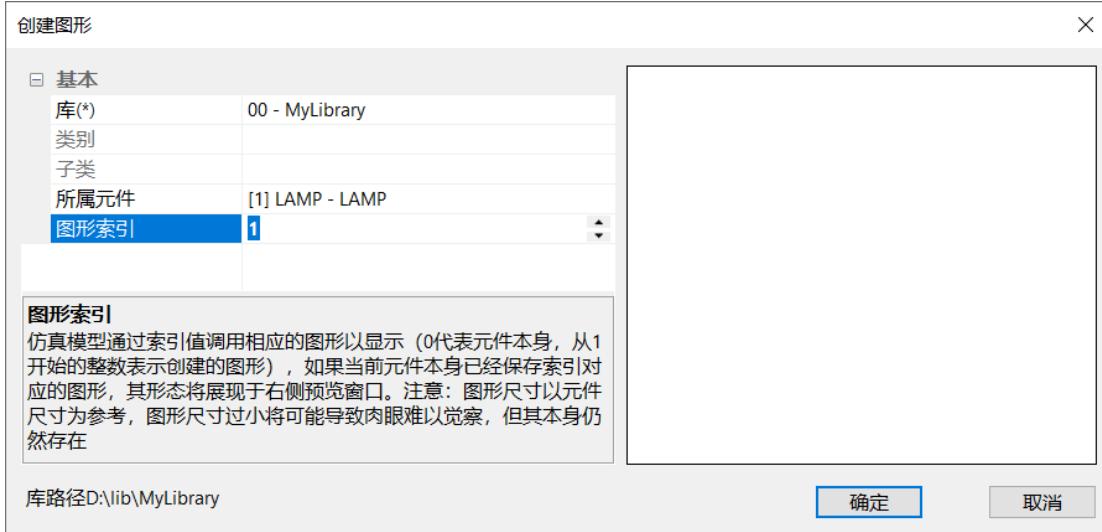
之后将弹出下图所示“创建图形”对话框，首先找到刚刚创建的元件( 此处为“LAMP” )。

图形索引是一个非常重要的参数，仿真模型是通过图形索引进行图形的绘制。例如，你将图形索引设置为 1，则在仿真模型中调用 DrawSymbol(1) 函数即可将该图形绘制出来，简单吧？！

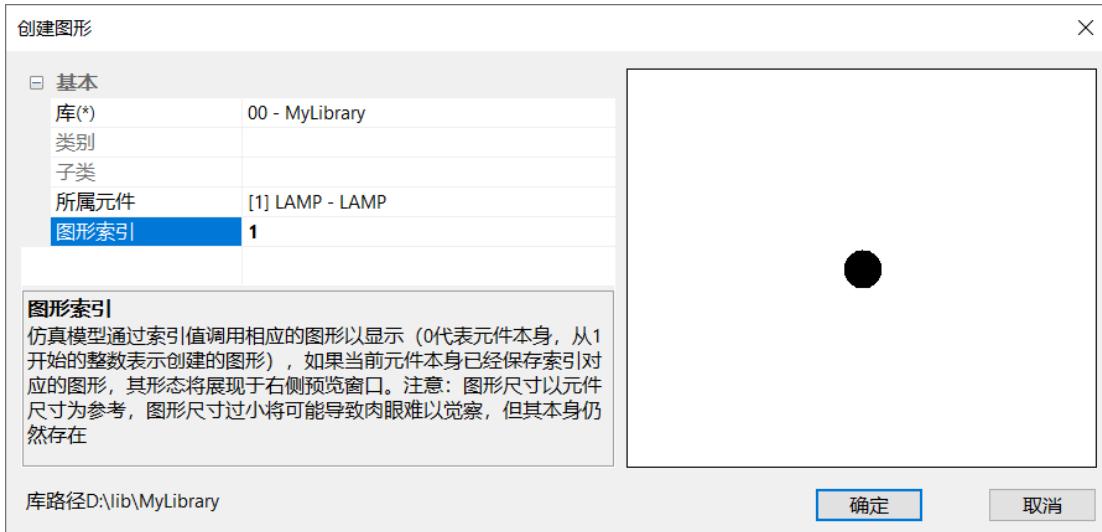


请特别注意：**图形索引 0 代表元件本身，所以你设置的图形索引必须大于 0**。我们设置

为 1，再单击“确定”按钮即可，相应地状态如下图所示。



很明显，上图右侧是空的，说明该图形索引位置还不存在图形，如果你在创建图形后再次进入并找到相同的索引，刚刚保存的图形就出现了，如下图所示。



好的，元件原理图符号这一块已经完成，我们从元件库中添加该元件到原理图（位号为 L1），只需要双击“选择元件”对话框中的“LAMP”项即可，相应的效果如下图所示（最右侧那个）



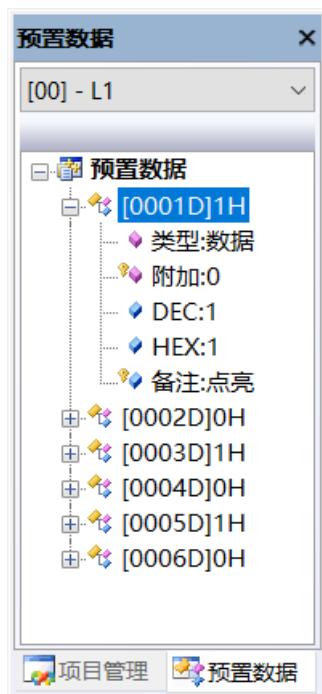
如果 L1 处于选中状态，“元件 选项”对话框会自动弹出，其中就包含了你刚刚创建元件时确定的一些参数（此处仅将参考编号更改为 L1，可选）



之后我们预置一些数据，选中 L1 后右击并选择“预置数据”项，在弹出的“预置数据”对话框中将本文最开始的预置数据输入进去，然后单击“确定”按钮即可。

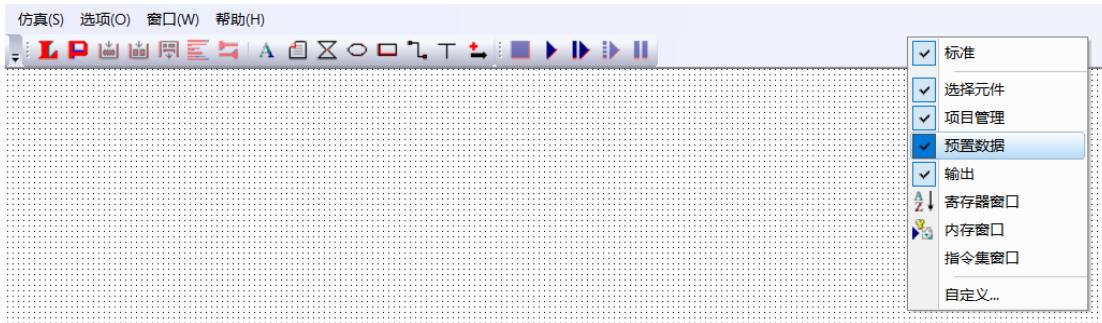


如果数据预置成功，“预置数据”窗口中将会出现相应的数据，如下图所示：中括号中  
的数字表示编号，字母表示类型（D 表示数据、C 表示指令、S 表示图形）

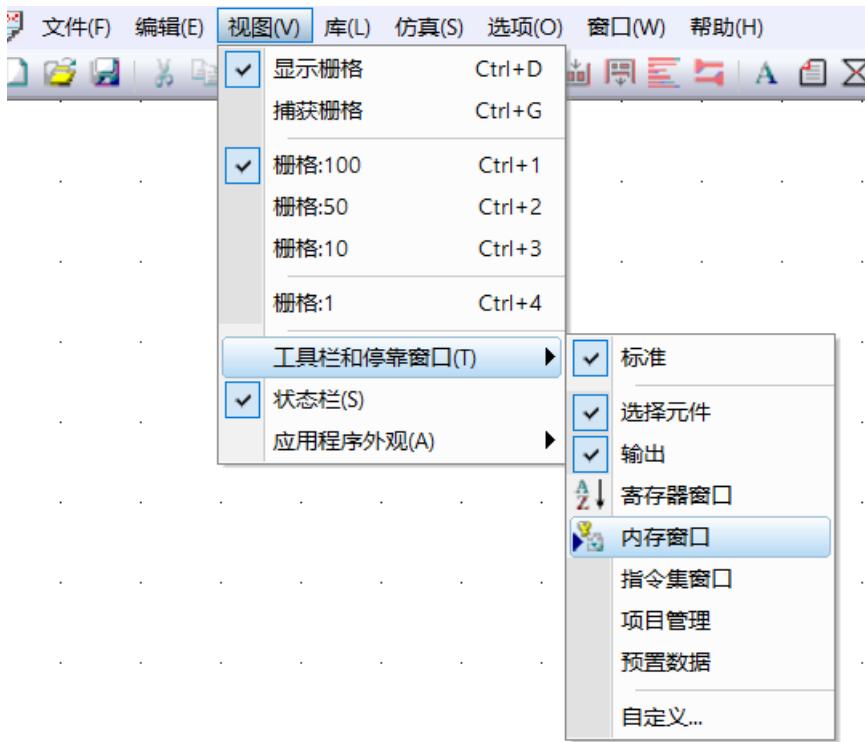


如果“预置数据”窗口处于隐藏状态，你可以右击工具栏，在弹出的快捷菜单中选中相

应的窗口即可，如下图所示，



当然，你也可以进入“视图”菜单栏，再进入“工具栏和依靠窗口”项中选中相应项即可，如下图所示。



接下来，单击工具栏上的“单步”仿真按钮 ，却发现 LAMP 的状态没有发生任何改变，因为你现在只是填入了仿真模型文件名“LAMP.DLL”，但是并未创建该文件，所以接下来你需要进行仿真模型文件的开发。

### 3.1.4 新建动态链接库工程

现在开始使用 Visual Studio ( 演示版本为 2022 ) 创建动态链接库。

( 1 ) 打开 Visual Studio, 如果之前并未创建任何工程, 左侧“打开最近使用的内容”栏是空的, 你可以单击右侧“创建新项目”项即可进入下一步创建新项目步骤。



如果 Visual Studio 已经处于项目打开状态, 你可以并选择【文件(F)】菜单→【新建(N)】→【项目(P)…】, 同样可创建新的项目。



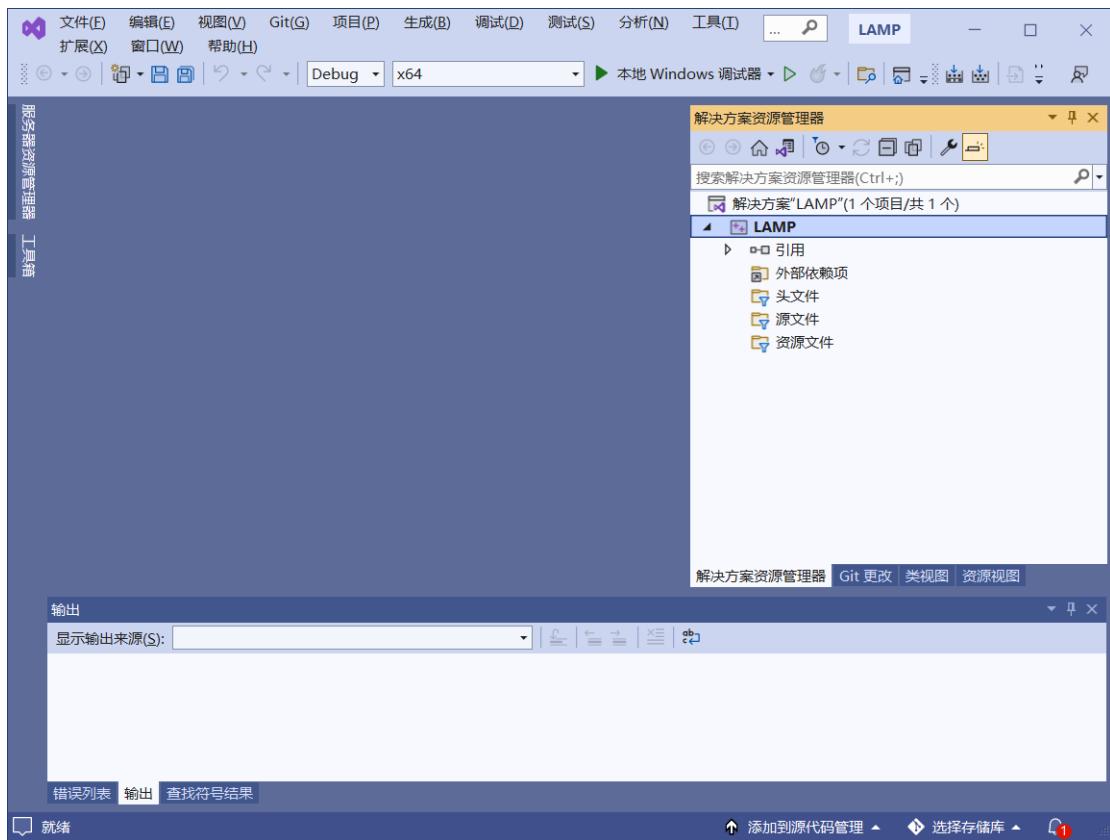
( 2 ) 执行上一步骤后, 你将进入下图所示“创建新项目”对话框, 从右侧列表内选中“空项目”即可(不是名为“动态链接库”的那一项)



(3) 单击“下一步”按钮后即可进入“配置新项目”对话框，从中输入项目的名称（此处为“LAMP”）及相应的路径（此处为“D:\dll\LAMP\_1\”）。值得一提的是，**项目名称并非必须为“LAMP”**，只不过默认情况下，编译后生成的 DLL 文件名与项目名称相同（此处为“LAMP.DLL”）。当然，即便项目名称并不是 LAMP，你也可以更改工程配置参数以指定生成的 DLL 文件名称（后述）

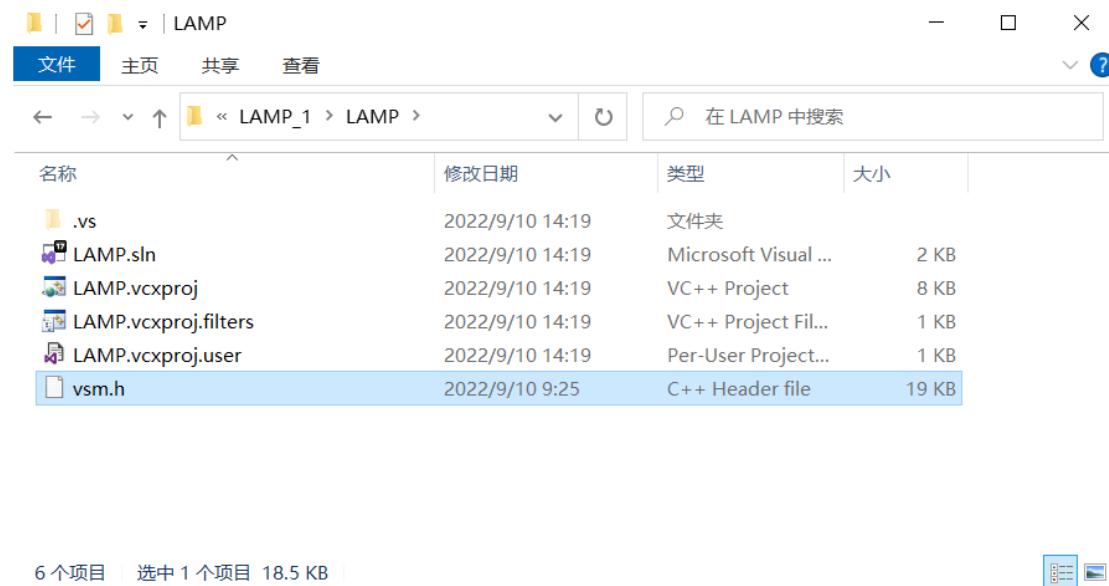


(4) 单击右下角的“创建”按钮即可完成空项目的创建工作，此时的状态如下图所示，正如你所见到的，里面什么都没有。



(5) 添加 vsm.h 头文件。为了开发可供系统调用的仿真模型，你需要在当前工程中添

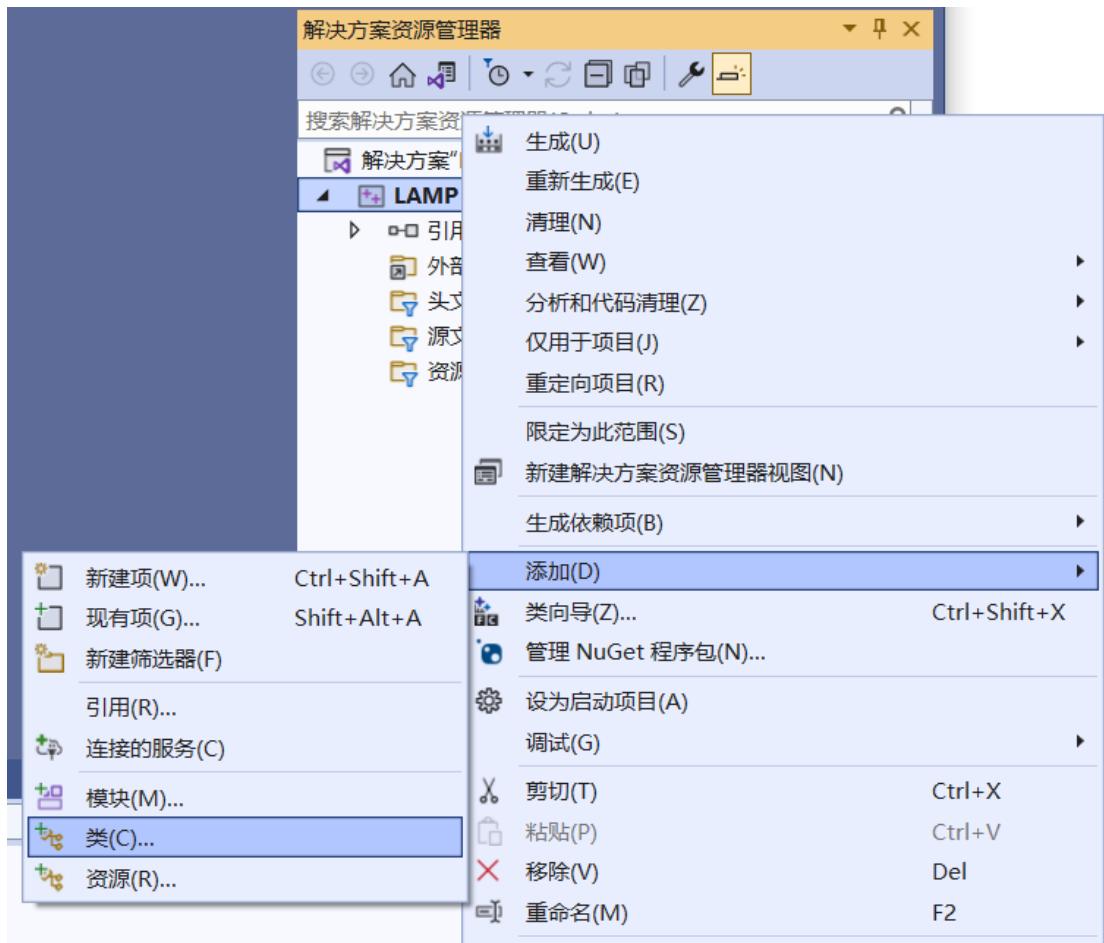
加 vsm.h。在 VisualCom 软件平台安装目录下的 vsm 文件夹下就存在该文件，将其复制到当前工程中，如下图所示：



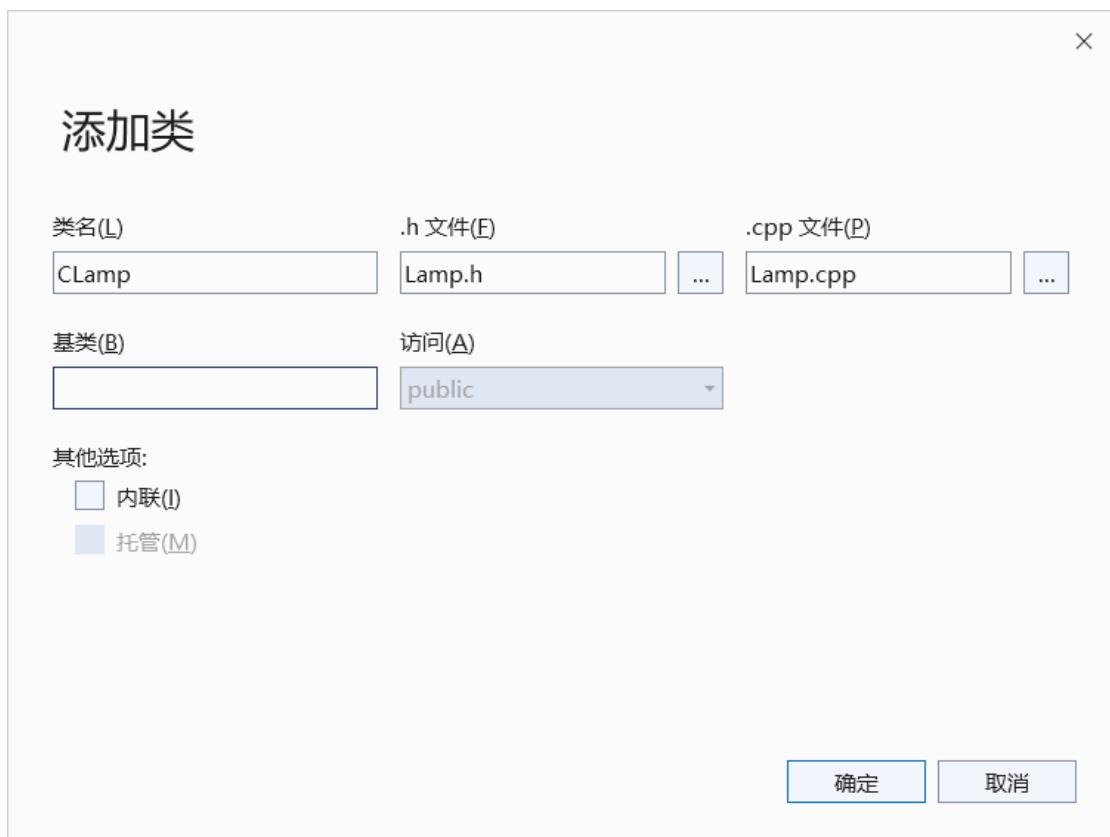
### 3.1.5 仿真模型开发

接下来正式开始仿真模型相关的编程，具体如下所示：

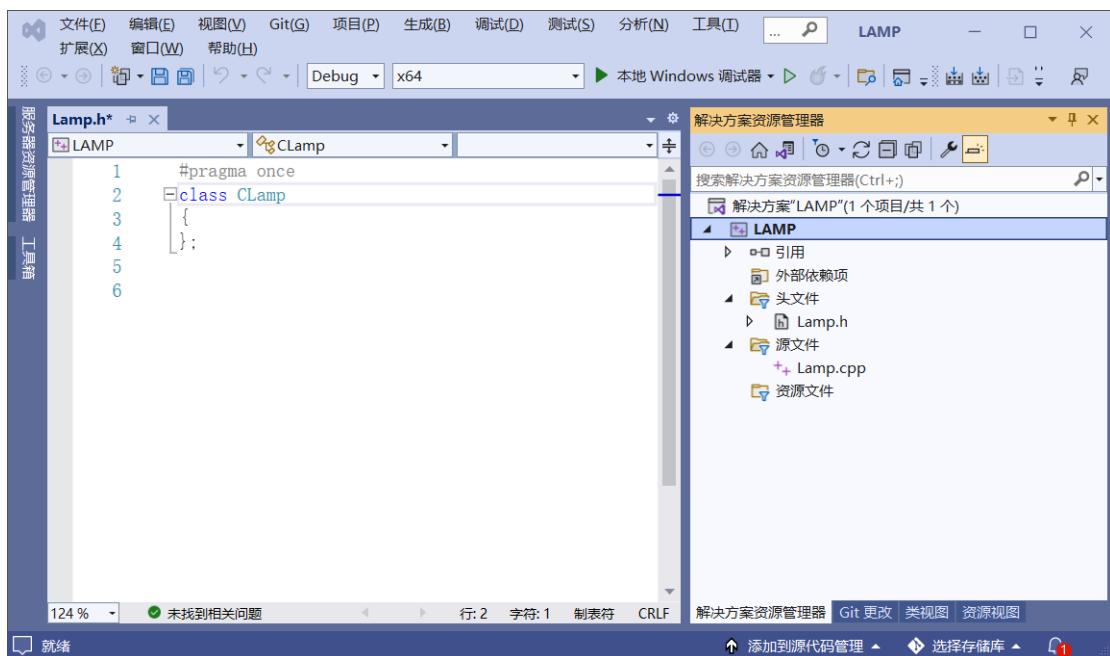
( 1 ) 创建新类。开发仿真模型其实就是新建一个类并实现虚函数的过程，首先我们创建一个类（此处为“CLamp”，名称并不重要）。同样在“解决方案资源管理器”窗口中右击工程名“LAMP”，并从中选择【添加(D)】→【类(C)…】（如果选择【添加(D)】→【现有项(G)…】，则可以添加已经存在的文件），如下图所示。



之后将弹出“添加类”对话框，从中设置类名（此处为“CLamp”）及相应的.h(.cpp)文件名即可。



创建类后的效果如下图所示，工程中添加了 Lamp.h 头文件及 Lamp.cpp 源文件（也可以通过添加现有项将 vsm.h 头文件加入进到当前工程）。



(2) 接下来看看修改后的头文件 Lamp.h 与源文件 Lamp.cpp，相应的内容如下图所示：

## 头文件 Lamp.h

```
1 #pragma once
2 #include "vsm.h"
3
4 class CLamp : public CDSIMMODEL
5 {
6     ICOMPONENT* component; //代表元件
7     BOOL m_LampStatus; //代表灯泡的状态, FALSE为灭, TRUE为亮
8 public:
9     CLamp(); //构造函数
10    ~CLamp(); //析构函数
11 public: //以下为仿真模型开发需要实现的接口
12     CDSIMMODEL* GetSimModel(TCHAR* device);
13     ICOMPONENT* GetComponentPtr();
14     LONG GetTimeInterval(INT dat);
15     BOOL IntervalProcess(RUNMODES mode);
16     VOID Initialize(ICOMPONENT* cpt, DSIMMODES smode);
17     VOID Setup(SIMDATA* sdat);
18     BOOL Simulate(ABSTIME time, RUNMODES mode);
19     VOID CallBack(ABSTIME time, EVENTID eventid);
20     BOOL Indicate(REALTIME time, ACTIVECDATA* data);
21     VOID Animate(INT element, ACTIVECDATA* data);
22     BOOL Plot(ACTIVESTATE state);
23     BOOL Actuate(WORD key, DPOINT p, UINT flags);
24 };
```

## 源文件 Lamp.cpp

```

1   #include "Lamp.h"
2
3   extern "C" CDSIMMODEL __declspec(dllexport) * CreatedSimModel()
4   {
5       return new CLamp;
6   }
7
8   extern "C" VOID __declspec(dllexport) DeletedSimModel(CDSIMMODEL * model)
9   {
10      delete (CLamp*)model;
11  }
12
13  CLamp::CLamp()
14  {
15      m_LampStatus = FALSE;
16  }
17
18  CLamp::~CLamp()
19  {
20  }
21
22  CDSIMMODEL* CLamp::GetSimModel(TCHAR* device)
23  {
24      return this;
25  }
26
27  ICOMPONENT* CLamp::GetComponentPtr()
28  {
29      return component;
30  }
31
32  LONG CLamp::GetTimeInterval(INT dat)
33  {
34      return -1;
35  }
36
37  BOOL CLamp::IntervalProcess(RUNMODES mode)
38  {
39      return FALSE;
40  }
41
42  VOID CLamp::Initialize(ICOMPONENT* cpt, DSIMMODES smode)
43  {
44      component = cpt;                                //初始化本地元件指针
45  }
46
47  VOID CLamp::Setup(SIMDATA* sdat)
48  {
49  }
50
51
52  BOOL CLamp::Simulate(ABSTIME time, RUNMODES mode)
53  {
54      PRODATA pro_data_tmp;
55      component->GetProData(pro_data_tmp);           //获取预置数据
56
57      if (pro_data_tmp.data & 0x1)                  //判断预置数据最低位
58      {
59          m_LampStatus = TRUE;                      //设置灯泡为点亮状态
60      }
61      else

```

```

62     {
63         m_LampStatus = FALSE;                                //设置灯泡为熄灭状态
64     }
65 }
66 return FALSE;
67 }

68
69 VOID CLamp::CallBack(ABSTIME time, EVENTID eventid)
70 {
71 }

72
73
74 BOOL CLamp::Indicate(REALTIME time, ACTIVEDATA* data)
75 {
76     return FALSE;
77 }

78
79 VOID CLamp::Animate(INT element, ACTIVEDATA* data)
80 {
81     component->BeginCache();                           //开始缓存
82
83     DPOINT sym1_offset = component->GetSymbolOffset(1); //获取图形1的偏移位置
84
85     if (!m_LampStatus)                                 //如果为熄灭状态，就绘制图形1
86     {
87         component->DrawSymbol(1, NULL, NULL, NULL, sym1_offset.x, sym1_offset.y);
88     }
89
90     component->EndCache();                           //结束缓存
91 }

92
93 BOOL CLamp::Plot(ACTIVESTATE state)
94 {
95     Animate(0, NULL);                                //调用Animate绘制图形
96
97     return TRUE;
98 }

99
100 BOOL CLamp::Actuate(WORD key, DPOINT p, UINT flags)
101 {
102     return FALSE;
103 }

```

虽然源文件中的代码行数超过 100，但实际上，属于自己添加的有效代码行数量还不到 10（其它保持默认即可），我们从上到下挑重点简单分析一下：

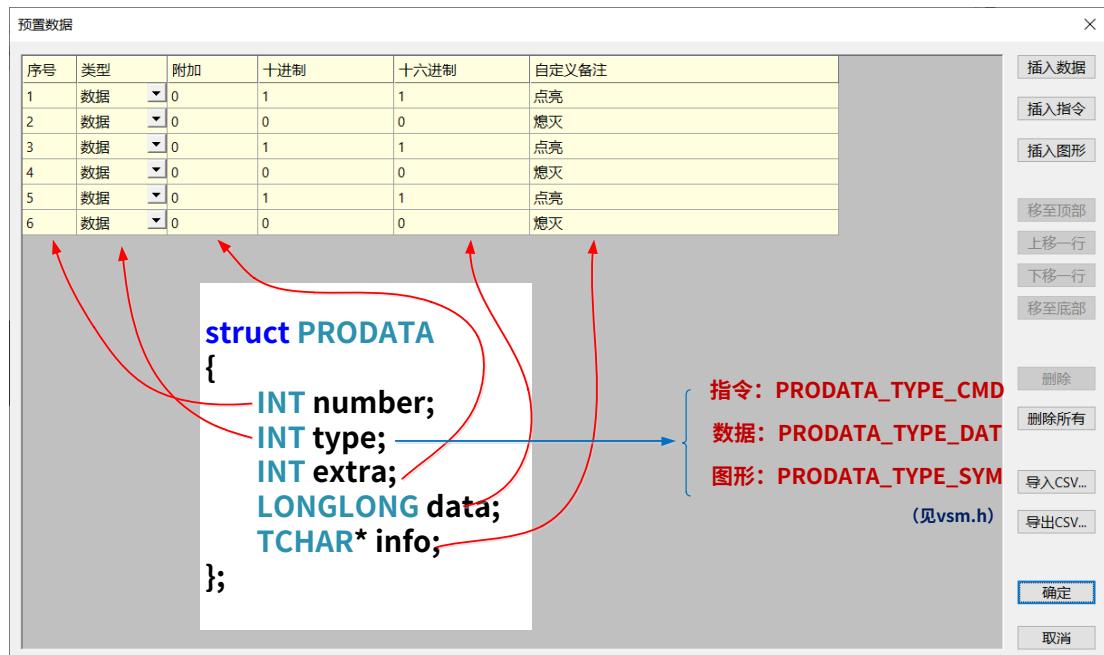
其一，开头以 `extern "C"` 修饰的语句用来导出当前类到 DLL，你只需要照抄并根据当前的类名更改新建（`new`）与删除（`delete`）的名称即可（此处为“`CLamp`”），系统在进入仿真状态后会调用 `CreateSimModel` 函数返回当前类的一个实例，而当结束仿真状态时，则会调用 `DeleteSimModel` 函数删除已经创建的实例（以释放相应的资源）；

其二，在构造函数中，我们将 `m_LampStatus` 初始化为 `FALSE`，表示灯泡默认处于熄

灭状态；

其三，GetSimMode, GetComponentPtr, GetTimeInteval, IntervalProcess, Initialize, Setup 函数都保持默认即可，其中，系统会调用 Initialize 函数并传入一个代表元件的指针，你只需要将其赋给本地声明的 component，后续就可以使用 component 指针进行元件的各种操作了。

其四，Simulate 函数用来处理预置数据（或从远程硬件模块中采集的数据，具体见第 4 章），在单步运行状态下，你需要使用 GetProData 函数依次获取预置数据，对其进行判断后再进行相应的处理。GetProData 函数需要你传递一个 PRODATA 类型变量，其结构（见 vsm.h）与预置数据对话框的对应关系如下图所示。灯泡元件的行为非常简单，所以在获取预置数据后，只需要在 if 语句中判断数据的最低位，再根据结果改变 m\_LampStatus 变量即可（当然，你也可以自行约定预置数据的用途，只需要在元件模型中进行相应处理即可）。



其四，CallBack, Indicate 函数暂时不用理会。

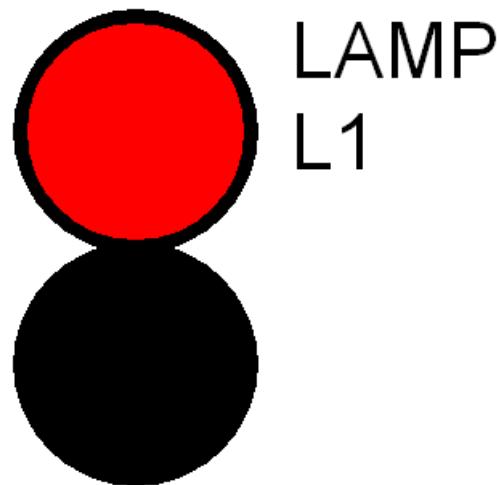
其五，Animate 中主要进行绘图相关的操作（**只是在缓存中绘图，并不是在屏幕中绘**

图)。在 BeginCache()与 EndCache( )之间,首先定义了一个 sym1\_offset, 它包含 double 类型的坐标 x, y, 其结构如下图所示:

```
struct DPOINT
{
    DOUBLE x;
    DOUBLE y;

};
```

本例首先通过调用 GetSymbolOffset(1)来获取图形 1 的偏移位置,为什么要这么做呢?因为绘制的图形是以元件的右下角(含引脚)为参考(即原点),坐标值往左或上为负,往右或下为正值。如果你直接调用 DrawSymbol(1)函数(DrawSymbol 存在多个重载函数,仅传入一个图形索引值则表示偏移位置为 0),相应的图形会出现在元件的右下角,如下图所示:(由于本例比较简单,所以看上去像在正下方)

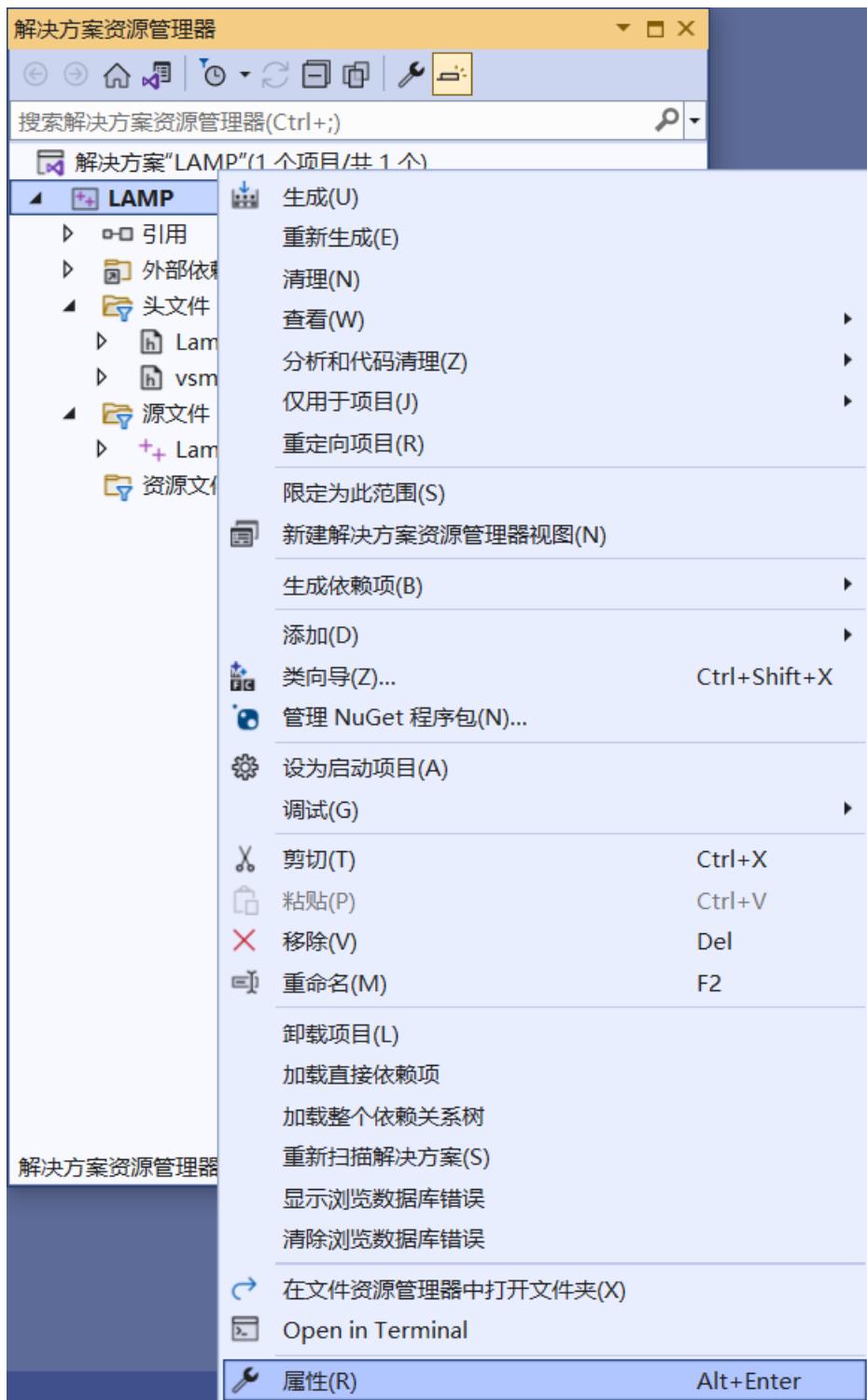


但是很多开发者可能更习惯以左上角为参考,为方便这些用户,你可以先调用

GetSymbolOffset 函数获取需要绘制的图形的偏移值, 再将该偏移值随 DrawSymbol 绘制, 这样图形肯定就会出现在左上角 ( 后续以该坐标为参考进行后续绘图即可 )。本例中左侧或上侧均不存在引脚, 所以 DrawSymbol 之后恰好就会覆盖原来的红色圆圈, 也就达到了灯泡熄灭的效果。

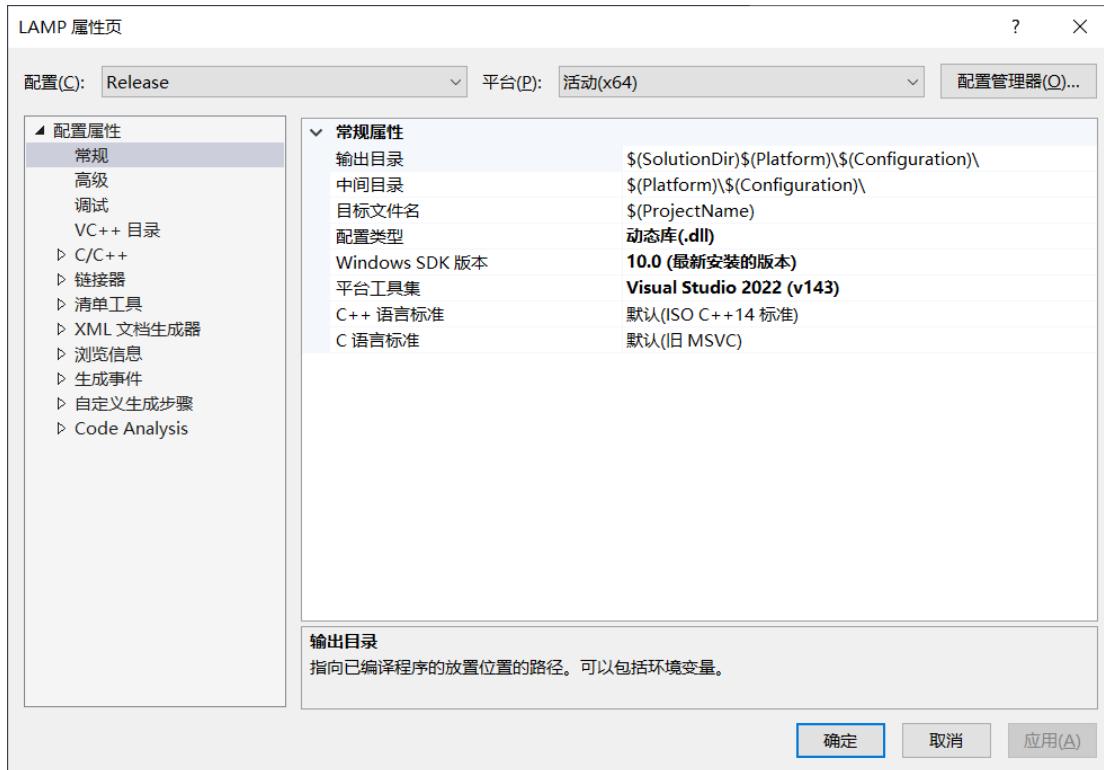
其六, 系统根据设置的刷新时间循环不停地调用 Plot 函数, 如果你需要刷新元件的状态, 需要将代码写在这里。其中调用了刚刚介绍的 Animate 函数, 一定要记得返回 TRUE, 这样视图才会实时刷新。

其七, Actuate 函数用来处理鼠标或按键事件, 暂时不用理会  
( 3 )编译工程。代码编写完毕后就应该进行编程阶段, 但在此之前**必须**配置一些参数,  
**否则后续编译时可能会出错或生成的 DLL 文件不可用**。如下图所示, 右击“解决方案资源管理器”窗口中的工程名“LAMP”, 在弹出的快捷菜单中选择其中的“属性”项 ( 最下面那一项 )。

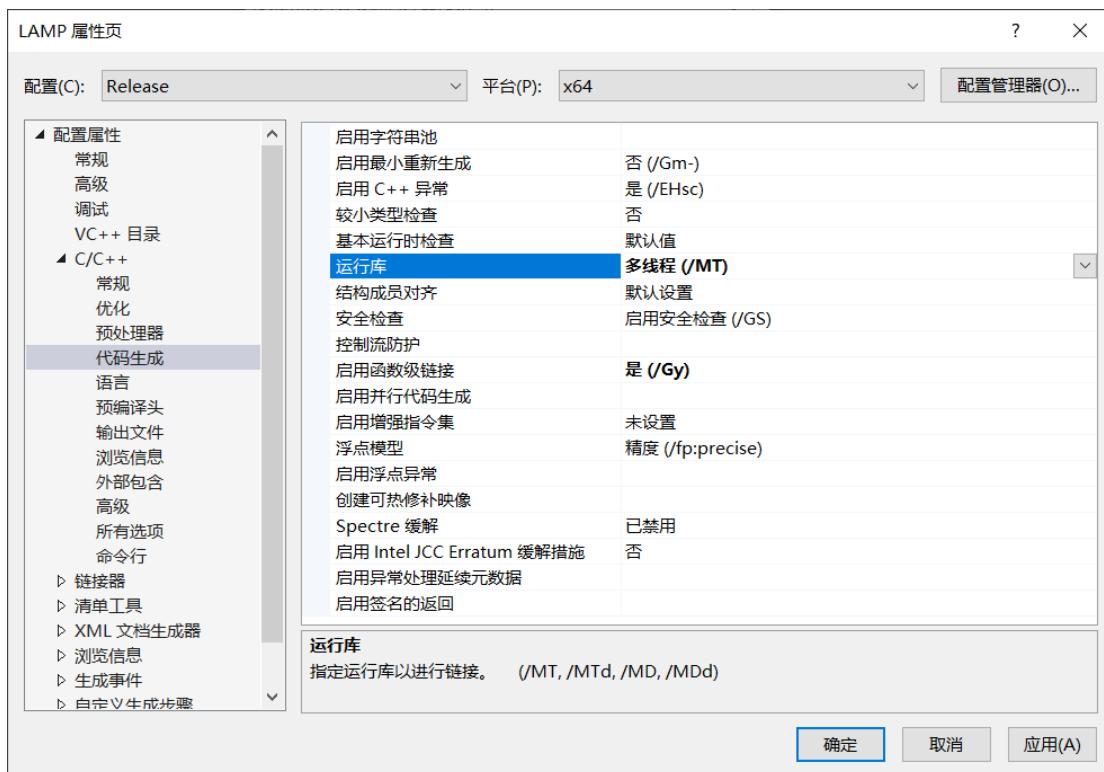


之后将弹出下图所示“LAMP 属性页”对话框，首先你需要关注“常规”类别中的两个项目。其一，“配置类型”项默认为“应用程序(.exe)”，你应该将其修改为“**动态库(.dll)**”，因为现在我们正在进行动态链接库的开发，而动态链接库是没有 main 入口的，否则编译后就会出错，也就不会生成 DLL 文件。其二，“目标文件名”项默认为“\$(ProjectName)”，

也就代表以工程名称作为目标文件。对于该项目，也就意味着编译后将生成名为“LAMP.DLL”的文件。如果你需要的元件模型文件名称与工程名称不相同，在此更改即可（不需要输入“.DLL”）。本例的工程名与需要的目标名相同，所以不需要更改（**注意：“配置”列表中是“Release”，“平台”列表则与你选择的 VisualCom 软件安装程序对应，编译时需要选择相对应的平台，此处为 x64，后述**）。

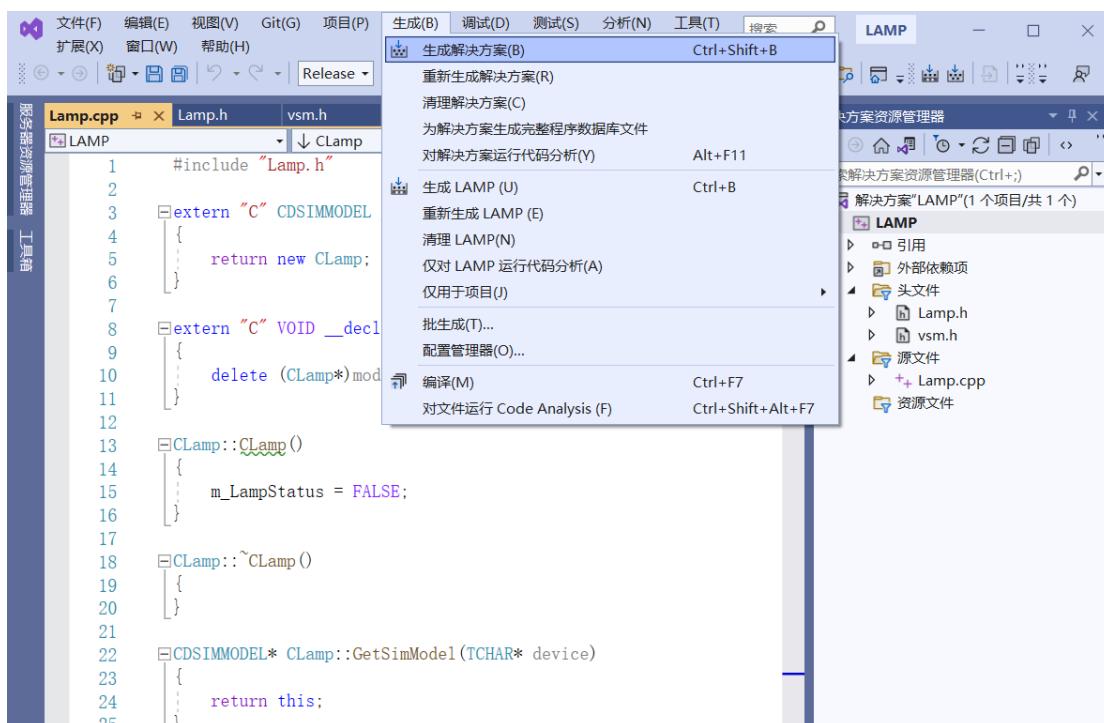


然后再切换到“C/C++”类别中的“代码生成”项，确保“运行库”项为“**多线程( /MT )**”，如下图所示，这也是编译 VisualCom 软件平台时的选项，两者需要保持一致，否则即使 DLL 文件已经生成，也将会无法正常加载仿真模型。



( 4 )完成前述参数配置后即可开始进行工程编译。选择【生成】→【生成解决方案( R )】

即可开始工程编译的状态，如下图所示。



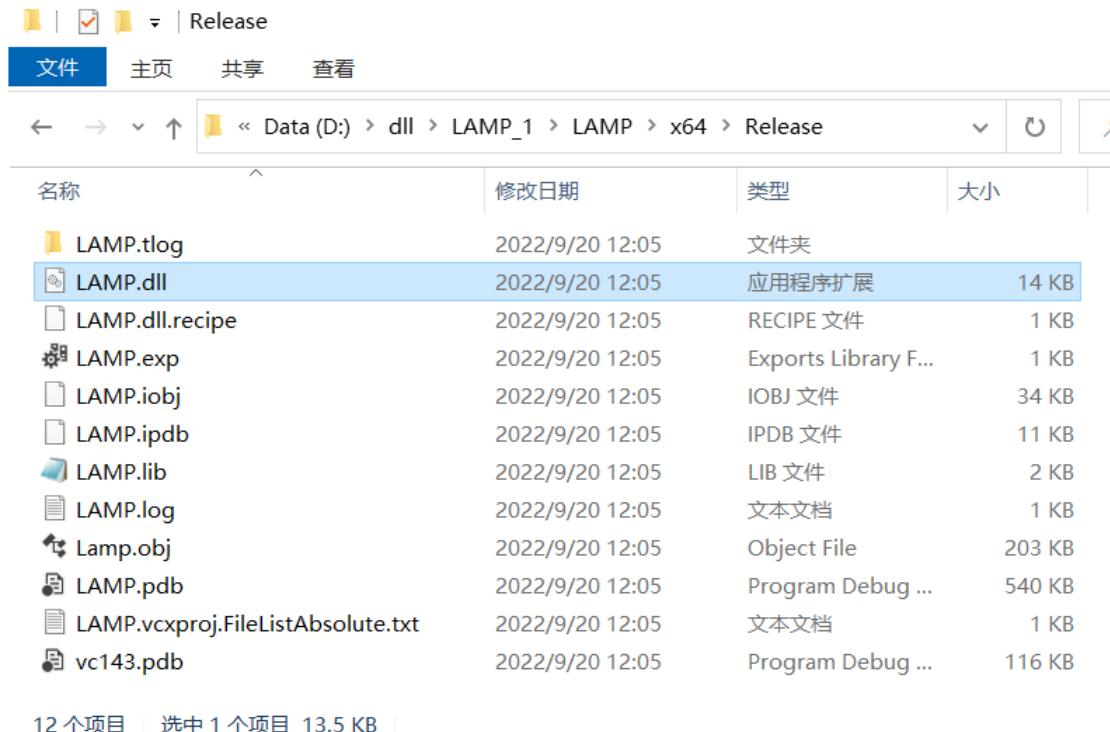
需要特别注意的是，**编译时也应该选择“Release”，平台为“x64”**，如下图所示（对于 VisualCom x64 版本，选择“x86”生成的 DLL 文件将无法正常加载）



(5) 如果一切顺利，编译过程将成功结束，相应地状态如下图所示

```
输出
显示输出来源(S): 生成
已启动生成...
1>----- 已启动生成: 项目: LAMP, 配置: Release x64 -----
1>Lamp.cpp
1> 正在创建库 D:\dll\LAMP_1\LAMP\x64\Release\LAMP.lib 和对象 D:\dll\LAMP_1\LAMP\x64\Release\LAMP.exp
1>正在生成代码
1>Previous IPDB not found, fall back to full compilation.
1>All 19 functions were compiled because no usable IPDB/IOBJ from previous compilation was found.
1>已完成代码的生成
1>LAMP.vcxproj -> D:\dll\LAMP_1\LAMP\x64\Release\LAMP.dll
===== 生成: 成功 1 个, 失败 0 个, 最新 0 个, 跳过 0 个 ======
```

同时在目录(此处为“D:\dll\LAMP\_1\LAMP\x64\Release”)下将出现一个 Lamp.DLL 文件，至此仿真模型的开发流程已经结束，如下图所示：



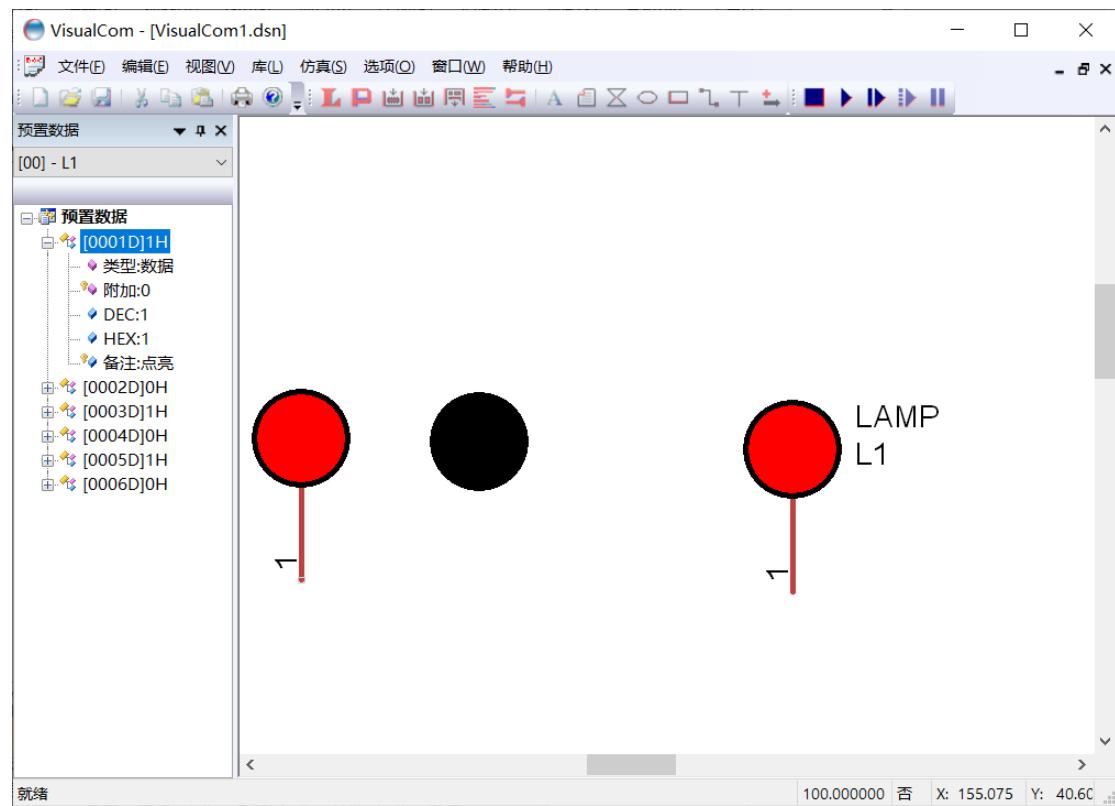
12 个项目 选中 1 个项目 13.5 KB

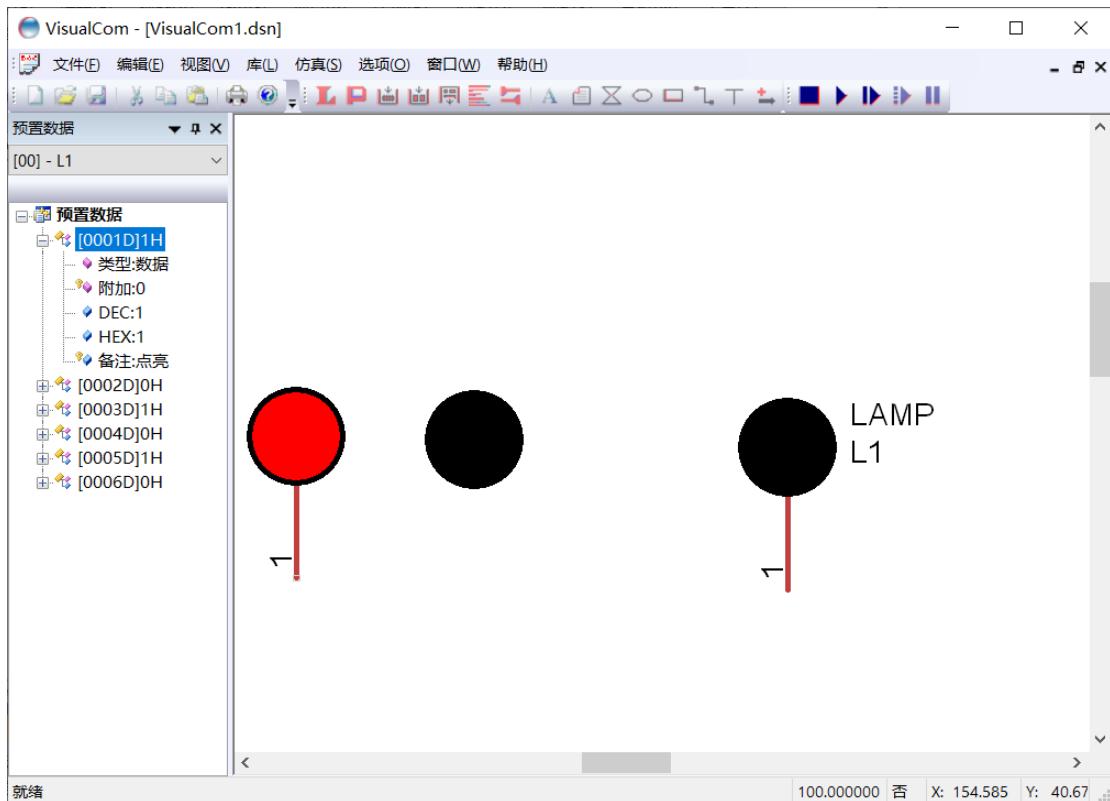
### 3.1.5 将 DLL 文件拷贝到 model 文件夹中

有了 Lamp.DLL 文件后，你得让 VisualCom 软件平台能够根据元件设置的仿真文件名找到它，**为此你需要将 DLL 文件拷贝到 VisualCom 软件平台安装目录下的 model 文件夹中**（里面已经存在一些 DLL 文件，你不会错过的），VisualCom 会在该目录下查找需要的仿真模型文件，之后就可以正式仿真了

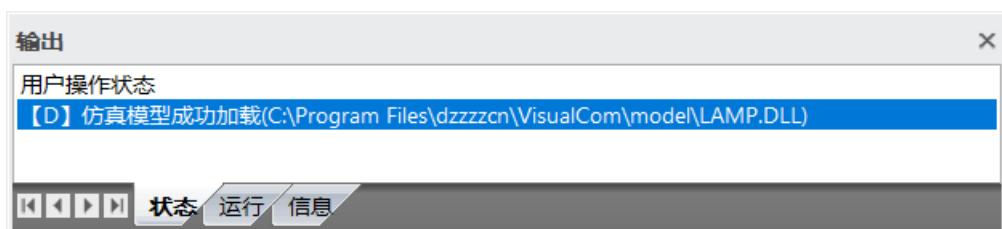
### 3.1.6 运行仿真

现在开始运行单步仿真，灯泡点亮与熄灭的状态分别如下两图所示。当然，现在的灯泡颜色确实有些丑，后续我们再来优化吧！





值得一提的是，在进入仿真状态后，如果元件的仿真模型被正确加载，“输出”窗口的“状态”标签页内将会显示相应的成功加载信息，这可以为模型开发是否正确提供一定的参考，类似如下图所示：



如果你想让灯泡自己按照一定的时间间隔依序执行（即闪烁），可以选择 VisualCom 软件平台的“选项”菜单下的“仿真”选项，在弹出的“仿真参数”对话框中将“暂停时间”项设置为 1000（即 1 分钟），之后再全速运行仿真（不是单步仿真），灯泡就会每隔 1 分钟转换一个状态，具体转换的次数取决于预置数据的多少。当然，后续还会开发另一种闪烁灯泡（预置数据最低位为 1 时闪烁，为 0 时不闪烁），这需要使用到线程调用，这种闪烁不取决于预置数据的多少，而取决于控制的状态。



## 3.2 优化灯泡仿真模型

虽然刚刚创建的灯泡已经可以使用了，但是实际使用起来还是存在一些小问题。其一，在**单步**运行仿真过程中，每执行一步，左侧“预置数据”窗口中的数据也将随变化（高亮项会往下移），但是很明显，前面仿真时却总是表现得无动于衷，这肯定是不正确的。其二，第一个预置数据为 1，所以执行第一步时，灯泡的状态应该不会变化，但是如果刷新时间设置得较长（例如 1 秒），你仍然可以看到灯泡先变成黑色，然后才转为红色，这也是不正常的，至少咱们得找到问题所在，不是吗？其三，Plot 函数中一直在调用 Animate（即便当前并未执行预置数据解析工作），在本例中可能没什么问题，但是在需要大量资源执行 Animate 函数的场合中，这是极大的资源浪费。

本节就来尝试解决这三个问题，为保留之前的代码状态，我们直接复制一份之前创建的工程，然后将文件夹修改为 LAMP\_2，而其中的头文件 Lamp.h 与源文件 Lamp.cpp 分别如下图所示：（仅显示修改处的代码）

### 头文件 Lamp.h

```
1 #pragma once
2 #include "vsm.h"
3
4 class CLamp : public CDSIMMODEL
5 {
6     ICOMPONENT* component; //代表元件
7     BOOL m_LampStatus; //代表灯泡的状态，FALSE为灭，TRUE为亮
8     BOOL m_RefreshFlag; //是否刷新视图，FALSE为否，TRUE为是
9 public:
10    CLamp(); //构造函数
11    ~CLamp(); //析构函数
12 public: //以下为仿真模型开发需要实现的接口
13    CDSIMMODEL* GetSimModel(TCHAR* device);
14    ICOMPONENT* GetComponentPtr();
15    LONG GetTimeInterval(INT dat);
16    BOOL IntervalProcess(RUNMODES mode);
17    VOID Initialize(ICOMPONENT* cpt, DSIMMODES smode);
18    VOID Setup(SIMDATA* sdat);
19    BOOL Simulate(ABSTIME time, RUNMODES mode);
20    VOID CallBack(ABSTIME time, EVENTID eventid);
21    BOOL Indicate(REALTIME time, ACTIVEDATA* data);
22    VOID Animate(INT element, ACTIVEDATA* data);
23    BOOL Plot(ACTIVESTATE state);
24    BOOL Actuate(WORD key, DPOINT p, UINT flags);
25};
```

### 源文件 Lamp.cpp:

```
13 CLamp::CLamp()
14 {
15     m_LampStatus = FALSE;
16     m_RefreshFlag = FALSE;
17 }
```

```

53     BOOL CLamp::Simulate(ABSTIME time, RUNMODES mode)
54     {
55         PRODATA pro_data_tmp;
56         if (component->GetProData(pro_data_tmp)) //获取预置数据
57         {
58             if (pro_data_tmp.data & 0x1) //判断预置数据最低位
59             {
60                 m_LampStatus = TRUE; //设置灯泡为点亮状态
61             }
62             else
63             {
64                 m_LampStatus = FALSE; //设置灯泡为熄灭状态
65             }
66             m_RefreshFlag = TRUE;
67             return TRUE;
68         }
69     }
70 }

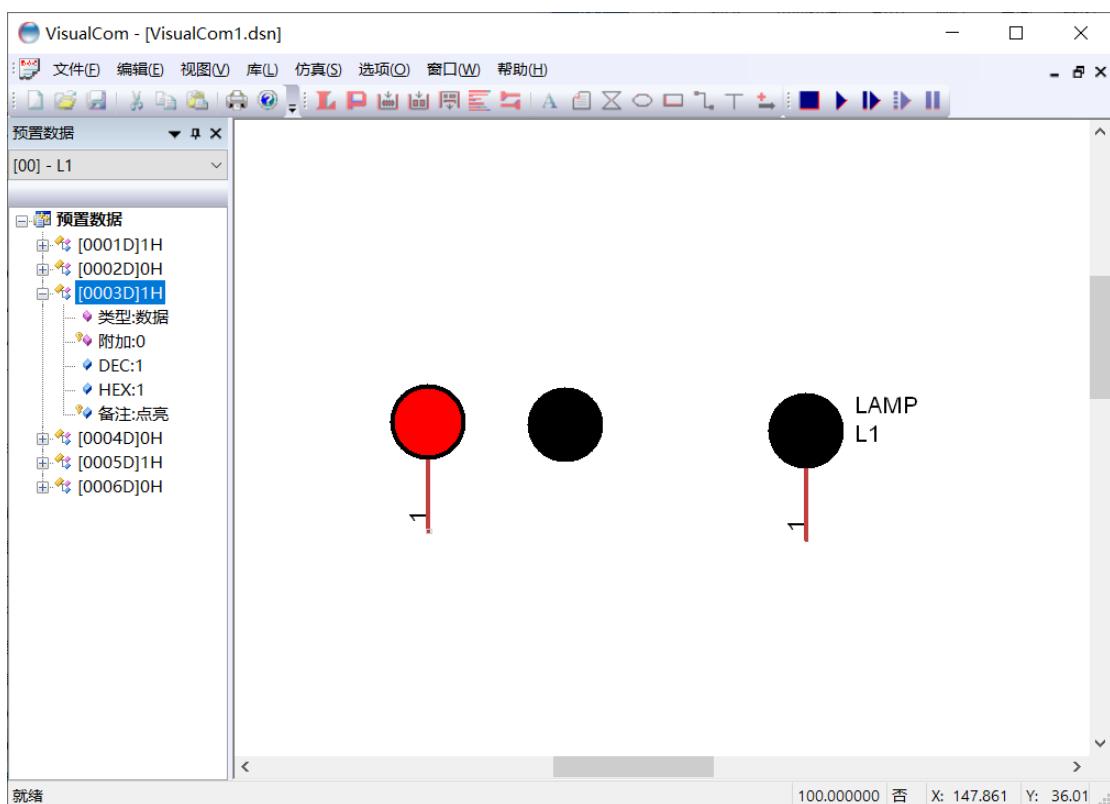
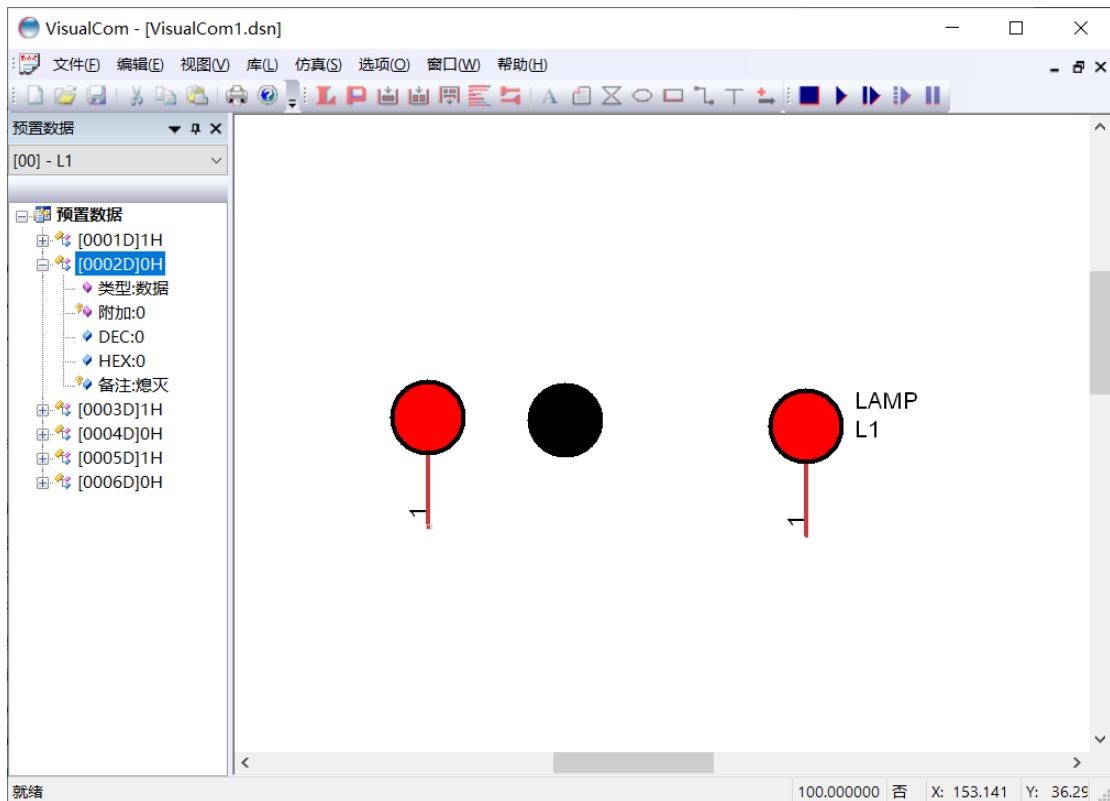
96     BOOL CLamp::Plot(ACTIVESTATE state)
97     {
98         if (m_RefreshFlag)
99         {
100             m_RefreshFlag = FALSE;
101             Animate(0, NULL); //调用Animate绘制图形
102             return TRUE;
103         }
104     }
105 }

```

Lamp.h 中仅添加了一个代表是否刷新视图的 BOOL 类型变量 m\_RefreshFlag ( FALSE 为否, TRUE 为是 ), 同时在 Lamp.cpp 的构造函数中将其初始化为 FALSE, 表示默认不刷新。

在 Simulate 函数中首先判断 GetProData 函数是否返回了有效数据 ( 如果答案是肯定的, 该函数将返回 TRUE ), 我们只有在获得了有效数据之后才将 m\_RefreshFlag 设置为 TRUE ( 用于在 Plot 函数中控制视图刷新 ), 并且 **Simulate 函数返回 TURE, 向系统表示预置数据已经处理, 这样“预置数据”窗口中的高亮项才会往下移**。而在 Plot 函数中, 我们先判断 m\_RefreshFlag 是否为 TRUE, 惟有如此才去执行 Animate, 之前在执行第一次预置数据时为什么会闪一下呢? 因为 Plot 函数总会被调用, 而 m\_LampStatus 初始值为 FALSE, 所以在进入仿真的那一瞬间 ( 预置数据还没有处理前 ), 黑色圆圈就被贴上去了。

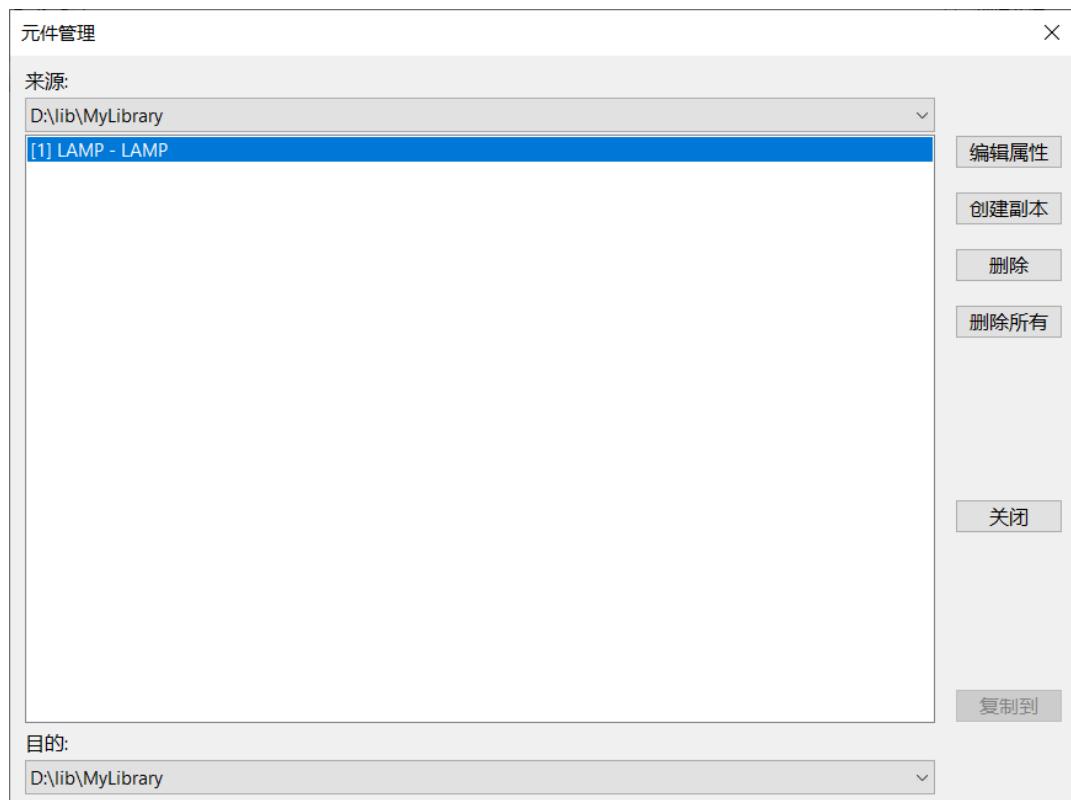
运行单步仿真后的状态如下图所示, 预置数据已经往下移了。



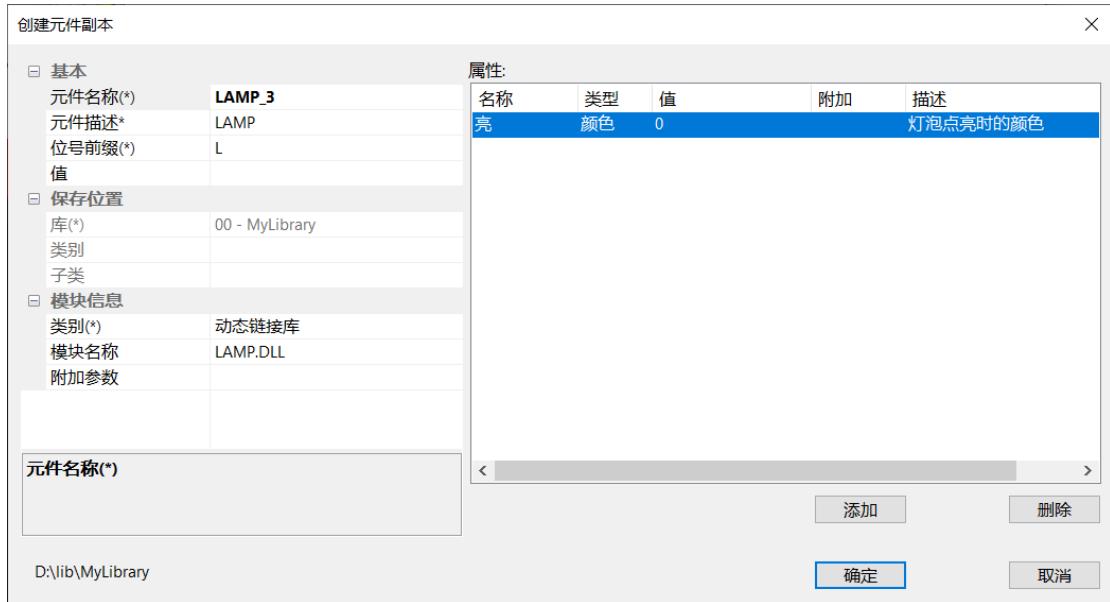
### 3.3 自定义灯泡的颜色

现在灯泡元件已经可以正常运行了，但是有些人可能会想：这个颜色我不是很喜欢，而且灯泡有很多种颜色，难道每种颜色都制作一种元件？Proteus 软件平台貌似是这么做的。其实没有必要，我们可以在创建元件时添加用户可以自定义的属性，然后在仿真模型中读取即可，接下来看看相应的步骤吧！在这之前，同样将之前的工程复制并重命名为 LAMP\_3。

( 1 )首先需要编辑属性。选择“库菜单”->“库元件管理”项，即可弹出下图所示“元件管理”对话框，目前只有一个库，而其中也仅有一个刚刚创建的元件。



我们将 LAMP 选中后单击右侧的“创建副本”按钮，表示使用当前元件的信息创建另一个元件（也可以单击“编辑属性”，这样就不会创建元件副本）。之后将弹出如图所示“创建元件副本”对话框，为区别于之前的 LAMP，可以将元件名称改为“LAMP\_3”（系统允许存在重名元件）。



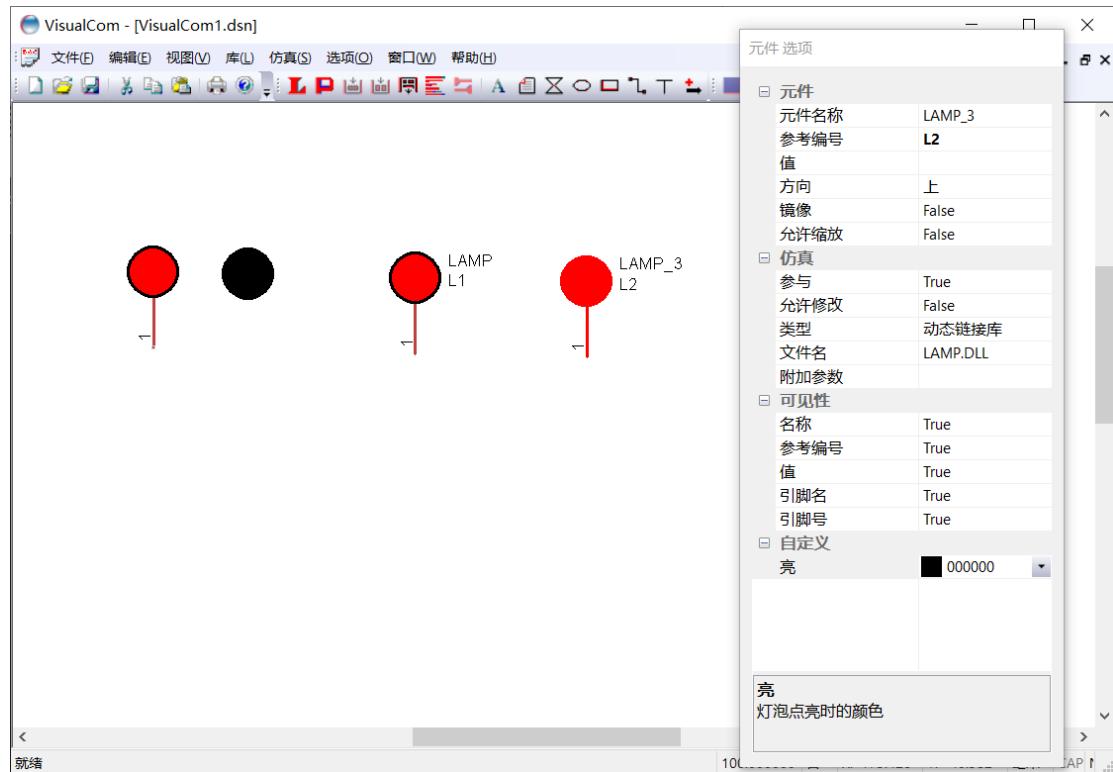
然后添加自定义属性。我们决定让用户自行输入点亮后的颜色。单击右侧的“添加”按钮即可加入一个属性行，其默认属性名称为“属性 100”，其 ID 默认为 100（用户自定义的属性 ID 均从 100 开始往下增加，更改属性名称不影响 ID 值）。

**“类型”列中包含文本、整数、布尔、颜色、文件、列表共 6 个选项**，这是用来做什么的呢？所有添加的属性默认都是文本，换句话说，默认情况下的信息都被当成文本。例如，你在仿真模型中读到某属性项的内容为“1”，但如果想将其作为整数，你需要进行转换。当然，你也可以选择“整数”类型，这样就可以通过一些函数直接读取到整数，也就省略了转换过程。

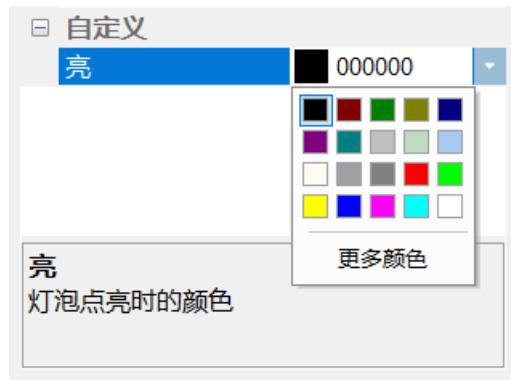
“值”列表示默认值（对于“布尔”类型，输入“0”表示 FALSE，输入“1”表示 TRUE）。  
**“附加”列表示可以控制列表的具体内容，如下表所示：**（颜色项请注意：R 与 B 值是反过来的。例如，红色是 0000FF，而不是 FF0000）。

类型	附加格式	说明(输入内容不含引号)
文本, 布尔, 颜色	无	无
整数	最小值,最大值	例如, “3,10” 表示将该属性项限制可输入的范围在 3~10 之间
文件	过滤显示文本 过滤格式	例如, “Bitmap Files(*.bmp) *.bmp” 表示打开文件对话框将仅显示*.bmp 格式文件
列表	项 1,项 2,项 3, …,项 n	例如, “RGB333,RGB444,RGB666” 表示列表中将显示三个选项。如果你选择了列表, 但并没有输入附加内容, 则仍然添加的是“文本”类型属性项

“描述”列的内容将出现在属性项描述中。以刚刚添加的“亮”属性项为例, 假设现在你已经添加了一个 LAMP\_3 元件到当前原理图中, 当选中元件时, “亮”属性项就会出现在“自定义”类别下, 具体如下图所示。



对于“颜色”类型的属性项，当你单击时会弹出一个可视化的颜色筛选器，你也可以单击“更多颜色”以调配符合要求的其它颜色，具体如下图所示。



好的，元件创建这一块的工作已经完成了，接下来开始进行仿真模型的开发。使用 Visual Studio 打开 LAMP\_3 文件夹下的工程，头文件 Lamp.h 的内容如下图所示。与之前有所不同的是，我们声明了一个 FILLSTYLE 类型的变量 m\_fs 用来保存用户输入的颜色（后面需要读取），其它并无不同。

```
1 #pragma once
2 #include "vsm.h"
3
4 class CLamp : public CDSIMMODEL
5 {
6     ICOMPONENT* component; //代表元件
7     BOOL m_LampStatus; //代表灯泡的状态, FALSE为灭, TRUE为亮
8     BOOL m_RefeshFlag; //是否刷新视图, FALSE为否, TRUE为是
9     DPOINT sym1_offset; //图形1的偏移位置
10    FILLSTYLE m_fs; //填充样式
11
12 public:
13     CLamp(); //构造函数
14     ~CLamp(); //析构函数
15     //以下为仿真模型开发需要实现的接口
16     CDSIMMODEL* GetSimModel(TCHAR* device);
17     ICOMPONENT* GetComponentPtr();
18     LONG GetTimeInterval(INT dat);
19     BOOL IntervalProcess(RUNMODES mode);
20     VOID Initialize(ICOMPONENT* cpt, DSIMMODES smode);
21     VOID Setup(SIMDATA* sdat);
22     BOOL Simulate(ABSTIME time, RUNMODES smode);
23     VOID CallBack(ABSTIME time, EVENTID eventid);
24     BOOL Indicate(REALTIME time, ACTIVEDATA* data);
25     VOID Animate(INT element, ACTIVEDATA* data);
26     BOOL Plot(ACTIVESTATE state);
27     BOOL Actuate(WORD key, DPOINT p, UINT flags);
28 }
```

Lamp.cpp 源文件的内容如下。首先在构造函数中对 m\_fs 进行了初始化，style 表示

填充的样式 ( HS\_SOLID 表示实心填充 ), color 表示填充颜色为红色, 这只是为了让变量

有一个确定值, 在进入仿真状态后还会初始化。

```
13  CLamp::CLamp()
14  {
15      m_LampStatus = FALSE;
16      m_RefreshFlag = FALSE;
17      sym1_offset.x = 0;
18      sym1_offset.y = 0;
19      m_fs.style = HS_SOLID;                                //实心填充
20      m_fs.color = BRIGHTRED;                             //红色
21  }
22
23  VOID CLamp::Initialize(ICOMPONENT* cpt, DSIMMODES smode)
24  {
25      component = cpt;                                    //初始化本地元件指针
26
27      sym1_offset = component->GetSymbolOffset(1);        //获取图形1的偏移位置
28
29      COLORREF color = 0;
30      if (component->GetColorFieldById(100, color))     //通过ID值获取用户定义的颜色
31      {
32          m_fs.color = color;
33      }
34  }
```

在 Initialize 函数中, 我们声明了一个 color 变量, 然后调用 GetColorFieldByID 函数读取 ID 为 100 的颜色, 如果该函数成功找到 ID 为 100 且类型为 “颜色” 的属性项, 该函数将返回 TRUE, 否则返回 FALSE, 读取完成就将相应的颜色保存在 m\_fs 变量中。最后在 Animate 函数中, 我们为 if 语句添加了一个 else 分支语句, 当 m\_LampStatus 为 TRUE 状态时, 同样绘制图形 1, 但此时的第 3 个形参不再是 NULL, 而是 m\_fs, 如下所示。

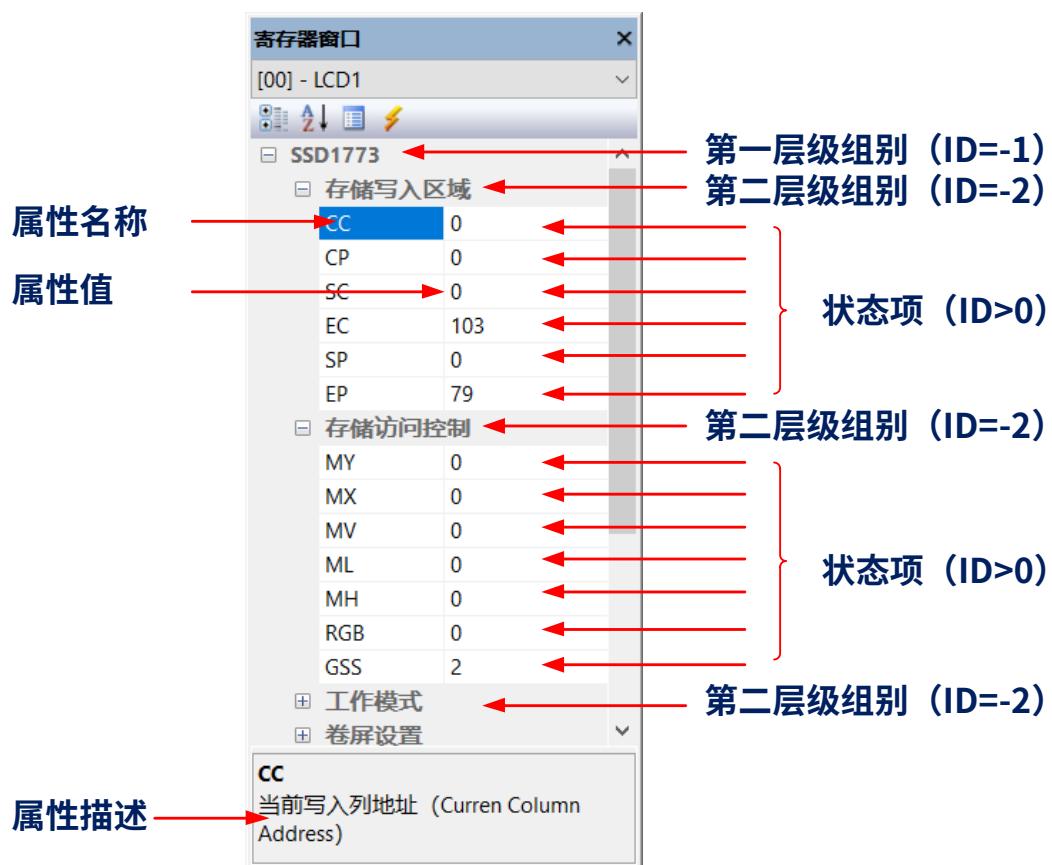
```
94  VOID CLamp::Animate(INT element, ACTIVEDATA* data)
95  {
96      component->BeginCache();                            //开始缓存
97
98      if (!m_LampStatus)                                  //如果为熄灭状态, 就绘制图形1
99      {
100          component->DrawSymbol(1, NULL, NULL, NULL, sym1_offset.x, sym1_offset.y);
101      }
102      else                                              //如果为点亮状态, 也绘制图形1, 但颜色为用户自定义
103      {
104          component->DrawSymbol(1, NULL, &m_fs, NULL, sym1_offset.x, sym1_offset.y);
105      }
106
107      component->EndCache();                           //结束缓存
108  }
```

最后请思考一下: 如果需要根据用户的需求分别改变灯泡点亮与熄灭的颜色, 该怎么做呢? 如果需要更改圆圈边框的颜色呢? ( 提示: LINESTYLE )

### 3.4 将灯泡状态添加到寄存器窗口

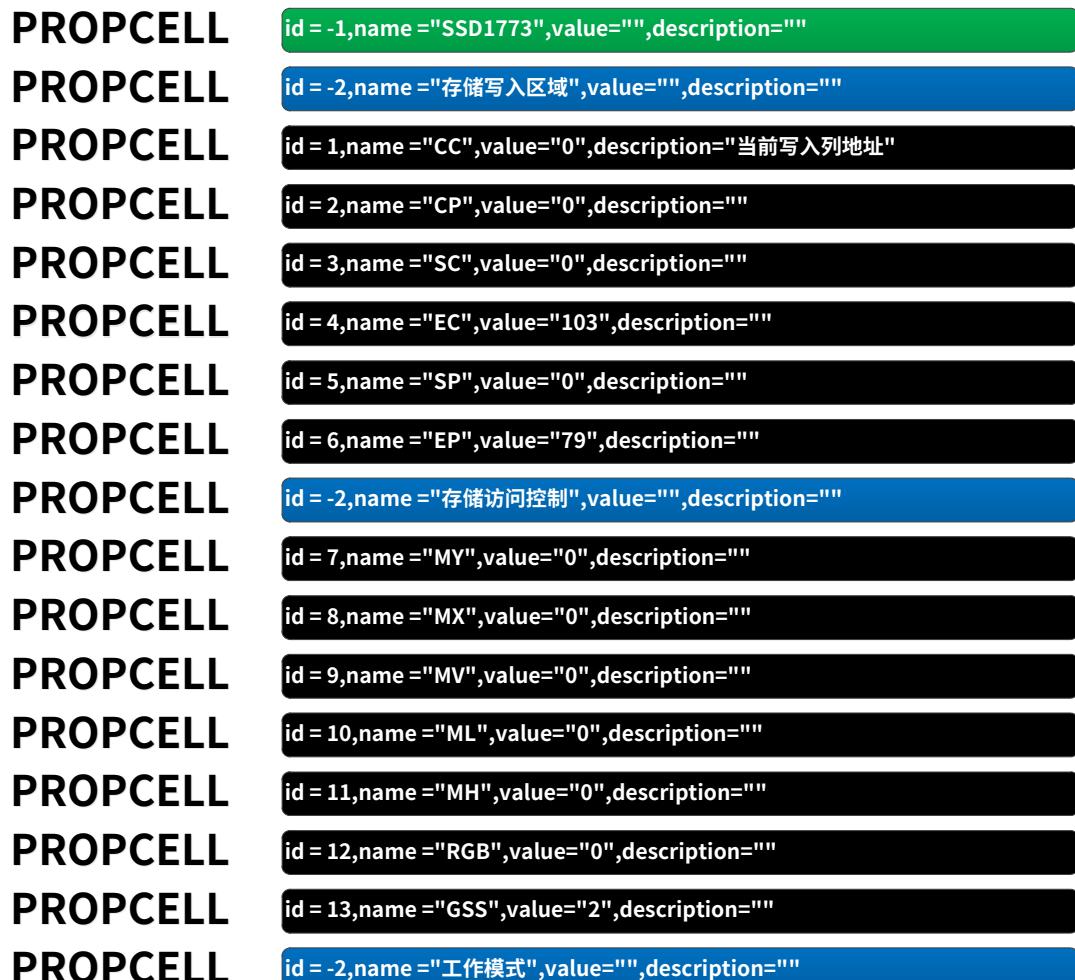
是不是看到系统自带的很多液晶显示模组元件在仿真时可以看到“寄存器”、“内存”或“指令集”窗口会实时出现当前的状态？虽然像灯泡元件如此简单的元件是不会存在什么寄存器等状态，但如果你想将一些状态加入到寄存器或其它窗口，也是完全可以的。一句话，“寄存器”、“内存”或“指令集”窗口中显示的内容由你自己全权决定，本节以“寄存器”窗口为例进行讨论，“内存”与“指令集”窗口也是相似的。

为了将状态正确显示到“寄存器”窗口当中，你需要理解“寄存器”窗口中的数据是如何组织。每一个寄存器状态项与属性项一样，都包含属性名称、属性值与属性描述，那么这些状态项是如何组织起来的呢？寄存器窗口中包含**层级组别**与**具体状态项**两类元素，前者仅有名称，后者则包含名称，值，描述、ID信息，类似如下图所示



为了让系统能够正确加载你的数据，你需要将所有寄存器状态组织都放在一个 PROPCELLS 类型链表中，而元素的识别则由 ID 决定。如果你想添加一个层级，就应该添加一个 ID 在-1~5 之间的 PROPCELL（值越小表示层级越低）。如果你想添加一个具体的状态项，则添加一个 ID 大于 0 的 PROPCELL（0 不使用）。另外，还需要特别注意一点：

**为了后续能够更新数据，必须为每个状态项添加唯一的 ID。**下图可以描述上图对应的链表结构（省略了属性描述 description）。



**此处省略ID>0且与前述项ID不同的状态项**



**此处省略ID>0且与前述项ID不同的状态项**

本节我们以“将 LED 的状态添加到‘寄存器’窗口”作为演示，同样将前述工程复制一份并重新命名为 LAMP\_4，进入工程后更改 Lamp.h 如下图所示，其中我们增加了一个

用来获取 PROPCELLS 链表的 GetRegisterPage 函数。

```
1  #pragma once
2  #include "vsm.h"
3
4  class CLamp : public CDSIMMODEL
5  {
6      ICOMPONENT* component; //代表元件
7      BOOL m_LampStatus; //代表灯泡的状态, FALSE为灭, TRUE为亮
8      BOOL m_RefeshFlag; //是否刷新视图, FALSE为否, TRUE为是
9      DPOINT syml_offset; //图形1的偏移位置
10     FILLSTYLE m_fs; //填充样式
11
12     public:
13         CLamp(); //构造函数
14         ~CLamp(); //析构函数
15         PROPCELLS GetRegisterPage(); //获取添加到寄存器窗口的状态
16     public: //以下为仿真模型开发需要实现的接口
17         CDSIMMODEL* GetSimModel(TCHAR* device);
18         ICOMPONENT* GetComponentPtr();
19         LONG GetTimeInterval(INT dat);
20         BOOL IntervalProcess(RUNMODES mode);
21         VOID Initialize(ICOMPONENT* cpt, DSIMMODES smode);
22         VOID Setup(SIMDATA* sdat);
23         BOOL Simulate(ABSTIME time, RUNMODES mode);
24         VOID CallBack(ABSTIME time, EVENTID eventid);
25         BOOL Indicate(REALTIME time, ACTIVEDATA* data);
26         VOID Animate(INT element, ACTIVEDATA* data);
27         BOOL Plot(ACTIVESTATE state);
28         BOOL Actuate(WORD key, DPOINT p, UINT flags);
29     };

```

Lamp.cpp 源文件内容如下图所示，首先看看 GetRegisterPage 函数，其中声明了一个 PROPCELLS 类型的变量 cellCollection，后续所有需要添加到“寄存器”窗口对应的 PROPCELL 都依序插入这个链表中，它也是该函数最后返回的对象。之后声明了一个 PROPCELL 类型的变量 cell，之后就是填充 cell 变量并插入到 cellCollection 中的过程。

代码首先创建了一个第一层级组别，其 ID 为-1，名称为“LED”（层级级别不需要值与描述）。之后又添加了第二个层级组别，其 ID 为-2，名称为“所有”，同样不需要值与描述（也可不插入此层级级别，仅作为演示）。当然，你还可以插更多层级（层级 ID 的最小值为-5）。之后就添加一个具体的状态项，之前将 index 初始化为 1，之所以用 index++，主要是为了多个状态项的插入方便（此处只有一个状态项，其实并无必要）。当需要的值都插入链表中后，然后返回 cellCollection 的副本即可。

那么 GetRegisterPage 函数应该在哪里调用呢？“寄存器”，“内存”，“指令集”窗口中的数据**仅在单步运行仿真时才有效**，所以你需要在处理有效预置数据后执行。所以在 Simulate 函数中取出了 PROPCELL 保存在 cells 中，然后调用 SegRegMemStatus 函数将其设置即可。第二个与第三个形参分别对应“内存”与“指令集”窗口，它们的结构完全与寄存器窗口相同，此处不再赘述（如果不需要设置某个窗口中的数据，将该形参设置为 NULL 即可）。

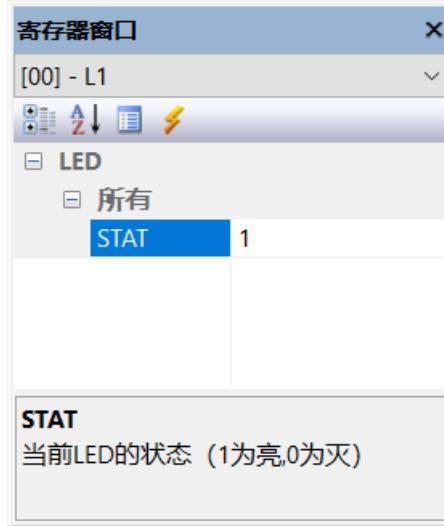
```
65  BOOL CLamp::Simulate(ABSTIME time, RUNMODES smode)
66  {
67      PRODATA pro_data_tmp;
68      if (component->GetProData(pro_data_tmp)) //获取预置数据
69      {
70          if (pro_data_tmp.data & 0x1) //判断预置数据最低位
71          {
72              m_LampStatus = TRUE; //设置灯泡为点亮状态
73          }
74          else
75          {
76              m_LampStatus = FALSE; //设置灯泡为熄灭状态
77          }
78
79          PROPCELLS cells = GetRegisterPage();
80          component->SetRegMemStatus(&cells, NULL, NULL); //设置寄存器窗口
81
82          m_RefreshFlag = TRUE;
83          return TRUE;
84      }
85      return FALSE;
86  }
87
88  PROPCELLS CLamp::GetRegisterPage()
89  {
90      PROPCELLS cellCollection; //返回的包括所有状态的链表
91      PROPCELL cell; //包含某个状态的临时单元，填充状态数据后再插入链表
92      UINT index = 1; //寄存器窗口中每个显示单元的惟一ID（多个状态时）
93
94      char str_name_tmp[512];
95      char str_value_tmp[512];
96      char str_info_tmp[512];
97
98      sprintf(str_name_tmp, "%s", "LED"); //第一层级组别名
99      sprintf(str_value_tmp, "%s", "");
100     sprintf(str_info_tmp, "%s", "");
101
102     cell.id = -1; //创建第一层级组别
103     cell.name = component->GetTCHARFromChar(str_name_tmp);
104     cell.value = component->GetTCHARFromChar(str_value_tmp);
105     cell.description = component->GetTCHARFromChar(str_info_tmp);
106     cellCollection.push_back(cell);
107 }
```

```

149     sprintf(str_name_tmp, "%s", "所有");      //创建第二层级组别（非必要，用于演示）
150     cell.id = -2;
151     cell.name = component->GetTCHARFromChar(str_name_tmp);
152     cell.value = component->GetTCHARFromChar(str_value_tmp);
153     cell.description = component->GetTCHARFromChar(str_info_tmp);
154     cellCollection.push_back(cell);
155
156
157     sprintf(str_name_tmp, "%s", "STAT");
158     sprintf(str_value_tmp, "%s", (m_LampStatus?"1":"0"));
159     sprintf(str_info_tmp, "%s", "当前LED的状态 (1为亮,0为灭)");
160     cell.id = index++;                         //创建具体的状态项
161     cell.name = component->GetTCHARFromChar(str_name_tmp);
162     cell.value = component->GetTCHARFromChar(str_value_tmp);
163     cell.description = component->GetTCHARFromChar(str_info_tmp);
164     cellCollection.push_back(cell);
165
166
167     return cellCollection;
}

```

我们来看看相应状态，如下图所示。



最后提示一下：如果你想往“输出”窗口的“信息”标签页中添加数据，只需要调用 SetOutputInfo 函数即可。

### 3.5 可控制的闪烁灯泡

现在再开发另一个新功能的灯炮：**假设该元件拥有与之前相同的预置数据，当最低位为 1 时灯泡闪烁 (500ms 间隔)，而为 0 时则不闪烁**。我们直接复制之前的 LAMP\_4 文件夹并将其重命名为 LAMP\_5，将其中的工程打开后，相应的 Lamp.h 头文件代码如下图所示：

```

1   #pragma once
2   #include "vsm.h"
3
4   class CLamp : public CDSIMMODEL
5   {
6       ICOMPONENT* component;           //代表元件
7       BOOL m_LampStatus;             //代表灯泡的状态, FALSE为灭, TRUE为亮
8       BOOL m_RefreshFlag;            //是否刷新视图, FALSE为否, TRUE为是
9       DPOINT sym1_offset;           //图形1的偏移位置
10      FILLSTYLE m_fs;              //填充样式
11
12      BOOL m_LampCtrl;              //控制灯泡的状态
13      LONG m_SysInvokingCount;     //需要系统调用的次数
14      LONG m_CurrentInvokingCount; //当前调用次数
15
16      public:
17          CLamp();                  //构造函数
18          ~CLamp();                 //析构函数
19          PROPCELLS GetRegisterPage(); //获取添加到寄存器窗口的状态
20
21          public:                   //以下为仿真模型开发需要实现的接口
22              CDSIMMODEL* GetSimModel(TCHAR* device);
23              ICOMPONENT* GetComponentPtr();
24              LONG GetTimeInterval(INT dat);
25              BOOL IntervalProcess(RUNMODES mode);
26              VOID Initialize(ICOMPONENT* cpt, DSIMMODES smode);
27              VOID Setup(SIMDATA* sdat);
28              BOOL Simulate(ABSTIME time, RUNMODES mode);
29              VOID CallBack(ABSTIME time, EVENTID eventid);
30              BOOL Indicate(REALTIME time, ACTIVEDATA* data);
31              VOID Animate(INT element, ACTIVEDATA* data);
32              BOOL Plot(ACTIVESTATE state);
33              BOOL Actuate(WORD key, DPOINT p, UINT flags);
34      };

```

为了控制闪烁灯泡的状态，我们额外声明了一个 m\_LampCtrl，其值为 FALSE 表示不闪烁，TRUE 表示闪烁。那么 m\_SysInvokingCount 与 m\_CurrentInvokingCount 又是什么呢？之前已经提过，系统会启动一个线程循环调用 IntervalProcess 函数，但每隔多长时间会调用呢？系统仅支持 1ms、10ms、100ms、1000ms 共 4 个等级，你可以通过实现 GetTimeInterval 函数返回一个值，当其返回值在 1、2~10、11~100、>100 时，系统调用 IntervalProcess 函数的对应时间间隔可能是 1ms、10ms、100ms、1000ms。

之所以描述为可能，是因为当前原理图中可能存在多个需要指定不同时间间隔的元件，而系统在对所有元件调用 GetTimerInterval 函数获取时间间隔后，会将时间间隔设置为其中的最小值，所以当前元件返回的时间间隔可能并不是最终系统调用相应仿真模型中 IntervalProcess 函数的时间间隔，所以你需要使用 ICOMPONENT 结构体中的 GetThreadTimeInterval 函数确定最终的时间间隔（除非仿真模型返回值为 1）。

举个例子，当前原理图中存在两个分别返回 10 与 100 的元件 A 与 B，当进入仿真状态后，系统会将时间间隔设置为 10ms，这也是你使用 ICOMPONENT 结构体中的 GetThreadTimeInterval 函数获得的值。对于元件 B 而言（对于元件 A 也同样如此），你需要先通过时间间隔计算调用次数（此处为  $100/10=10$ ），仿真模型中每 10 次调用 IntervalProcess 函数后才是真正执行周期性代码的时机。这里的 m\_SysInvokingCount 就是根据系统最终设置的时间间隔计算出来的调用次数，而 m\_CurrentInvokingCount 变量就是每调用一次就累加一次的计数值，如果计数值到达 m\_SysInvokingCount，说明时间间隔已经到，这时应该执行需要周期更新的代码，同进应该将 m\_CurrentInvokingCount 清零准备下一次计算，简单吧！

Lamp.cpp 源文件中主要代码如下图所示。构造函数中的初始化代码不必多言，我们直接看 GetTimeInterval 函数，其中返回了 100，这也就意味着，系统最终设置的时间间隔不会小于此时，但到底是多少呢？我们在 Setup 函数中（**此时所有元件都已经调用完了 GetTimeInterval 函数**）调用了 GetThreadTimeInterval 函数获取了具体的时间间隔，然后计算相应的次数。例如，当前系统的时间间隔为 10ms，而需要的时间为 500ms，则应该设置为 50，其它依此类推。而在 IntervalProcess 函数中，如果 m\_LampCtrl 为 TRUE，表示灯泡进入闪烁状态。当到达 500ms 时，将灯泡的状态取返即可。而当 m\_LampCtrl 为 FALSE 时，就将次数清零，同时将灯泡设置为处于熄灭状态。

```

13  CLamp::CLamp()
14  {
15      m_LampStatus = FALSE;
16      m_RefreshFlag = FALSE;
17      sym1_offset.x = 0;
18      sym1_offset.y = 0;
19      m_fs.style = HS_SOLID; //实心填充
20      m_fs.color = BRIGHTRED; //红色
21
22      m_LampCtrl = FALSE;
23      m_SysInvokingCount = 0;
24      m_CurrentInvokingCount = 0;
25  }
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41  LONG CLamp::GetTimeInterval (INT dat)
42  {
43      return 100; //返回仿真模型要求的时间间隔
44  }
45
46  BOOL CLamp::IntervalProcess (RUNMODES mode)
47  {
48      if (m_LampCtrl)
49      {
50          if (m_CurrentInvokingCount < m_SysInvokingCount)
51          {
52              m_CurrentInvokingCount++;
53          }
54          else
55          {
56              m_CurrentInvokingCount = 0;
57              m_LampStatus = (!m_LampStatus); //状态取反
58          }
59          m_RefreshFlag = TRUE;
60      }
61      else
62      {
63          m_CurrentInvokingCount = 0;
64          m_LampStatus = FALSE;
65      }
66
67      return FALSE;
68  }
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83  VOID CLamp::Setup (SIMDATA* sdat)
84  {
85      LONG t = component->GetThreadTimeInterval(); //判断系统时间间隔
86
87      if (t > 10)
88      {
89          m_SysInvokingCount = 5;
90      }
91      else if (t > 1)
92      {
93          m_SysInvokingCount = 50;
94      }
95      else if (t == 1)
96      {
97          m_SysInvokingCount = 500;
98      }
99  }

```

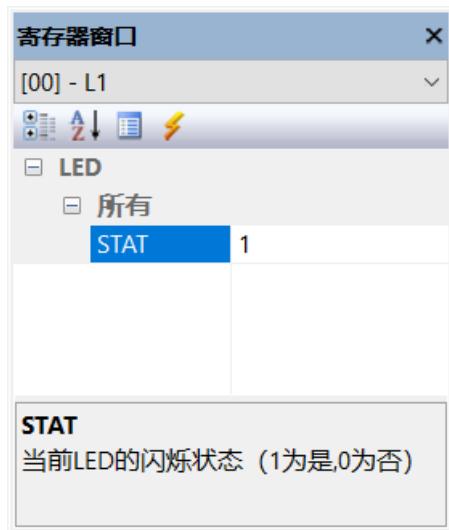
```

101    □BOOL CLamp::Simulate(ABSTIME time, RUNMODES smode)
102    {
103        □ PRODATA pro_data_tmp;
104        if (component->GetProData(pro_data_tmp)) //获取预置数据
105        {
106            □ if (pro_data_tmp.data & 0x1) //判断预置数据最低位
107            {
108                m_LampCtrl = TRUE; //设置灯泡为点亮状态
109            }
110            else
111            {
112                m_LampCtrl = FALSE; //设置灯泡为熄灭状态
113            }
114
115            PROPCELLS cells = GetRegisterPage();
116            component->SetRegMemStatus(&cells, NULL, NULL); //设置寄存器窗口
117
118            m_RefreshFlag = TRUE;
119            return TRUE;
120        }
121    }
122
166    □PROPCELLS CLamp::GetRegisterPage()
167    {
168        PROPCELLS cellCollection; //返回的包括所有状态的链表
169        PROPCELL cell; //包含某个状态的临时单元，填充状态数据后再插入链表
170        UINT index = 1; //寄存器窗口中每个显示单元的惟一ID（多个状态时）
171
172        char str_name_tmp[512];
173        char str_value_tmp[512];
174        char str_info_tmp[512];
175
176        sprintf(str_name_tmp, "%s", "LED"); //第一层级组别名
177        sprintf(str_value_tmp, "%s", "");
178        sprintf(str_info_tmp, "%s", "");
179
180        cell.id = -1; //创建第一层级组别
181        cell.name = component->GetTCHARFromChar(str_name_tmp);
182        cell.value = component->GetTCHARFromChar(str_value_tmp);
183        cell.description = component->GetTCHARFromChar(str_info_tmp);
184        cellCollection.push_back(cell);
185
186        sprintf(str_name_tmp, "%s", "所有"); //创建第二层级组别（非必要，用于演示）
187        cell.id = -2;
188        cell.name = component->GetTCHARFromChar(str_name_tmp);
189        cell.value = component->GetTCHARFromChar(str_value_tmp);
190        cell.description = component->GetTCHARFromChar(str_info_tmp);
191        cellCollection.push_back(cell);
192
193        sprintf(str_name_tmp, "%s", "STAT");
194        sprintf(str_value_tmp, "%s", (m_LampCtrl ? "1" : "0"));
195        sprintf(str_info_tmp, "%s", "当前LED的闪烁状态（1为是,0为否）");
196        cell.id = index++; //创建具体的状态项
197        cell.name = component->GetTCHARFromChar(str_name_tmp);
198        cell.value = component->GetTCHARFromChar(str_value_tmp);
199        cell.description = component->GetTCHARFromChar(str_info_tmp);
200        cellCollection.push_back(cell);
201
202    }
203

```

值得一提的是，如果 IntervalProcess 返回 TRUE，则表示刷新视图，此刷新时间间隔独立于系统设置的刷新间隔，你可以自行尝试一下，此处不再赘述。

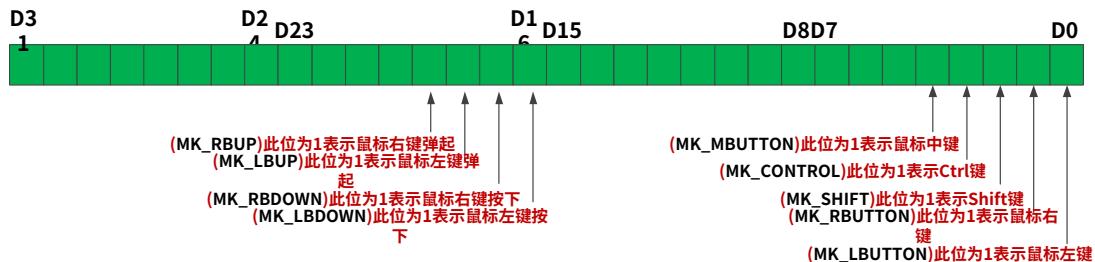
最后提醒，此时寄存器窗口应该是 m\_LampCtrl 的状态（而不是 m\_LampStatus），如下图所示。



## 3.6 单击动作改变灯泡的状态

通过预置数据改变灯泡的闪烁实在太不高级了，所以我们打算修改一下：**当鼠标单击元件时，灯泡就会闪烁，当鼠标再次单击时，灯泡就不再闪烁**。怎么样获取单击事件呢？当你在系统中单击后，系统会给元件发送当前的单击坐标（以元件右下角为参考），同时还跟随了一个 flags 标记。例如 MK\_LBUTTON 表示左键按下，MK\_RBUTTON 表示右键键下。为了响应单击事件，首先你需要判断该坐标是否在元件的范围内，如果答案是肯定的，就再判断标记位，就这么简单！

实现响应鼠标事件的关键之处在于 Actuate 函数，鼠标与按键输入都将通过该函数处理。如果是鼠标事件发生，key 值将会是 0，而 p 值为单击坐标（以元件右下角为参考），flags 值则表示发生动作的是左键、中键、右键、Ctrl 键、Shift 键以及按下还是弹起事件，具体如下图所示：



我们直接复制之前的 LAMP\_5 文件夹并将其重命名为 LAMP\_6，将其中的工程打开后，头文件不需要修改，源文件的主要修改之处如下图所示。

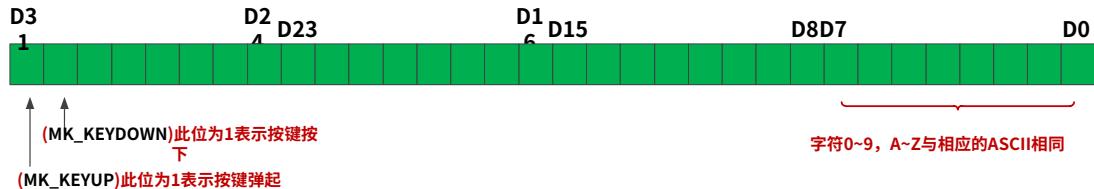
```
101     BOOL CLamp::Simulate(ABSTIME time, RUNMODES smode)
102     {
103     }
104 }
```

```
143     BOOL CLamp::Actuate(WORD key, DPOINT p, UINT flags)
144     {
145
146         if (component->InsideComponent(p))          //判断单击处坐标是否处于元件范围内
147         {
148
149             if (flags & MK_LBUTTON)
150             {
151                 m_LampCtrl = (!m_LampCtrl);
152
153                 m_RefreshFlag = TRUE;
154             }
155
156             /***以下实现鼠标左键按下点亮灯泡，松开熄灭灯泡
157             if (flags & MK_LBUTTONDOWN)
158             {
159                 m_LampCtrl = TRUE;
160             }
161             else if (flags & MK_LBUP)
162             {
163                 m_LampCtrl = FALSE;
164             }
165
166             m_RefreshFlag = TRUE;
167             */
168         }
169
170     }
171 }
```

代码中首先调用 InsideComponent 函数判断鼠标单击位置是否处于元件范围内（其实就是先通过 GetSize 函数获取元件的尺寸，然后分别判断 X 轴与 Y 轴大小），如果答案是肯定的，就判断 MK\_LBUTTON 标记再进行相应的处理即可。如果想实现鼠标左键按下就闪烁，松开就不闪烁呢？你可以进一步判断是否为按下或弹起状态，具体见注释行代码：

### 3.7 按键动作改变灯泡的状态

使用按键也同样可能改变灯泡的状态，当在系统中按下按键时，同样会调用 Actuate 函数，其中 key 的低 8 位包含的按键码，还有两位表示为按下还是弹起，具体如下图所示：



我们需要实现“A”键控制灯泡闪烁（按一次闪烁，再按一次不闪烁，依此类型），相应的代码如下图所示，此处不再赘述。

```
143     BOOL CLamp::Actuate(WORD key, DPOINT p, UINT flags)
144     {
145         if ((key & VK_KEY_MASK) == 'A') //判断按键码
146         {
147             if (key & MK_KEYDOWN) //判断是否为按下
148             {
149                 m_LampCtrl = (!m_LampCtrl);
150                 m_RefreshFlag = TRUE;
151             }
152         }
153     }
154     return FALSE;
155 }
```

# 第4章 与远程模块连接

从V2.1.0开始，你可以将元件模型与远程硬件模块连接，这样就可以控制硬件输出或采集数据输入（从而与元件模型进行各种交互行为），换句话说，当存在硬件模块与VisualCom软件平台连接时，你也就相当于拥有海量的元器件，它们可以实实在在地与现实的系统（例如，单片机、FPGA、DSP）进行通讯，并且能够真实模拟元器件的行为，而不需要再花费大量成本去重复购买元器件。

本小节详尽阐述如何使元件模型与远程硬件模块连接及开发时需要注意的事项。

## 4.1 元件创建

如果你已经决定与远程模块连接，创建元件时必须详细且正确地定义引脚参数，首要的一点便是：**必须为每一个引脚分配唯一的编号（可以是数字或字母），这一点极其重要，而引脚名则是可选的**。右图中的引脚编号为“6”，引脚名称为“E”。

其次，引脚的“特殊”参数也应该详细定义。虽然“特殊”参数的定义并非必须（因为它们都可以在“引脚分配”对话框中再次更改），但如果元件创建时已详细定义，后续进行引脚分配时就不需要重复去修改（因为每次调用元



件到原理图中的默认引脚参数都不是正确的)，也就能节省很多重复的工作。

“特殊”参数主要包括“电气类型”、“默认电平”、“触发类型”（“控制器”与“引脚类型”为预留）。“电气类型”包括“输入”、“输出”、“双向”、“电源”、“接地”、“无”共6种，你可以在模型开发时获取该值。需要注意的是，只有“输出”类型的引脚才会将硬件模块的引脚设置为“输出”，其它类型的引脚都会默认设置为输入。当然，无论此处设置的引脚电气类型为何，你都可以在元件模型开发过程中随时通过调用函数改变（例如，**硬件模块并没有“双向”类型，但可以在模型开发中通过随时修改引脚类型来实现**）。

创建元件时设置“电气类型”的另一个意义在于：如果某些引脚并不打算在元件模型中使用（只是为了在原理图中“看”），那么你可以设置为“电源”、“接地”或“无”，如此一来，使用“引脚分配”对话框中的**自动引脚分配功能**时将不会对此类型引脚予以处理（后述）。

“默认电平”是该引脚对应的默认电平，这也是在进入仿真状态后，远程模块各个引脚对应的电平状态（如果引脚已分配有效的引脚编号），你也可以在开发模型中获取此值。

“触发类型”可以定义一些事件触发采集数据。默认情况下，远程模块会以100ms（当然，你可以修改甚至关闭）的间隔采集数据并返回到VisualCom软件平台，这些数据称为定时（等时）采样数据，但是有些时候，一些事件触发采集数据对一些时序应用非常方便。例如，串接通讯接口的时钟通常就有触发边沿，如果时钟的周期小于定时采集间隔，定时采样就不一定能够采集到，此时就可以定义该引脚为触发类型，可选“无”（默认），“上升沿”、“下降沿”、“双边沿”（触发类型本质上由远程模块中单片机的中断来完成）。

需要注意的是，触发类型的引脚设置应该根据功能来定义，假设你现在开发“四线SPI数据分析接口”元件，由于SPI通常在时钟边沿采集或输出数据，所以这种接口只需要将时钟引脚设置触发类型即可，虽然你也可以将所有引脚都设置触发类型，但是这样做对数据的解析并没有太大的意义（除非你想将每个引脚的每个瞬间记录下来），反而还增加了数据传

输的负担。另外，触发类型引脚是珍贵的资源，虽然每个引脚都可以配置为触发类型，但同时可以使用的引脚却是有限的，所以并不建议在不需要触发类型引脚时而将其设置触发类型，这里举几个例子进一步阐述触发类型的设置依据，你也可以观察系统自带库中的元件及相应的引脚配置（通常情况下，系统自带库中元件的引脚触发类型配置是相对较优的）。

（1）简单的 LED、直流电机、蜂鸣器、逻辑器件等不需要设置触发类型，因为这些元件仅根据等时采样的数据即可完成所有动作

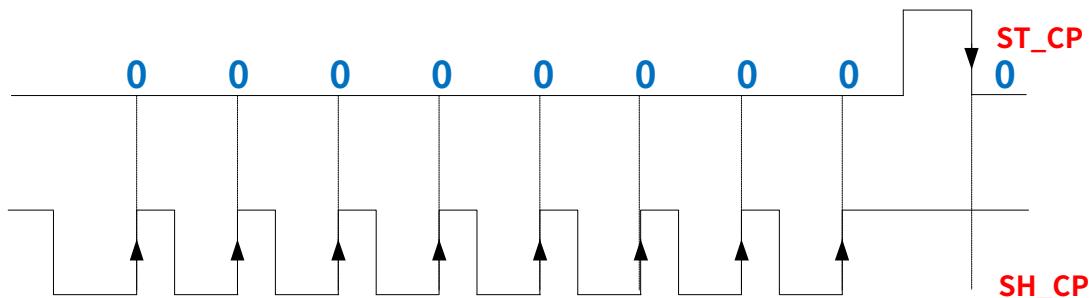
（2）动态扫描的 LED 或点阵是否需要设置触发类型呢？设置触发类型当然可以实现，但以 8 位动态扫描的数码管来说，你需要 8 个触发引脚，这已经接近可同时配置触发类型数量的极限，如果当前设计中还存在其它一些时序控制元件，就很可能不够用了，所以这类型器件不必设置触发类型，只需要将使用等时采样的数据（将采样速率提升即可），详情见系统自带库中的动态扫描 LED 数码管或点阵元件。

（3）6800 接口仅需要将引脚“E”设置为触发类型（下降沿）即可，而将其它引脚设置为触发类型对元件模型解析数据并没有太大的意义，而且还加重了数据传输到主机的负担。8080 接口仅需要将引脚“WR\_N”与“RD\_N”设置为触发类型（上升沿）即可。

（4）3 线 SPI 接口只需要将“CLK”设置为触发类型即可。  
当然，有些特殊情况下也可以增加触发类型引脚。例如，PT6312 使用的串行接口有些特殊，其发送的第一个字节总是指令，连续发送的字节则都是数据。在元件模型中，虽然我们可以根据时钟的数量来判断并接收数据，但是在这种情况下，仅将 CLK 引脚设置为触发类型是不够的，因为你无法判断连续发送的下一个字节是经过 CS 拉高再拉低发送（也就是再次发送指令），还是未改变 CS 连续发送（也就是发送数据）。为此可以将 CS 的设置为触发边沿，也就可以通过判断第二个字节之前 CS 是否触发来判断数据类型。

再以 74HC595 为例，理论上只需要将移位时钟（上升沿）与锁存时钟（下降沿）设置

为触发类型即可，但是在数据判断逻辑上可能会存在一些问题，因为锁存时钟的下降沿可能无法检测到（考虑到接口协议的兼容性），为什么呢？我们来看下图，移位时钟 SH\_CP 每次上升沿到来，也会返回 ST\_CP 当前的状态（此例为低电平），所以数据移位的操作是可以完成的，但是当锁存时钟 ST\_CP 的下降沿到来会发生什么呢？此时采集到的 ST\_CP 数据也是低电平，所以，你无法判断 ST\_CP 的低电平是由移位还是锁存时钟采集的，也就很难适实进行锁存。所以，我们应该将 ST\_CP 设置为双边沿触发。



有些人可能会想：为什么 LCD1602 只需要将引脚 E 设置为下降沿（而不需要设置为双边沿）呢？它们的数据锁存协议是相似的呀！因为 LCD1602 只有一个触发类型引脚，只要有事件数据采集进来，很容易就可以判断是引脚 E 引发的下降沿。

总之，触发类型的设置不一定必须与实际元件的原理相关，关键在于让元件模型在“消耗更少的触发类型引脚”的前提下也能够正确解析数据。

言归正传，我们的灯泡元件只有一个引脚，其编号默认为“**1**”，名称为“**空**”，“电气类型”则默认为“**输入**”，“默认电平”为“**高**”，也算符合灯泡元件的要求，所以这部分不需要额外进行更改，但对于你创建的其它新器件，应该对“特殊”参数进行适当地调整。

## 4.2 模型开发

灯泡元件既然已经创建完成，接下来开始进行仿真模型的开发。我们决定在前述工程 LAMP\_3 的基础上更改，将其复制后重新命名为 LAMP\_8，接下来就开启连接远程模块之

旅吧！

### 4.2.1 声明引脚（或总线）

首先应该声明一个引脚用来接收(或发送)数据,相应头文件 Lamp.h 内容如下图所示,只是在之前的基础上声明了 SIMPIN 类型的 input\_pin, 它代表着与远程引脚之间的联系(当然, 后面还要初始化)。

```
1      #pragma once
2      #include "vsm.h"
3
4      class CLamp : public CDSIMMODEL
5      {
6          ICOMPONENT* component;           //代表元件
7          BOOL m_LampStatus;   //代表灯泡的状态, FALSE为灭, TRUE为亮
8          BOOL m_RefreshFlag;  //是否刷新视图, FALSE为否, TRUE为是
9          DPOINT sym1_offset;           //图形1的偏移位置
10         FILLSTYLE m_fs;             //填充样式
11     public:
12         SIMPIN input_pin;           //声明引脚
13     public:
14         CLamp();                  //构造函数
15         ~CLamp();                //析构函数
16
17     public:                      //以下为仿真模型开发需要实现的接口
18         CDSIMMODEL* GetSimModel(TCHAR* device);
19         ICOMPONENT* GetComponentPtr();
20         LONG GetTimeInterval(INT dat);
21         BOOL IntervalProcess(RUNMODES mode);
22         VOID Initialize(ICOMPONENT* cpt, DSIMMODES smode);
23         VOID Setup(SIMDATA* sdat);
24         BOOL Simulate(ABSTIME time, RUNMODES smode);
25         VOID CallBack(ABSTIME time, EVENTID eventid);
26         BOOL Indicate(REALTIME time, ACTIVEDATA* data);
27         VOID Animate(INT element, ACTIVEDATA* data);
28         BOOL Plot(ACTIVESTATE state);
29         BOOL Actuate(WORD key, DPOINT p, UINT flags);
30     };
```

有些时候, 声明总线可能更方便完成对多个引脚进行交互操作, 此时你也可以声明引脚总线 SIMBUS。例如, LED 数码管的引脚比较多, 你可以声明 SIMPIN sega,segb,

segc, segd, sege, segf, segg……。为了一次性读取多个引脚，可以声明一个总线：SIMBUS seg\_bus（当然，后续还需要将引脚加入）

## 4.2.2 设置引脚（或总线）

声明完引脚（或总线）后，你还必须从系统中获取相应的接口（惟有如此，才能与远程硬件模块连接），只需要使用 GetSimPinByNumber 函数即可。源文件 Lamp.cpp 如下图所示，在 Setup 函数中（也可以在 Initialize 函数中）调用 GetSimPinByNumber 函数获取引脚编号为“1”的接口（成功获取则返加 TRUE），并将其与 input\_pin 对接即可）。如果你创建的元件的引脚存在唯一的名称，也可以使用 GetSimPinByName 函数，使用方法也是相似的。值得一提的是，**无论使用哪种方式，名称或编号中的字母都区分大小写。**

```
60     VOID CLamp::Setup(SIMDATA* sdat)
61     {
62         char tmp[10] = "1";                                //引脚编号
63         component->GetSimPinByNumber(input_pin, component->GetTCHARFromChar(tmp));
64     }
```

如果已经声明引脚总线，你也可以在 Setup 中对其进行初始化。同样以数码管为例，首先通过 GetSimPinByNumber 或 GetSimPinByName 初始化引脚，然后通过 AddToPinBus 函数将其添加到引脚总线，类似如下所示

```
component->AddPinToBus(seg_bus, sega);
component->AddPinToBus(seg_bus, segb);
component->AddPinToBus(seg_bus, segc);
...

```

需要注意的是，后续读取总线数据时，第一个添加的引脚对应总线数据的最低位。例如，获取到的数码管总线数据为 0x21，则表示 sega 与 segf 为高电平，其它为低电平。

### 4.2.3 获取或设置引脚（或总线）信号

当引脚（或总线）初始化完成后，就可以获取或设置远程引脚的电平数据，那么这些代码应该放在哪里呢？当然是在 Simulate 函数中。前面我们已经提过，Simulate 函数通过 GetProdata 函数逐条取出预置数据，如果你决定保留预置数据这项功能（也就是说，在“预置数据”窗口中预置的数据也还是能够解析），你可以将解析数据的代码放到 GetProdata 函数之前执行，当你解析完从远程硬件模块采集的数据之后，使用 AddToProData 函数将数据添加到预置数据列表即可。这样的话，如果元件本来带有预置数据，则 VisualCom 软件平台会先执行（相当于初始化元件），一旦预置数据全部执行完毕后，紧接着就会执行使用 AddToProdata 函数添加的预置数据。

如果你觉得预置数据很麻烦，也可以将处理预置数据的代码全部删除（新的元件模型中也可以不用实现），这样你就可以将解析出来的数据直接来控制元件的行为（仿真效果），只不过这样一来，预置数据就不会再有用了（**因为在元件模型中并没有解析，本质上，“预置的数据”与“采集的数据”是完全一致的，只是进入的途径不同而已**）。VisualCom 软件平台自带库中元件都使用第一种，这样即便你并没有远程硬件模块，也可以使用预置数据功能了解元件的实际效果。

我们来看看灯泡元件是如何采集数据的，相应的 Simulate 函数如下图所示。首先我们调用了 PinDataValid 函数判断采集数据是否有效，该函数返回 TRUE 表示采集的数据已经到来。之后通过调用 GetPinData 函数获取引脚的电平，再将其打包到 PRODATA 类型的变量 pro\_data\_tmp，最后调用 AddProData 函数将其“塞”到预置数据列表即可，简单吧！

```

66     □ BOOL CLamp::Simulate(ABSTIME time, RUNMODES mode)
67     {
68         PRODATA pro_data_tmp;
69
70         if (component->PinDataValid())                                //如果已经采集到数据
71         {
72             BYTE dat;
73             component->GetPinData(dat, input_pin);                  //获取该引脚的数据
74             pro_data_tmp.data = dat;
75             component->AddProData(pro_data_tmp);           //将该引脚数据添加到预置数据列表
76         }
77
78         if (component->GetProData(pro_data_tmp))                //获取预置数据
79         {
80             if (pro_data_tmp.data & 0x1)                           //判断预置数据最低位
81             {
82                 m_LampStatus = TRUE;                            //设置灯泡为点亮状态
83             }
84         }
85     }

```

当然，你还可以做得更好一点。例如，很有可能连续采集的数据是相同的，而你可以仅将变化的数据放到预置数据列表中，也就可以节省资源，这一点就不赘述了。

GetPinData 函数可以获取所有类型的采集数据，如果你只想获取某类数据，也可以调用 GetPinDataByType 函数，其原型如下所示，

```
BOOL GetPinDataByType(BYTE& dat,
                      SIMPIN& pin,
                      const SIMPIN& spin,
                      RDATATYPE type);
```

其中，最后一个参数表示类型，RDATATYPE 定义如下图所示。其中，RDT\_INVALID 表示无效数据，仿真开始时就初始化为该值。RDT\_OVERFLOW 表示采集数据已经溢出的数据，默认情况下该数据不会采集进来（除非使用 EnableOverflowData 函数开启）。 RDT\_TIMREAD 代表等时采样数据，RDT\_EVENTREAD 代表事件采样数据， RDT\_HOSTREAD 代表**主动读取的数据**（spin 形参也仅对此类数据采集有效，后述）， RDT\_CONTROLLER 为系统预留。

```

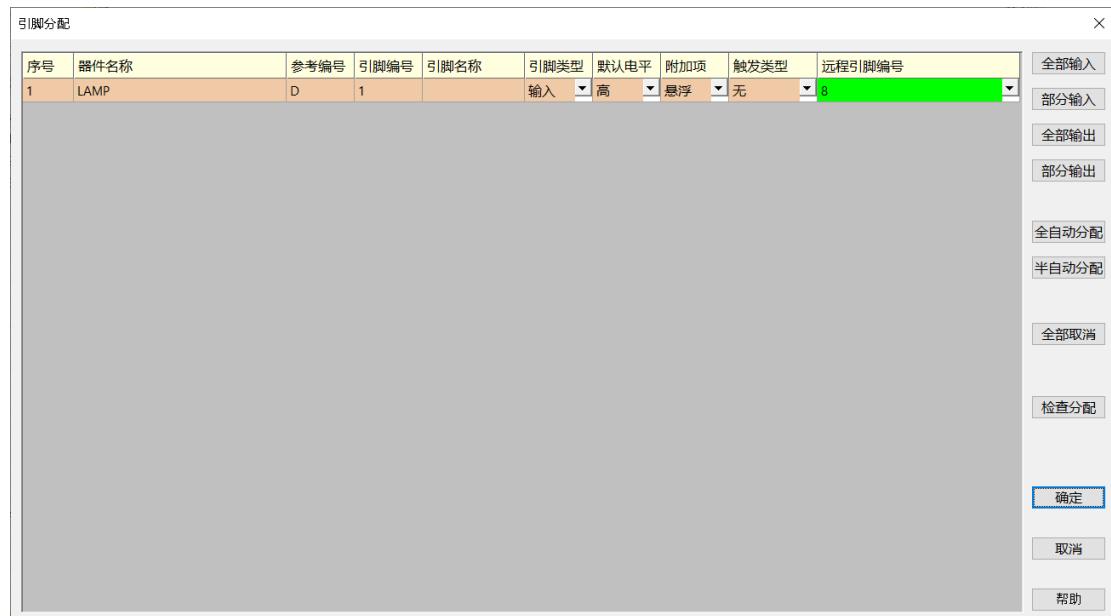
enum RDATATYPE
{
    RDT_INVALID = 0x0,
    RDT_OVERFLOW = 0x1,
    RDT_TIMREAD = 0x2,
    RDT_EVENTREAD = 0x4,
    RDT_HOSTREAD = 0x8,
    RDT_CONTROLLER = 0x10
};

```

当然，这些类型数据的获取也有单独的函数，分别对应为 GetOverflowPinData、GetNormalPinData（等时采样数据）、GetEventPinData、GetHostReadPinData，而引脚总线的数据读取也有相似的函数，具体参考第 5 章，此处不再赘述。

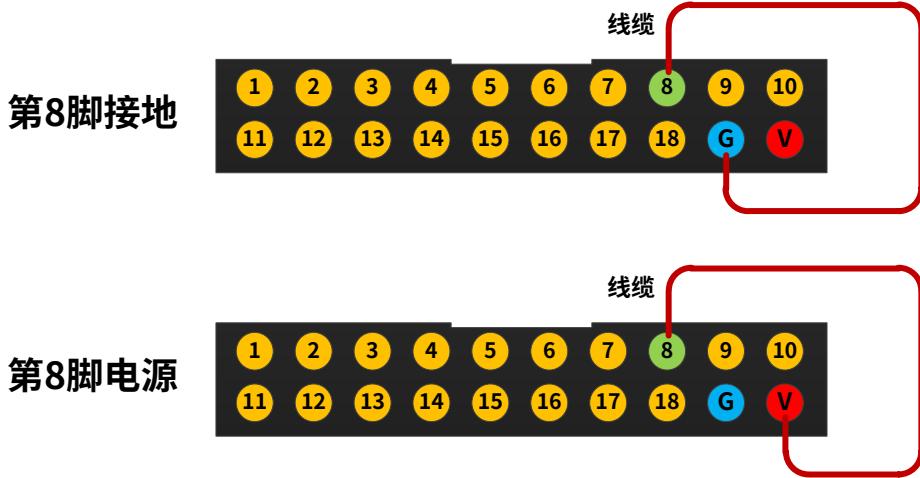
### 4.3 与硬件模块联合仿真

简单的灯泡元件模型已经开发完成，将其编译后放到 VisualCom 软件平台安装目录下的 model 文件夹中，然后进入“引脚分配”对话框将其引脚编号分配为“8”（这是自带案例分配的引脚编号，你也可以分配其它引脚编号），如下图所示。



如果远程硬件模块已经正确连接，当进入仿真状态后，该灯泡的状态将随远程模块中的

第 8 脚电平状态而变化，相应的接口连接示意如下图所示。如果使用配套的测试模块，也可以直接方便地按下相应引脚对应的按键即可，此处不再赘述。



## 4.4 更多细节

前面已经简要阐述单个引脚数据的读取过程，本节则对多个引脚电平数据读取、单个或多个引脚电平数据设置、修改电气类型、主动读取数据、获取默认的引脚分配状态、设置数据电平延时等细节进行讨论。

### 4.4.1 多个引脚数据读取

多个引脚数据的读取与单个引脚数据的读取相似，只需要将多个引脚添加到引脚总线中，再使用 GetPinBusData（或其它相似函数）读取即可，此处不再赘述。

### 4.4.2 输出数据

输出数据则使用 SetPinData（或 SetPinBusData），它们将直接改变硬件模块的引脚

状态。当然，还有一些直接设置引脚电平高低的函数，本质上是用 SetPinData 函数的包装，此处不再赘述。例如，SetPinDataHigh 与 SetPinDataLow 分别表示设置引脚电平为“高”与“低”。

#### 4.4.3 更改引脚的电气类型

如果有必要，你也可以实时更改远程模块的引脚电气类型。例如，系统自带的 LCD1602 测试接口（元件型号 IF6800C01/02）可以驱动与远程硬件模块连接的实际 LCD1602，也可以读取 LCD1602 的忙标记。在输出数据的时候，总线引脚是输出类型，而在读取数据时，总线引脚是输入类型，此时你可以使用 SetPinMode ( SetPinBusMode ) 函数进行切换即可，相应的函数原型如下图所示。

**VOID SetPinMode(SIMPIN& pin,  
RPINMODE mode);**

其中，RPINTTYPE 的定义如下图所示，对于 VC02 来说，有效的电气类型为输入悬浮 ( RPM\_INF )、输入下拉 ( RPM\_IPD )、输入上拉 ( RPM\_IPU )、推挽输出(RPM\_OUTPP)，开漏输出(RPM\_OUTOD)。

```
enum RPINMODE
{
    RPM_INF = 0,// Input Floating
    RPM_IPD = 1,// Input with Pull-Down Resistor
    RPM_IPU = 2,// Input with Pull-Up Resistor
    RPM_AIN = 3,// Input Analog
    RPM_OUTPP = 4,//Push-Pull Output
    RPM_OUTOD = 5,//Open-Drain Output
    RPM_AFPP = 6,//Alternate Function:Push-Pull Output
    RPM_AFOD = 7,//Alternate Function:Open-Drain Output
};
```

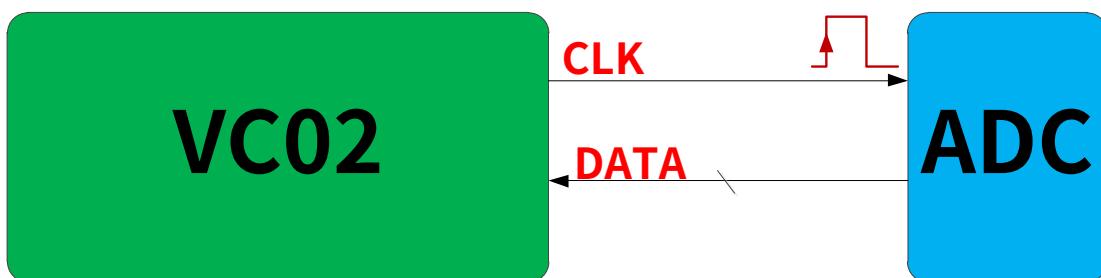
#### 4.4.4 主动读取数据

“主动读取数据”是什么意思呢？以前述 LCD1602 为例，怎么样读取 LCD1602 的忙标记呢？这种情况仅通过等时或事件类型数据是无法判断的，因为无法从海量数据中找到想要的那个数据。这种情况下，你应该使用 ReadPinData 函数主动读取数据，其原型如下图所示。

**VOID ReadPinData(SIMPIN pin);**

ReadPinData 函数需要传入一个引脚作为形参，与 GetPinData 函数不同，ReadPinData 函数会即时读取远程模块的引脚数据，如果后续在获取该数据时的引脚与你传入的相同，说明就是想要数据，所以在前述 GetHostReadPinData 函数中，你需要传入一个 spin 形参，它应该与给 ReadPinData 函数传入的引脚相同。为什么要这样做呢？当然还是为了区分主动读取数据的归属！假设一个元件中存在多个需要主动读取远程硬件模块的引脚数据，你如何确定哪个读取的数据是你想要的呢？只需要在读取时传入不同的 pin 即可，这个引脚通常是发起数据读取的引脚（非必须，仅要求唯一）。例如，对于 LCD1602，你可以将引脚 “E” 作为发起数据读取的引脚。

作为另一个完整的例子，假设远程硬件模块外面接了一个 ADC 数据采集芯片，它需要在时钟上升沿才会输出数据，如下图所示。



当需要读取这个芯片数据时，我们可以在元件模型中将时钟引脚拉高（假设默认电平为低，调用 SetPinDataHigh 即可），这样就产生了上升沿。接下来调用 ReadPinData 函数读

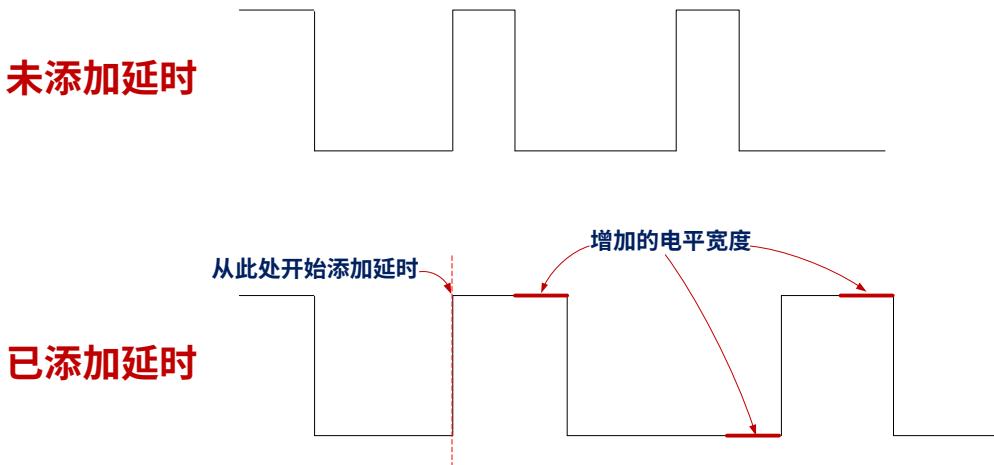
取数据（因为此时上升沿已经产生，也就可以读取数据），最后再将时钟引脚拉低（调用 SetPinDataLow 即可）即可。之后，ReadPinData 函数采集的数据就会回传到系统中，你就可以使用 GetHostReadPinData（或 GetHostReadPinBusData）函数获取了！

#### 4.4.5 获取引脚的分配状态

对于有些引脚比较多，但在实际使用时并不需要全部使用的元件，你也可以根据引脚的分配状态设置默认的状态，什么意思呢？假设某个元件有一个低电平复位的引脚，但是这个引脚不是必须的，你可以这样设计元件模型：**当该引脚没有分配时，将该引脚默认状态设置为高（即不复位状态）。**为实现这种功能，你可以使用 GetPinAssginMask（或 GetPinBusAssignMask），对于单个引脚而言，获得的值为 1 表示已分配，0 表示未分配，flag 表示是否考虑平台的设置。例如，VC02 的引脚编号范围为 1~18，而某个引脚却分配为 20，如果 flag 为 TRUE，则返回 0（表示未分配），如果 flag 为 FALSE，则返回 1（表示已分配）。

#### 4.4.6 设置数据电平延时

设置数据电平延时实际上就是在两个“设置电平指令”之间增加延时，你只需要调用 SetPinDataDelay 函数即可。举个例子，现在连续循环使用 SetPinDataLow 与 SetPinDataHigh 函数设置某个引脚电平，这也是使引脚电平翻转最快的方式（输出的不一定完全是绝对意义上的方波），如果在改变两个引脚电平之间调用 SetPinDataDelay 函数，也就能够加大每个电平的宽度，示意如下图所示：



#### 4.4.7 错误编程

在进行元件模型开发时，有两点需要特别注意，其一，**切勿使用“while 等语句等待某个数据是否符合预期”之类的编程方式，而应该改为使用标记的方式**，因为你必须处理完上一次数据(即完成一次 Simulate 函数调用)后才能进行下一个数据的处理，不然就会“死”在函数里面。

举个例子，现在要在往“与远程硬件模块连接的 LCD1602”发送指令或数据前需要读取并确认忙标记，你就不能使用类似下面的方式：

```
do {
    component->GetHostReadPinBusData(bus_dat, …);
} while(bus_dat&0x80);
```

而应该定义一个标记(此处为 BOOL 类型的 lcd\_busy，TRUE 表示忙，FALSE 表示空闲)，相应的编程示意如下所示：

```

If (lcd_busy)
{
    component->GetHostReadPinBusData(bus_dat, …);
    if (!(bus_dat&0x80))
    {
        lcd_busy = FALSE;
    }
}

If (!lcd_busy)
{
    //正常写入数据或指令的操作
}

```

其二，在判断边沿时应该注意逻辑的正确性。假设现在有一个设置为上升沿触发的时钟引脚 clk\_pin，那么使用下图所示方式判断上升沿是错误的，因为能够通过 GetEventPinData 函数成功获取到数据 dat 后，其值肯定是高电平“1”，那么再去判断“clk\_pin 为上升沿”的语句总是会返回 FALSE(因为上升沿的前一个数据必须是低电平)。

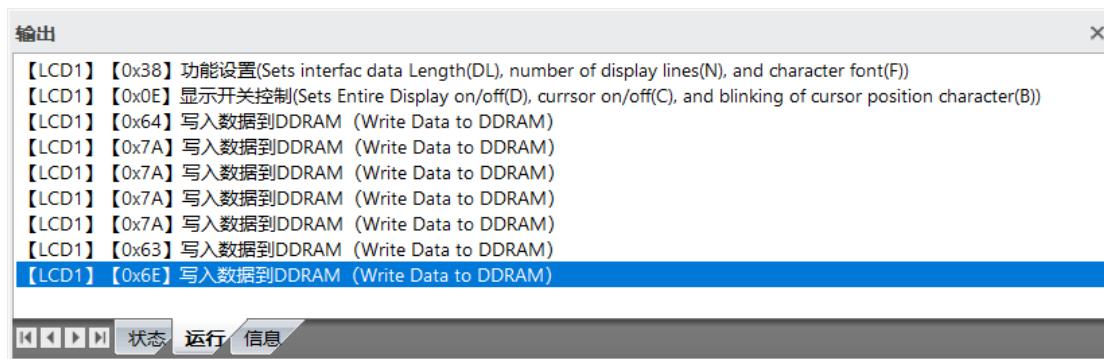
```

if (component->GetEventPinData(dat, clk_pin))
{
    if (component->IsPosEdge(clk_pin))
    {
        //上升沿时执行的行为
    }
}

```

#### 4.4.8 调试信息

调试信息对元件模型的开发至关重要，你可以调用函数 SetOutputInfo 与 SetOutputRun 分别往“输出”窗口的“信息”与“运行”标签中将信息输出，前者仅在单步运行仿真状态下有效，后者则在全步仿真运行状态下有效。以“基于 HD44780 的 LCM1602”(型号 LM016)为例，当进入全步仿真状态时，相应运行信息类似如下图所示。



如果你希望显示运行信息，**首先**需要将“仿真参数”对话框中的“运行信息”项设置为 True，这是针对当前设计中所有元件的运行信息显示控制总开关。由于运行信息的显示也需要占用一定的资源，因此默认情况下是关闭的。**其次**，每个元件也有一个“运行信息”开关，同样以 LM016 为例，其元件选项对话框类似如下图所示，只有进一步将“仿真”类别下“运行信息”设置为 True 才能显示运行信息。



# 第5章 API 函数

为最大程度简化仿真模型的开发，VisualCom 软件平台将所有接口函数归类到 **IDSIMMODEL** 与 **ICOMPONENT** 结构体，前者是你在进行开发时需要实现的接口，后者则是你可以进行操作（或使用的功能），随着版本的更新，**ICOMPONENT** 结构体中可能还会更新更多接口函数，本节主要阐述其中所有可供使用的函数。

## 5.1 **IDSIMMODEL**

此结构体中包含了所有你需要实现的接口，当你进行仿真模型开发时，需要创建一个以该结构体为基类的类，由于其中都是纯虚函数，所以你必须实现所有函数。当然，不少函数只是 VisualCom 软件平台预留（实际并不使用），暂时并不需要关注。

### 5.1.1 获取仿真模型

函数原型	<code>CDSIMMODEL* GetSimModel(TCHAR* device);</code>
作用	获取字符串对应的仿真模型
形参	<code>device</code> : 指定元件的字符串
返回值	仿真模型
注意事项	目前版本只需要返回当前仿真模型即可

### 5.1.2 获取元件指针

函数原型	<code>ICOMPONENT* GetComponentPtr();</code>
作用	获取指向元件的指针

形参	无
返回值	指向元件的指针
注意事项	目前版本只需要返回 Initialize 传入的指针即可

### 5.1.3 线程相关

有些器件可能需要周期性进行状态更新（例如，有些液晶模型的光标闪烁、实时时钟芯片的计数累加），这是系统通过周期性调用 IntervalProcess 来实现的，而调用的时间间隔则可以通过 GetTimerInterval 函数指定。

需要特别注意是，VisualCom 软件平台仅支持 1ms、10ms、100ms、1000ms 共 4 种时间间隔（默认为 1000ms），当你返回的值在 1、2~10、11~100、>100 时，系统调用 IntervalProcess 函数的对应时间间隔可能是 1ms、10ms、100ms、1000ms。之所以描述为可能，是因为你的原理图中可能存在多个需要指定不同时间间隔的元件，而系统在对所有元件调用 GetTimerInterval 函数获取时间间隔后，会将时间间隔设置为其中的最小值，所以当前元件返回的时间间隔可能并不是最终系统调用相应仿真模型中 IntervalProcess 函数的时间间隔，所以你需要使用 ICOMPONENT 结构体中的 GetThreadTimeInterval 函数确定最终的时间间隔。

举个例子，当前原理图中存在两个分别返回 10 与 100 的元件 A 与 B，当进入仿真状态后，系统会将时间间隔设置为 10ms，这也是你使用 ICOMPONENT 结构体中的 GetThreadTimeInterval 函数获得的值。对于元件 B 而言（对于元件 B 也同样如此），你需要先通过时间间隔计算调用次数（此处为  $100/10=10$ ），仿真模型中每 10 次调用 IntervalProcess 函数后才是真正执行周期性代码的时机。

函数原型	LONG GetTimeInterval(INT dat);
作用	获取仿真模型指定的时间
形参	预留
返回值	指定的时间
注意事项	勿在 Initialize 函数中调用此函数，因为此时系统尚未读取完所有元件需要的时 间间隔

函数原型	BOOL IntervalProcess(RUNMODES mode);
作用	系统在指定时间间隔循环调用的函数
形参	当前仿真模型，预留
返回值	返回 TRUE 表示立即刷新视图
注意事项	此刷新间隔独立于系统设置的刷新间隔

#### 5.1.4 初始化相关

进入仿真状态后，如果元件对应的仿真模型正确加载，系统首先会调用 Initialize 函数，并且会传入代表当前元件的指针以及当前的仿真状态。当所有仿真元件的 Initialize 函数被调用后，系统会调用 Setup 函数，其中提供第二次初始化的机会（与 Initialize 函数相同，每次进入仿真状态后，Setup 函数只会调用一次）。例如，ICOMPONENT 结构体中的 GetThreadTimeInterval 函数就是在 Setup 函数中调用的最佳时机。

函数原型	VOID Initialize(ICOMPONENT* cpt, DSIMMODES smode);
作用	初始化仿真模型

形参	cpt: 代表元件的指针 smode: 当前的仿真状态
返回值	无
注意事项	无

函数原型	VOID Setup(SIMDATA* sdat);
作用	初始化仿真模型
形参	预留
返回值	无
注意事项	无

## 5.1.5 解析预置数据

进入仿真状态后，如果元件存在预置数据（并且参与仿真），系统会调用 Simulate 函数逐条获取预置数据以进行数据解析。

函数原型	BOOL Simulate(ABSTIME time, RUNMODES mode);
作用	解析预置数据
形参	time: 绝对时间（预留） mode: 当前仿真模式
返回值	如果预置数据被有效处理，应该返回 TRUE，否则应返回 FALSE
注意事项	只有此函数返回 TRUE，“预置数据”窗口中的高亮项才会往下移动

## 5.1.6 动画

当预置数据被解析后，视图可能需要进行更新，你可以在 Animate 函数中进行必要的绘制（只是在缓存中绘图，是否在屏幕上绘图取决于 Plot 函数）。也就是说，Animate 函数不会被系统直接调用，所以你也可以自定义一个函数进行图形绘制。

函数原型	VOID Animate(INT element, ACTIVEDATA* data);
作用	绘制图形
形参	预留
返回值	无
注意事项	无

## 5.1.7 绘制图形

系统在指定的“刷新时间”都会调用 Plot 函数，只有当其返回 TRUE 时，元件的视图状态才会被更新。

函数原型	BOOL Plot(ACTIVESTATE state);
作用	绘制图形
形参	预留
返回值	无
注意事项	无

## 5.1.8 按键或鼠标交互

如果你需要使用鼠标或按键控制元件的仿真状态，可以通过 Actuate 函数来实现。

函数原型	BOOL Actuate(WORD key, DPOINT p, UINT flags);
作用	按键或鼠标交互
形参	key: 按键码（非按键事件时为 0） p: 鼠标单击的位置（以元件右下角为参考，非鼠标事件时为（0, 0）） flags: 标记
返回值	预留
注意事项	无

## 5.1.9 其它

系统预留的其它函数

函数原型	VOID CallBack(ABSTIME time, EVENTID eventid);
作用	预留
形参	预留
返回值	预留
注意事项	无

函数原型	BOOL Indicate(REALTIME time, ACTIVEDATA* data);
作用	预留

形参	预留
返回值	预留
注意事项	无

## 5.2 ICOMPONENT

ICOMPONENT 结构体包含所有可供操作的接口函数，主要分为获取图形信息、获取自定义属性信息、获取系统信息、获取或设置元件状态信息、绘制图形、设置图片、控制声音播放、字符串格式转换与比较、与远程模块交互、缓存控制几大部分，详细描述如下

### 5.2.1 获取图形信息

获取图形信息是为了更方便准确在想要位置绘制相应的图形。例如，将图形绘制到元件时，默认的参考点在元件的右下角。当你想往元件左上角绘制图形时，可以通过 GetSymbolOffset 函数获得图形相对于元件的偏移，再将该偏移值随图形绘制即可得到相应的效果，而获取的偏移值也可作为后续图形绘制的参考。

函数原型	DPOINT GetPosRightBottom();
作用	获取元件右下角坐标（以整个原理图文件左上角为参考）
形参	无
返回值	元件右下角的坐标（逻辑坐标）
注意事项	无

函数原型	DPOINT GetPosLeftTop();
------	-------------------------

作用	获取元件左上角坐标（以整个原理图文件左上角为参考）
形参	无
返回值	元件左上角的坐标（逻辑坐标）
注意事项	无

函数原型	DPOINT GetSize();
作用	获取元件的尺寸
形参	无
返回值	元件的尺寸
注意事项	x 表示宽度，y 表示高度，(逻辑尺寸)

函数原型	BOOL GetSymbolSize(INT index, DPOINT& p);
作用	获取图形的尺寸
形参	index: 图形索引 p: 接收尺寸的变量
返回值	获取尺寸成功则返回 TRUE，否则返回 FALSE
注意事项	无

函数原型	BOOL GetLastSymbolSize(DPOINT& p);
作用	获取上一次绘制的图形的尺寸
形参	无

返回值	获取尺寸成功则返回 TRUE, 否则返回 FALSE
注意事项	无

函数原型	DPOINT GetLastSymbolOffset();
作用	获取上次绘制的图形相对于左上角的偏移量
形参	无
返回值	获取失败则返回 ( 0, 0 )
注意事项	无

函数原型	VOID GetComponentPos(DPOINT& a, DPOINT& b);
作用	获取元件的左上与右下的坐标点
形参	a,b: 接收坐标的两个变量
返回值	无
注意事项	无

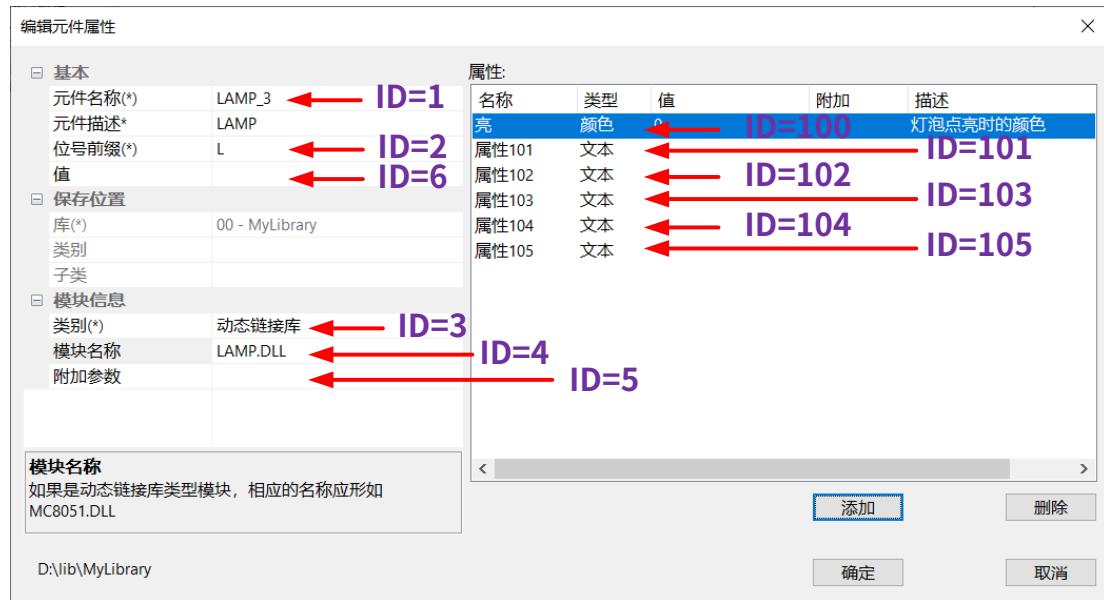
函数原型	BOOL GetLastSymbolArea( DPOINT& a, DPOINT& b, BOOL flag);
作用	获取上次绘制图形的左上与右下坐标点
形参	a,b: 接收坐标的两个变量 flag: 是否加上偏移值 ( TRUE 表示添加 )
返回值	成功则返回 TRUE, 失败则返回 FALSE

注意事项	无
------	---

函数原型	BOOL InsideComponent(DPOINT p);
作用	判断坐标是否在元件范围内
形参	p: 从元件右下角为参考的坐标
返回值	在元件范围内则返回 TRUE, 否则返回 FALSE
注意事项	无

## 5.2.2 获取用户自定义属性信息

获取自定义属性信息的方式主要有两种，其一，通过属性名称。如果期望获得正确的属性项信息，请不要设置名称相同的属性，否则你会获得第一个符合要求的属性项信息。其二，通过属性 ID（**推荐**）。在“创建（编辑）元件属性”的属性列表中，属性项的 ID 均从 100 开始往下依次增加（0~99 为系统预留）。以下图为例，ID 值为 1~6 对应系统项，“亮”属性项的属性名称为“亮”，而其 ID 为“100”，“属性 101”项的 ID 则为 101，其它依此类推（更改属性名称不影响 ID）。



函数原型	TCHAR* GetFieldByName(TCHAR* name);
作用	通过属性名称获取相应属性中的字符串
形参	name: 自定义的属性名称
返回值	属性项中的字符串
注意事项	未找到给定名称对应的属性项则返回空

函数原型	TCHAR* GetFieldById(INT id);
作用	通过属性的 ID 获取相应属性中的字符串
形参	id: 自定义的属性的 ID
返回值	属性项中的字符串
注意事项	未找到给定 ID 对应的属性项则返回空

函数原型	BOOL GetLongFieldByName(TCHAR* name, LONG& dat);
------	--

<b>作用</b>	通过属性的名称获取“整数”类型属性项中的长整型值
<b>形参</b>	<p>name: 给定的属性名称</p> <p>dat: 接收长整型值的变量</p>
<b>返回值</b>	如果找到给定名称对应的属性项，且属性的类型为“整数”，则返回 TRUE，否则返回 FALSE
<b>注意事项</b>	如果找到的自定义属性的类型不为“整数”，该函数将返回 FALSE

<b>函数原型</b>	<code>GetLongFieldById(INT id, LONG&amp; dat);</code>
<b>作用</b>	通过属性的 ID 获取“整数”类型属性项中的长整型值
<b>形参</b>	<p>id: 给定的属性 ID</p> <p>dat: 接收长整型值的变量</p>
<b>返回值</b>	如果找到给定 ID 对应的属性项，且属性的类型为“整数”，则返回 TRUE，否则返回 FALSE
<b>注意事项</b>	如果找到的自定义属性的类型不为“整数”，该函数将返回 FALSE

<b>函数原型</b>	<code>GetBoolFieldByName(TCHAR* name, BOOL&amp; dat);</code>
<b>作用</b>	通过属性的名称获取“布尔”类型属性项中的布尔值
<b>形参</b>	<p>name: 给定的属性名称</p> <p>dat: 接收布尔值的变量</p>
<b>返回值</b>	如果找到给定名称对应的属性项，且属性的类型为“布尔”，则返回 TRUE，否则返回 FALSE

	则返回 FALSE
注意事项	如果找到的自定义属性的类型不为“布尔”，该函数将返回 FALSE

函数原型	<code>GetBoolFieldById(INT id, BOOL&amp; dat);</code>
作用	通过属性的 ID 获取“布尔”类型属性项中的布尔值
形参	<p><code>id</code>: 给定的属性 ID</p> <p><code>dat</code>: 接收布尔值的变量</p>
返回值	如果找到给定 ID 对应的属性项，且属性的类型为“布尔”，则返回 TRUE，否则返回 FALSE
注意事项	如果找到的自定义属性的类型不为“布尔”，该函数将返回 FALSE

函数原型	<code>GetColorFieldByName(TCHAR* name, COLORREF&amp; color);</code>
作用	通过属性的名称获取“颜色”类型属性项中的颜色值
形参	<p><code>name</code>: 给定的属性名称</p> <p><code>color</code>: 接收颜色值的变量</p>
返回值	如果找到给定名称对应的属性项，且属性的类型为“颜色”，则返回 TRUE，否则返回 FALSE
注意事项	如果找到的自定义属性的类型不为“颜色”，该函数将返回 FALSE

函数原型	<code>GetColorFieldById(INT id, COLORREF&amp; color);</code>
作用	通过属性的 ID 获取“颜色”类型属性项中的颜色值
形参	<p><code>id</code>: 给定的属性 ID</p> <p><code>color</code>: 接收颜色值的变量</p>
返回值	如果找到给定 ID 对应的属性项，且属性的类型为“颜色”，则返回 TRUE，否则返回 FALSE
注意事项	如果找到的自定义属性的类型不为“颜色”，该函数将返回 FALSE

函数原型	<code>TCHAR* GetComponentReference();</code>
作用	获取元件的参考编号
形参	无
返回值	元件的参考编号
注意事项	无

函数原型	<code>TCHAR* GetComponentName();</code>
作用	获取元件的名称
形参	无
返回值	元件的参考名称
注意事项	无

以下为获取字段相关信息，通常情况下使用不到。字段索引与 ID 存在的意义不相同，

当你在创建元件时，一些输入的信息都会保存在链表中。例如，链表的前 6 个单元保存的值依次为“元件名称”、“元件参考编号”、“仿真类型”、“仿真模型文件名”、“额外参数”、“元件值”，它们的索引与 ID 值是相同的，但是从链表第 7 个单元开始(后续可能会更新)，保存的是自定义属性项，所以字段索引为 6 对应的 ID 为 100。

函数原型	INT GetFieldIndexByName(TCHAR* name);
作用	获取名称相应的字段索引
形参	name: 字段的名称
返回值	字段的索引
注意事项	无

函数原型	INT GetFieldIndexById(INT id);
作用	获取 ID 相应的字段索引
形参	id: 字段的 ID
返回值	字段的索引
注意事项	无

函数原型	INT GetFieldCount() ;
作用	获取字段的个数
形参	无
返回值	字段的个数

## 注意事项

无

### 5.2.3 获取系统信息

有时候，你可能需要获取系统信息以完成一定的功能，前面介绍过的系统调用 GetThreadTimeInterval 函数的时间间隔就是如此。当然，有时候，你也可以获取主程序的绝对完整路径(以便进行文件的读写)，这也可以使用 GetAppDirectory 函数实现。例如，“基于 ST7920 的液晶显示模组”内置的 GB2312 编码字库就需要读取位于系统安装目录下的 font 文件夹中字库文件。

函数原型	TCHAR* GetAppDirectory(TCHAR* subfolder, BOOL reverse_flag);
作用	获取主程序或下一级目录的全路径
形参	subfolder: 下一级目录, reverse_flag: 是否将字符 “\” 替换为 “/”
返回值	给定参数的目录的路径
注意事项	如果 subfolder 为空，则返回主程序所在的路径

函数原型	LONG GetThreadTimeInterval();
作用	系统通过该函数返回仿真模型设置的时间间隔
形参	无
返回值	时间间隔
注意事项	系统仅支持 1ms、10ms、100ms、1000ms 共 4 种时间间隔( 默认为 1000ms )，

当你返回的值在 1、2~10、11~100、>100 时，系统调用 IntervalProcess 函数的对应时间间隔可能是 1ms、10ms、100ms、1000ms。之所以描述为可能，是因为你的设计中可能存在多个需要指定不同时间间隔的元件，而系统在对所有元件调用 GetTimerInterval 函数获取时间间隔后，会将系统时间间隔设置为其中的最小值，所以当前元件返回的时间间隔可能并不是最终系统调用 IntervalProcess 函数的时间间隔，所以你需要使用 ICOMPONENT 结构体中的 GetThreadTimeInterval 函数确定最终的时间间隔。

举个例子，当前原理图中存在两个分别返回 10 与 100 的元件 A 与 B，当进入仿真状态后，系统会将时间间隔设置为 10ms，这也是你使用 ICOMPONENT 结构体中的 GetThreadTimeInterval 函数获得的值。对于元件 B 而言（对于元件 B 也同样如此），你需要先获得时间间隔的次数（即  $100/10=10$ ），仿真模型中每 10 次调用 IntervalProcess 函数后才是真正执行周期性代码的时机。

#### 5.2.4 获取或设置元件状态信息

系统在进入仿真状态后需要获取预置数据进行解析，在解析过程中也可能会产生一些临时寄存器或内存的状态信息，你可以根据开发需要将这些信息反馈给用户（主要放在“寄存器”、“内存”、“指令集”、“输出”窗口），或接收用户的输入的信息来改变仿真模型的状态。

函数原型	BOOL GetProData(CPRODATA& pro_data, BOOL backup = TRUE);
作用	获取一条预置数据
形参	pro_data: 接收预置数据的变量 backup: 是否将取出的数据进行备份（预留）
返回值	获得有效预置数据返回 TRUE，否则返回 FALSE

注意事项	备份功能只是预留，预置数据保存在链表中，一般情况下，取出一条预置数据后就会将其删除，将其备份可另作他用
------	---

函数原型	<code>BOOL GetUserStatus(PROPCELL&amp; status, INT mode);</code>
作用	获取用户通过“寄存器”或“内存”窗口修改的数据（以直接修改仿真模型的状态）
形参	status: 接收修改数据的变量 mode: 当前仿真模式
返回值	获得有效修改数据返回 TRUE，否则返回 FALSE
注意事项	无

函数原型	<code>VOID SetRegMemStatus( PROPCELLS* reg, PROPCELLS* mem, PROPCELLS* cmd);</code>
作用	将元件的当前状态放到“寄存器”、“内存”、“指令集”窗口中（为用户反馈当前状态）
形参	reg: 指向“寄存器”窗口数据的指针 mem: 指向“内存”窗口数据的指针 cmd: 指向“指令集”窗口数据的指针
返回值	无
注意事项	如果不将当前状态放到“寄存器”、“内存”、“指令集”窗口中，给形参赋 NULL 即可

函数原型	VOID SetOutputInfo(TCHAR* str);
作用	将元件的预置数据解析结果放到“输出”窗口的“信息”标签中（为用户反馈预置数据的解析结果）
形参	str: 指向字符串的指针
返回值	无
注意事项	无

函数原型	VOID SetOutputRun(TCHAR* str);
作用	将元件的预置数据解析结果放到“输出”窗口的“运行”标签中（为用户反馈预置数据的解析结果）
形参	str: 指向字符串的指针
返回值	无
注意事项	无

## 5.2.5 绘制图形

往元件中绘制图形的方式主要存在两种，其一，你需要创建一些图形（与灯泡元件相同），然后通过对称的图形索引绘制即可，绝大多数元件都可以使用这种方式（尤其是形状不规则的图形，Display 库中的所有元件都是使用此种方式），这种方式存在多个重载函数，函数名称均为 DrawSymbol。其二，通过设备上下文（Device Context）绘制常用的矩形、圆、文本等方式，这种方式需要你进行画笔（Pen）、画刷（Brush）、字体（Font）的设置（画笔通常用来绘制线条，画刷则用来设置填充参数），相关的绘图函数均以 Draw 开头，包括

DrawRectangle、DrawCircle、DrawLine、DrawPolygon、DrawText。

以下为第一种绘制图形方式相关的函数

函数原型	VOID DrawSymbol(INT index, LINESTYLE* ls, FILLSTYLE* fs, FONTSTYLE* ts, double dx = 0, double dy = 0);
作用	将图形索引对应的图形绘制到元件中
形参	index: 图形索引 ls: 边框样式 ( 使用原来的样式则传入 NULL 即可 ) fs: 填充样式 ( 使用原来的样式则传入 NULL 即可 ) ts: 文字样式 ( 使用原来的样式则传入 NULL 即可 ) dx: 图形相对于元件左上角的 X 轴偏移量 dy: 图形相对于元件左上角的 Y 轴偏移量
返回值	无
注意事项	无

函数原型	VOID DrawSymbol( INT index, LINESTYLE* ls, double dx = 0, double dy = 0);
作用	将图形索引对应的图形绘制到元件中 ( 仅允许改变边框样式 )
形参	index: 图形索引 ls: 边框样式 ( 使用原来的样式则传入 NULL 即可 ) dx: 图形相对于元件左上角的 X 轴偏移量

	dy: 图形相对于元件左上角的 Y 轴偏移量
返回值	无
注意事项	无

函数原型	<pre>VOID DrawSymbol(     INT index, FILLSTYLE* fs, double dx = 0, double dy = 0);</pre>
作用	将图形索引对应的图形绘制到元件中（仅允许改变填充样式）
形参	index: 图形索引 fs: 填充样式（使用原来的样式则传入 NULL 即可） dx: 图形相对于元件左上角的 X 轴偏移量 dy: 图形相对于元件左上角的 Y 轴偏移量
返回值	无
注意事项	无

函数原型	<pre>VOID DrawSymbol(     INT index, FONTSTYLE* ts, double dx = 0, double dy = 0);</pre>
作用	将图形索引对应的图形绘制到元件中（仅允许改变文字样式）
形参	index: 图形索引 ts: 文字样式（使用原来的样式则传入 NULL 即可） dx: 图形相对于元件左上角的 X 轴偏移量 dy: 图形相对于元件左上角的 Y 轴偏移量

返回值	无
注意事项	无

函数原型	VOID DrawSymbol(INT index, double dx = 0, double dy = 0);
作用	将图形索引对应的图形绘制到元件中（不改变样式）
形参	index: 图形索引 dx: 图形相对于元件左上角的 X 轴偏移量 dy: 图形相对于元件左上角的 Y 轴偏移量
返回值	无
注意事项	无

以下为第二种绘制图形方式中设置画笔、画刷相关的函数

函数原型	VOID SetPenColor(COLORREF col);
作用	设置画笔的颜色，也就是画出的线条的颜色
形参	col: 颜色值（RGB888 模式）
返回值	无
注意事项	无

函数原型	VOID SetPenWidth(INT width);
作用	设置画笔的宽度，也就是画出的线条的粗细程度
形参	width: 宽度

返回值	无
注意事项	无

函数原型	VOID SetPenStyle(INT style);
作用	设置画笔的样式
形参	<p>style: 样式的编号, 可取值如下</p> <p>PS_SOLID ( 实心 )</p> <p>PS_DASH ( 虚线 )</p> <p>PS_DOT ( 点线 )</p> <p>PS_DASHDOT ( 虚点线 )</p> <p>PS_DASHDOTDOT ( 虚点点线 )</p>
返回值	无
注意事项	无

函数原型	VOID SetBrushColor(COLORREF col);
作用	设置画刷的颜色, 也就是填充颜色
形参	col: 颜色值 ( RGB888 模式 )
返回值	无
注意事项	无

函数原型	VOID SetBrushStyle(INT style);
------	--------------------------------

作用	设置画刷的颜色，也就是填充颜色
形参	<p>style: 样式的编号，可取值如下</p> <p>HS_SOLID ( 实心 )</p> <p>HS_HORIZONTAL ( 垂直 )</p> <p>HS_VERTICAL ( 水平 )</p> <p>HS_FDIAGONAL ( 左上 )</p> <p>HS_BDIAGONAL ( 右上 )</p> <p>HS_CROSS ( 正交 )</p> <p>HS_DIAGCROSS ( 斜交 )</p>
返回值	无
注意事项	无

以下为第二种绘制图形方式中设置字体相关的函数

函数原型	VOID SetTextStyle(LOGFONT lf);
作用	设置字体格式 ( LOGFONT 结构体中包含了下面涉及的字体 )
形参	代表字体的变量
返回值	无
注意事项	无

函数原型	VOID SetTextFont(TCHAR* font);
作用	设置字体
形参	代表字体的名称。例如，“楷体_GB2312”

返回值	无
注意事项	无

函数原型	VOID SetTextSize(INT size);
作用	设置字体大小
形参	size: 字体大小值
返回值	无
注意事项	本质上是设置字体的高度

函数原型	VOID SetTextBold(BOOL flag);
作用	设置字体加粗与否
形参	flag: 加粗应传入 TRUE, 否则传入 FALSE
返回值	无
注意事项	无

函数原型	VOID SetTextItalic(BOOL flag);
作用	设置字体倾斜与否
形参	flag: 倾斜应传入 TRUE, 否则传入 FALSE
返回值	无
注意事项	无

函数原型	<code>VOID SetUnderline(BOOL flag);</code>
作用	设置字体添加下划线与否
形参	<code>flag</code> : 添加下划线应传入 TRUE, 否则传入 FALSE
返回值	无
注意事项	无

函数原型	<code>VOID SetTextHeight(LONG height);</code>
作用	设置字体的高度
形参	<code>height</code> : 高度值
返回值	无
注意事项	无

函数原型	<code>SetTextWidth(LONG width);</code>
作用	设置字体的宽度
形参	<code>width</code> : 宽度值
返回值	无
注意事项	无

以下为第二种绘制图形方式中绘制函数

函数原型	<code>VOID DrawRectangle(DOUBLE x1, DOUBLE y1, DOUBLE x2, DOUBLE y2, DOUBLE dx, DOUBLE dy);</code>
------	--

作用	绘制矩形
形参	x1: 图形的左上角 X 轴坐标 y1: 图形的左上角 Y 轴坐标 x2: 图形的右下角 X 轴坐标 y2: 图形的右下角 Y 轴坐标 dx: 相对元件右下角坐标的 X 轴偏移量 dy: 相对元件右下角坐标的 Y 轴偏移量
返回值	无
相似函数原型	VOID DrawRectangle(DRECT rect, DPOINT offset);
注意事项	图形坐标值以元件右下角坐标为参考

函数原型	VOID DrawCircle(DOUBLE x1, DOUBLE y1, DOUBLE x2, DOUBLE y2, DOUBLE dx, DOUBLE dy);
作用	绘制圆 ( 或椭圆 )
形参	x1: 图形的左上角 X 轴坐标 y1: 图形的左上角 Y 轴坐标 x2: 图形的右下角 X 轴坐标 y2: 图形的右下角 Y 轴坐标 dx: 相对元件右下角坐标的 X 轴偏移量 dy: 相对元件右下角坐标的 Y 轴偏移量
返回值	无

相似函数原型	<code>VOID DrawCircle(DRECT rect, DPOINT offset);</code>
注意事项	图形坐标值以元件右下角坐标为参考

函数原型	<code>VOID DrawLine(DOUBLE x1, DOUBLE y1, DOUBLE x2, DOUBLE y2, DOUBLE dx, DOUBLE dy);</code>
作用	绘制线条
形参	<p>x1: 第一个点的 X 轴坐标</p> <p>y1: 第一个点的 Y 轴坐标</p> <p>x2: 第二个点的 X 轴坐标</p> <p>y2: 第二个点的 Y 轴坐标</p> <p>dx: 相对元件右下角坐标的 X 轴偏移量</p> <p>dy: 相对元件右下角坐标的 Y 轴偏移量</p>
返回值	无
相似函数原型	<code>VOID DrawLine(DRECT rect, DPOINT offset);</code>
注意事项	图形坐标值以元件右下角坐标为参考

函数原型	<code>VOID DrawPolygon(DPOINTS pc, DPOINT offset);</code>
作用	绘制多边形
形参	<p>pc: 多个点的集合 ( 向量 )</p> <p>dx: 相对元件右下角坐标的 X 轴偏移量</p> <p>dy: 相对元件右下角坐标的 Y 轴偏移量</p>

返回值	无
相似函数原型	无
注意事项	图形坐标值以元件右下角坐标为参考

函数原型	VOID DrawText(DOUBLE x1, DOUBLE y1, DOUBLE dx, DOUBLE dy, TCHAR* str, INT dir = 1, INT flag = 0);
作用	绘制文本
形参	x1: 文本位置的 X 轴坐标  y1: 文本位置的 Y 轴坐标  dx: 相对元件右下角坐标的 X 轴偏移量  dy: 相对元件右下角坐标的 Y 轴偏移量  dir: 1 表示垂直, 3 表示水平
返回值	无
相似函数原型	无
注意事项	无

## 5.2.6 设置图片与读取像素颜色

有些元件（例如液晶显示模组）允许设置一张位图（\*.bmp）进行初始化，为简化用户开发相应的仿真模型，每个元件都包含了一个 Image 对象，你可以在创建元件时添加一个“文件”类型的属性项，由用户选择相应的图片路径，而你只需要将属性项的名称或 ID 设置完毕即可完成 Image 对象的初始化，之后就可以方便读取 X 与 Y 坐标的颜色。

函数原型	<code>BOOL SetBitmapByName(TCHAR* name);</code>
作用	根据指定属性名称对应属性项中的图片路径初始化 Image 对象
形参	<code>name:</code> 属性项名称
返回值	如果 Image 对象初始化成功则返回 TRUE, 否则返回 FALSE
相似函数原型	<code>BOOL SetBitmapById(INT id);</code>
注意事项	无

函数原型	<code>COLORREF GetBitmapPixel(INT x, INT y);</code>
作用	获取指定 X 与 Y 坐标对应的像素颜色
形参	<code>x:</code> 水平坐标 <code>y:</code> 垂直坐标
返回值	返回指定坐标的颜色值, 如果坐标超出范围则返回黑色 ( 0 )
相似函数原型	无
注意事项	无

### 5.2.7 控制声音播放

有些元件 ( 例如蜂鸣器 ) 需要一些声音文件, 相应也有一些函数可供使用, 同样你需要在创建元件时添加一个 “文件” 类型的属性项。

函数原型	<code>BOOL CyclePlaySoundByName(TCHAR* name);</code>
作用	循环播放指定属性名称对应的属性项中指定的*.wav 文件

形参	属性项的名称
返回值	-
相似函数原型	BOOL CyclePlaySoundById(INT id);
注意事项	无

函数原型	BOOL StopPlayAllSound();
作用	停止播放所有*.wav 文件
形参	无
返回值	-
相似函数原型	无
注意事项	无

## 5.2.8 字符串格式转换与比较

有些元件(例如蜂鸣器)需要一些声音文件,相应也有一些函数可供使用,同样你需要在创建元件时添加一个“文件”类型的属性项。

函数原型	TCHAR* GetTCHARFromChar(CHAR* str);
作用	将 char 类型字符串转换为 TCHAR 类型字符串
形参	str: char 类型字符串
返回值	TCHAR 类型字符串
相似函数原型	CHAR* GetCharFromTCHAR(TCHAR* str)

注意事项	无
------	---

函数原型	BOOL TCHARCompare(TCHAR* str1, TCHAR* str2);
作用	比较两个 TCHAR 类型字符串是否相同（区分大小写）
形参	str1,str2: TCHAR 类型字符串
返回值	相同返回 TRUE, 否则返回 FALSE
相似函数原型	无
注意事项	无

## 5.2.9 与远程模块交互

此部分函数主要与远程模块交互相关，主要包括设置引脚（或总线）、获取/设置/判断引脚（或总线）电平、电平延时、获取默认电气类型/电平等。

函数原型	BOOL GetSimPinByNumber(SIMPIN& p, TCHAR* num);
作用	通过引脚编号获取相应的引脚接口
形参	p: 需要获取的接口对应的引脚 num: 给定的引脚编号
返回值	如果获取引脚接口成功则返回 TRUE, 否则返回 FALSE
相似函数原型	BOOL GetSimPinByName(SIMPIN& p, TCHAR* name);
注意事项	无

函数原型	VOID AddPinToBus(SIMBUS& bus, SIMPIN& pin) ;
------	--

作用	将引脚添加到总线
形参	bus: 引脚添加到的总线 pin: 将要添加到总线的引脚
返回值	无
注意事项	无

函数原型	BOOL IsLogicHigh(const SIMPIN& pin);
作用	判断引脚电平是否为逻辑 “1”
形参	pin: 需要判断电平对应的引脚
返回值	如果引脚电平为高则返回 TRUE, 否则返回 FALSE
相似函数原型	BOOL IsLogicLow(const SIMPIN& pin) ;
注意事项	无

函数原型	BOOL IsPosEdge(const SIMPIN& pin);
作用	判断引脚电平是否对应为上升沿
形参	pin: 需要判断电平对应的引脚
返回值	如果为上升沿则返回 TRUE, 否则返回 FALSE
相似函数原型	BOOL IsNegEdge(const SIMPIN& pin); //下降沿 BOOL IsEdge(const SIMPIN& pin) ;//双边沿
注意事项	无

函数原型	<code>BOOL PinDataValid();</code>
作用	判断采集的数据是否有效
形参	无
返回值	如果采集数据有效则返回 TRUE, 否则返回 FALSE
注意事项	无

函数原型	<code>VOID EnableOverflowData();</code>
作用	是否将溢出数据当作采集的数据
形参	无
返回值	无
注意事项	默认情况下不处理溢出数据

函数原型	<code>BOOL GetPinData(BYTE&amp; dat, SIMPIN&amp; pin);</code>
作用	获取引脚对应的数据 ( 不区分采集数据类型 )
形参	<code>dat</code> : 返回的引脚电平数据, 仅 “1” 与 “0” 有效 <code>pin</code> : 需要获取数据的引脚
返回值	如果成功获取引脚电平数据则返回 TRUE, 否则返回 FALSE
相似函数原型	<code>BOOL GetPinBusData(ULONGLONG&amp; dat, SIMBUS&amp; bus);</code>
注意事项	无

函数原型	<code>BOOL GetPinDataByType(BYTE&amp; dat, SIMPIN&amp; pin,</code>
------	--

	<code>const SIMPIN&amp; spin, RDATATYPE type)</code>
<b>作用</b>	根据给定的数据类型标记获取引脚的数据
<b>形参</b>	<p>dat: 返回的引脚电平数据, 仅 “1” 与 “0” 有效</p> <p>pin: 需要获取数据的引脚</p> <p>spin: 与调用 ReadPinData 函数时传入的相同引脚</p> <p>type: 类型标记。RDT_INVALID ( 无效数据 ) RDT_OVERFLOW ( 溢出数据 ) RDT_TIMREAD ( 等时数据 ) RDT_EVENTREAD ( 事件数据 ) RDT_HOSTREAD ( 主动读取数据 ) RDT_CONTROLLER ( 控制器数据 )</p>
<b>返回值</b>	如果成功获取引脚电平数据则返回 TRUE, 否则返回 FALSE
<b>相似函数原型</b>	<code>BOOL GetPinBusDataByType(ULLONG&amp; dat, SIMBUS&amp; bus, const SIMPIN&amp; spin, RDATATYPE type);</code>
<b>注意事项</b>	无

<b>函数原型</b>	<code>BOOL GetNormalPinData(BYTE&amp; dat, SIMPIN&amp; pin);</code>
<b>作用</b>	获取引脚采集的等时数据
<b>形参</b>	<p>dat: 返回的引脚电平数据, 仅 “1” 与 “0” 有效</p> <p>pin: 需要获取数据的引脚</p>
<b>返回值</b>	如果成功获取引脚电平数据则返回 TRUE, 否则返回 FALSE
<b>相似函数原型</b>	<code>BOOL GetNormalPinBusData(ULLONG&amp; dat, SIMBUS&amp; bus) ;</code>
<b>注意事项</b>	无

函数原型	<code>BOOL GetEventPinData(BYTE&amp; dat, SIMPIN&amp; pin);</code>
作用	获取引脚采集的事件数据
形参	<p>dat: 返回的引脚电平数据, 仅 “1” 与 “0” 有效</p> <p>pin: 需要获取数据的引脚</p>
返回值	如果成功获取引脚电平数据则返回 TRUE, 否则返回 FALSE
相似函数原型	<code>BOOL GetEventPinBusData(ULLONG&amp; dat, SIMBUS&amp; bus) ;</code>
注意事项	无

函数原型	<code>BOOL GetOverflowPinData(BYTE&amp; dat, SIMPIN&amp; pin) ;</code>
作用	获取引脚采集的溢出数据
形参	<p>dat: 返回的引脚电平数据, 仅 “1” 与 “0” 有效</p> <p>pin: 需要获取数据的引脚</p>
返回值	如果成功获取引脚电平数据则返回 TRUE, 否则返回 FALSE
相似函数原型	<code>BOOL GetOverflowPinBusData(ULLONG&amp; dat, SIMBUS&amp; bus) ;</code>
注意事项	无

函数原型	<code>BOOL GetHostReadPinData(BYTE&amp; dat, SIMPIN&amp; pin, const SIMPIN&amp; spin);</code>
作用	获取给定引脚 ( spin ) 对应的引脚采集的数据
形参	<p>dat: 返回的引脚电平数据, 仅 “1” 与 “0” 有效</p> <p>pin: 需要获取数据的引脚</p>

	spin: 给定的标记引脚
返回值	如果成功获取引脚电平数据则返回 TRUE, 否则返回 FALSE
相似函数原型	BOOL GetHostReadPinBusData(ULONGLONG& dat, SIMBUS& bus, const SIMPIN& spin);
注意事项	无

函数原型	BOOL GetPinDefaultMode(BYTE& mode, SIMPIN& pin) ;
作用	获取引脚默认的电气类型 (即“引脚分配”对话框中设置的电气类型)
形参	pin: 需要获取电气类型的引脚  mode: 返回的电气类型数据, 相应的值为输入 (LPM_INPUT)、输出 (LPM_OUTPUT)、双向 (LPM_INOUT)、接地 (LPM_GROUND)、电源 (LPM_POWER)、无 (LPM_NONE)
返回值	如果成功获取电气类型则返回 TRUE, 否则返回 FALSE
相似函数原型	BOOL GetPinBusDefaultMode(pmodeColection& pm, SIMBUS& bus) ;
注意事项	此处的电气类型并非直接与远程模块相关的电气类型

函数原型	BOOL GetPinDefaultData(BYTE& dat, SIMPIN& pin) ;
作用	获取引脚默认的电平 (即“引脚分配”对话框中设置的默认电平)
形参	dat: 返回的电平数据, 仅“1”与“0”有效  pin: 需要获取电平数据的引脚
返回值	如果成功获取默认电平则返回 TRUE, 否则返回 FALSE

相似函数原型	<code>BOOL GetPinBusDefaultData(ULL&amp; dat, SIMBUS&amp; bus);</code>
注意事项	无

函数原型	<code>BOOL GetPinAssignMask(BYTE&amp; dat, const SIMPIN&amp; pin, BOOL flag = TRUE);</code>
作用	获取引脚的引脚分配状态（即“引脚分配”对话框中设置的引脚分配状态）
形参	<p><code>dat</code>: 返回的引脚分配状态，“1”表示已分配</p> <p><code>pin</code>: 需要获取远程引脚编号分配状态的引脚</p> <p><code>flag</code>: 是否考虑当前设置的平台，此值为 <code>TRUE</code> 表示考虑，<code>FALSE</code> 为不考虑。</p> <p>例如，VC02 的远程引脚编号范围为 1~18，但某个引脚分配为 20（超出范围），如果 <code>flag</code> 为 <code>TRUE</code> 则返回 0（未分配），为 <code>FALSE</code> 则返回 1（已分配）</p>
返回值	如果成功获取引脚分配状态则返回 <code>TRUE</code> ，否则返回 <code>FALSE</code>
相似函数原型	<code>BOOL GetPinBusAssignMask(ULL&amp; dat, SIMBUS&amp; bus, BOOL flag = TRUE);</code>
注意事项	无

函数原型	<code>VOID ReadPinData(SIMPIN pin);</code>
作用	主动获取引脚的电平数据
形参	<p><code>pin</code>: 此次获取引脚电平数据的标记引脚，调用 <code>GetHostReadPinData</code> 函数时应该传入相同的数据以获取需要的电平数据</p>
返回值	无

注意事项	无
------	---

函数原型	VOID SetPinData(SIMPIN& pin, BYTE dat) ;
作用	设置引脚的电平
形参	dat: 设置的引脚电平 pin: 需要设置电平的引脚
返回值	无
相似函数原型	VOID SetPinBusData(SIMBUS& bus, ULONGLONG d) ;
注意事项	无

函数原型	VOID SetPinBusMode(SIMBUS& bus, RPINMODE mode);
作用	设置远程模块的引脚电气类型
形参	dat: 设置的引脚电气类型  pin: 需要设置电气类型的引脚, 有效的电气类型包括 RPM_IF ( 悬浮输入 ) RPM_IPD ( 输入下拉 )、RPM_IPU ( 输入上拉 )、RPM_OUTPP ( 输出推挽 ) RPM_OUTOD ( 输出开漏 )
返回值	无
相似函数原型	VOID SetPinBusMode(SIMBUS& bus, RPINMODE mode) ;
注意事项	无

函数原型	VOID SetPinDataDelay(WORD delay) ;
------	------------------------------------

作用	设置数据电平的延时
形参	delay: 电平延时的数量
返回值	无
注意事项	延时的单位并非准确值, 取决于远程目标执行一条指令的时长

### 5.2.10 缓存控制

函数原型	VOID BeginCache();
作用	开始缓存
形参	无
返回值	-
相似函数原型	VOID BeginCache(DRECT rect);
注意事项	必须与 EndCache 配对使用

函数原型	VOID EndCache();
作用	开始缓存
形参	无
返回值	-
相似函数原型	无
注意事项	必须与 BeginCache 配对使用

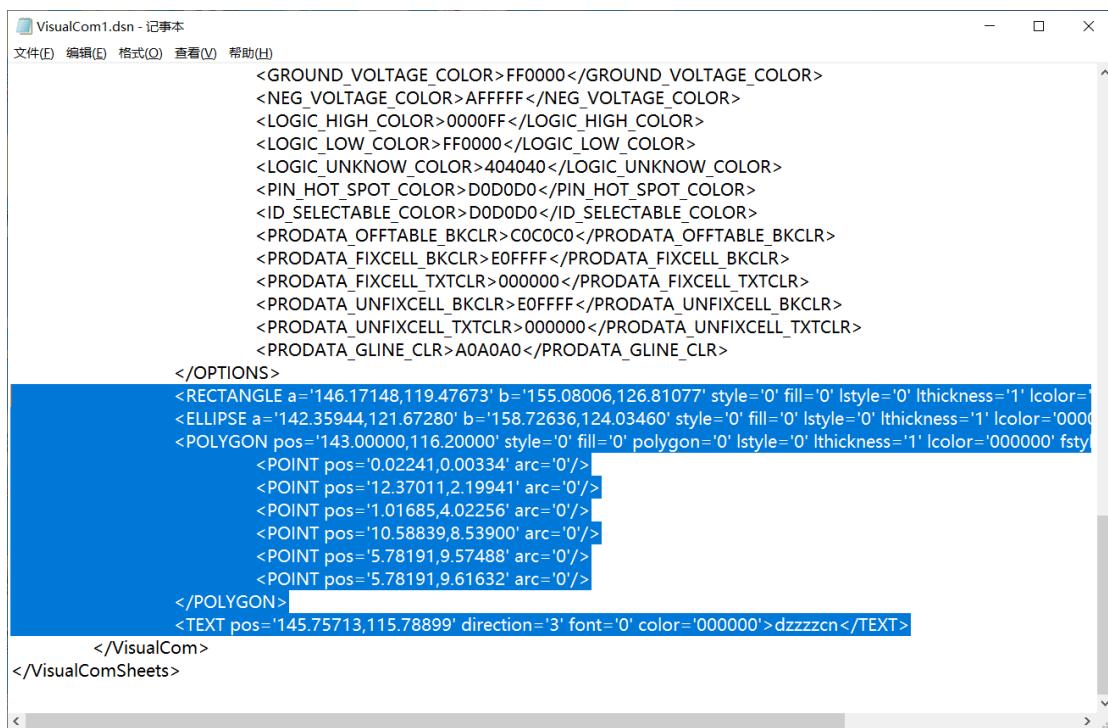


# 第 6 章 常见问题

本章包含一些你可能会遇到的问题解决方案，后续将会更新其它常见问题。

**6.1 我插入一个 LED 数码管图片试图描出相应的边框，弄到一半后就想看电影，所以就保存了，回来再打开就发现视图无法缩放，也看不到任何东西**

答：可以直接用记事本打开原理图文件，</OPTIONS>标签下就是相应的对象，只需要将其复制后粘贴到新原理图文件中的相同位置即可。



The screenshot shows a Windows Notepad window titled "VisualCom1.dsn - 记事本". The content of the file is a XML-like configuration for a schematic sheet. A blue rectangular selection highlights the following code block under the </OPTIONS> tag:

```
<RECTANGLE a='146.17148,119.47673' b='155.08006,126.81077' style='0' fill='0' lstyle='0' lthickness='1' lcolor='000000'>
<ELLIPSE a='142.35944,121.67280' b='158.72636,124.03460' style='0' fill='0' lstyle='0' lthickness='1' lcolor='000000'>
<POLYGON pos='143.00000,116.20000' style='0' fill='0' polygon='0' lstyle='0' lthickness='1' lcolor='000000' fsty='0'>
    <POINT pos='0.02241,0.00334' arc='0'/>
    <POINT pos='12.37011,2.19941' arc='0'/>
    <POINT pos='1.01685,4.02256' arc='0'/>
    <POINT pos='10.58839,8.53900' arc='0'/>
    <POINT pos='5.78191,9.57488' arc='0'/>
    <POINT pos='5.78191,9.61632' arc='0'/>
</POLYGON>
<TEXT pos='145.75713,115.78899' direction='3' font='0' color='000000'>dzzzcn</TEXT>
```

**6.2 本地计算机创建并开发的元件及仿真模型，如何在其它计算机中同样运行呢？**

答：将仿真的原理图文件（\*.dsn）及相应的 DLL 文件转过去即可（DLL 文件需要拷贝到 VisualCom 软件平台安装目录下的 model 文件夹）

**6.3 为什么 VisualCom 软件平台版本升级后，原来用得好好的仿真模型无法工作了呢？**

答：系统升级后，可能对 ICOMPONENT 结构体增加了更多的功能（IDSIMMODEL 结构体通常不会变化），这样原来仿真模型中的虚函数与新版本 VisualCom 软件平台之间的对

接关系也就不一样，你只要从 VisualCom 软件平台新的安装目录下获取最新的 vsm.h，再拷贝到自己的工程重新编译获得最新的 DLL 文件即可。

#### 6.4 为什么我开发的元件模型总是没反应，连最简单的读取引脚（打印调试信息）时都没有任何变化，但是“输出”窗口“状态”标签页中明明已经显示模型加载成功

答：有可能使用了错误版本的 vsm.h。

#### 6.5 与远程硬件模块配合使用的时候，我觉得输出电平数据的速度有些慢（并没有调用 SetPinDataDelay 函数添加延迟），有什么方法可以适当提升呢？

答：数据从电脑传输到远程硬件模块（即下行）的速度主要取决于“仿真参数”对话框中的“采集数据超时时间”项，其值代表每次从远程硬件模块读取数据的最大等待时间（默认 1000ms），适当减小该值可以提升数据下行速度（因为如果没有数据可读取，VisualCom 软件平台会一直读取直到超时（之后才会传输下行数据）。但是需要注意：**此值不宜设置过小，否则可能会出现数据包丢失，继而导致下行数据无响应。**



## 版本历史

日期	内容	备注
2020.12.4	第一个版本发布	
2021.1.10	V1.1 发布	
2023.1.25	V2.1.0	
