

# OPNET 用户指南翻译

以下文档为笔者根据 OPNET 的联机帮助文档翻译过来的，供大家学习参看，请勿用于商业用途，有问题可以和本人联系，由于本人主要是使用 Modeler 进行编程，所以只是翻译了这部分的内容。希望我的工作能起到抛砖引玉的作用，有更多的文档能够出现从而简便大家对 OPNET 的学习。联系 e-mail 为：[life\\_beauty@163.com](mailto:life_beauty@163.com)

## OPNET 文件格式介绍

OPNET 的文件众多，文件名后缀也各种各样，经常看到网上有人提出将 OPNET 产生的文件的文件名后缀作一个总结，下表列出了 OPNET 的常用文件。

常用文件名后缀及文件描述

文件名后缀	描述	文件格式
.ac	分析配置文件	二进制文件
.ah	动画文件	二进制文件
.bkg.i	背景图片	二进制文件
.ef	环境文件	ASCII 数据
.em.c	EMA C 代码	C 代码
.em.o	EMA 目标文件	目标代码
.em.x	EMA 执行程序	可执行程序
.esd.m	外部系统模型	二进制文件
.ets	外部工具支持文件	ASCII 数据
.ets.c	外部工具支持 C 代码	C 代码
.ets.cpp	外部工具支持 C++代码	C++代码
.ex.c	外部 C 代码	C 代码
.ex.cpp	外部 C++代码	C++代码
.ex.h	外部头文件	C/C++头文件
.ex.o	外部目标文件	目标代码
.fl.m	过滤器模型文件	二进制文件
.ici.m	ICI 模型文件	二进制文件
.lk.d	派生的链路模型	二进制文件
.lk.m	链路模型	二进制文件
.map.i	地图	二进制文件
.nd.d	派生的结点模型	二进制文件
.nd.m	结点模型	二进制文件
.nt.m	网络模型	二进制文件
.os	输出矢量	二进制文件
.ov	输出标量	二进制文件
.path.d	派生路径模型	二进制文件
.path.m	路径模型	二进制文件
.pb.m	探针模型	二进制文件
.pdf.m	概率密度函数	二进制文件/可编辑
.pdf.s	概率密度函数	二进制文件/可导入仿真
.pk.m	包格式模型	二进制文件
.pr.c	进程 C 代码	C 代码
.pr.cpp	进程 C++代码	C++代码
.pr.m	进程模型	二进制文件

.pr.o	进程模型	目标文件
.prj	项目模型	二进制文件
.ps.c	管道阶段 C 文件	C 代码
.ps.cpp	管道阶段 C++ 文件	C++ 代码
.ps.o	管道目标文件	目标文件
.sd	仿真描述	ASCII 文本
.seq	仿真序列	ASCII 数据
.sim	可执行的仿真	可执行文件

## OPNET Modeler 开发环境介绍

对于 Modeler 来说，使用三层建模机制来刻画系统模型，分别为网络层、结点层和进程层。相应的，Modeler 提供了 3 种编辑器来刻画这 3 个层次的模型，分别为网络编辑器、结点编辑器以及进程编辑器。另外 Modeler 还提供了很多其他的编辑器，方便了整个建模的过程，如包编辑器、图标编辑器、天线模型编辑器等等。Modeler 提供了项目编辑器来管理整个项目。


### 项目编辑器

Modeler 采用“项目 - 仿真环境”的方式来对网络进行模拟。一个项目就是一组仿真环境，每个仿真环境针对网络的不同方面。每个项目至少包含一个仿真环境。仿真环境是网络的一个实例，一般来说，一个仿真环境针对一种网络配置，而配置在这里的意思就是拓扑结构、协议、应用、流量，以及仿真设置。

OPNET 对网络的建模以三类对象为基础：子网、结点和链路。在项目编辑器中，可利用这三类对象创建和编辑网络模型，创建结点和链路的派生模型，定制网络环境，并实现仿真和对仿真结果的分析。

在项目编辑器中有多种方法来进行对网络模型的建模：

(1) 使用开始向导 (startup wizard) 来确定仿真环境的初始环境，在 Modeler 的 File 菜单下选择 new 就可以看到开始向导 (startup wizard)。在开始向导 (startup wizard) 中会提示您如何建立拓扑结构 (包括创建空的拓扑结构，从 OPNET 的 ACE 环境中创建，从 HP 的 OpenView 导入，从路由器的配置文件导入，从 ATM 的文本文件导入，从 Tivoli 的 NetView 导入，从 OPNET 的 VNE 服务器导入，从 XML 文件导入)，网络的规模 (包括世界，企业，校园，办公室，逻辑以及从地图中进行选择)，网络中需要使用到的一些技术和设备模型 (如 IP 还是 ATM，使用 Cisco 的设备还是 Juniper 的设备)。

(2) 也可以在项目编辑器里的对象面板选择自己需要的模型来进行建模。点击项目编辑器中的  图标，即可出现对象面板。从对象面板里拖拽需要的结点和链路模型至项目编辑器中的工作区间，搭建网络的拓扑结构。

(3) 使用快速配置选项 (Rapid Configuration) 来进行网络配置。快速配置在 topology 菜单下。

项目编辑器包括了一个用于创建和编辑网络模型的工作空间。子网络和结点作为对象被放置在工作空间中，并用图标来表示。连接线表示了结点和子网络间的通信链路。网络对象的特点用对象的属性来代表，决定了网络对象在整个模型内如何运作。

## 网络结点

Modeler 包含三种类型的结点，一种为固定结点，例如路由器，交换机，工作站，服务器等等都属于固定结点，一种为移动结点，例如移动台，车载通信系统等等都是移动结点，另一种为卫星结点，顾名思义是代表卫星。每种结点所支持的属性也不尽相同，如移动结点支持三维或者二维的移动轨迹，卫星结点支持卫星轨道。

## 子网

OPNET 的子网和 TCP/IP 的子网不是同一个概念，OPNET 的子网只是将网络中的一些元素抽象到一个对象中去。子网可以是固定子网，移动子网或者卫星子网。子网不具备任何行为，只是为了表示大型网络而提出的一个逻辑实体。一个简单的例子，如运营商的骨干网，假如把骨干网上的所有路由器都放到一个视图里，显得十分的乱，不如按照省份，将同一省份的路由器都放到同一个子网中，然后以省份的名称来命名每个子网的名字，构建成的网络看上去比较的有条理。

## 链路

相对固定结点，移动结点以及卫星结点，链路也有不同的类型，有点到点链路，总线链路以及无线链路。点到点链路在两个固定结点之间传输数据，总线链路是一个共享媒体，在多个结点之间传输数据，无线链路是在仿真中动态建立的，可以在任何无线的收发信机之间建立。卫星和移动链路必须通过无线链路来进行通信，而固定结点也可以通过无线链路建立通信连接。

## 菜单

项目编辑器为创建和处理网络模型提供了多种操作。从项目编辑器菜单栏中可以访问这些操作，项目编辑器菜单栏包括如下子菜单：

File——与高层功能相关的操作，诸如打开/关闭项目、保存仿真环境、导入模型以及打印图表和报告。

Edit——编辑控制程序运行的环境属性，以及维护文本和对象的操作。

View——影响编辑器视窗和其中内容的操作。

Scenarios——提供对项目中所包括的仿真环境的控制。

Topology——与网络拓扑相关的操作，包括建立网络和创建网络对象。

Traffic——与规定网络业务相关的操作，如导入业务文件和规定穿过网络的路由。

Protocols——与特定协议模型相关的操作。

Simulation——用于配置和运行仿真。

Results——控制统计结果搜集和查看。

Windows——列出所有已打开的编辑器窗口，并允许激活其中的一个。

Help——提供上下文相关帮助、联机文档和手册以及关于程序的相关信息。

## 结点编辑器

结点通常被看作设备或资源，数据在其中生成、传输、接收并被处理，由支持相应处理能力的硬件和软件共同组成。

OPNET结点编辑器提供了模拟内部功能所需的资源。在结点编辑器中，用户可以使用多种模块，每种模块实现了结点行为的某一方面，诸如数据生成、数据存储、数据的处理或

选路和数据的传输等。单个结点模型通常由多个模块(有时是几十个甚至几百个模块)组成。数据包流和统计线可将不同的模块相连，其中数据包流承载了模块间数据包的传输，统计线可实现对模块内变化量的监视。通过模块、数据包流和统计线的联合使用，用户可对结点的行为进行仿真，同时也可以将特定的接收器和发送器视为紧密相连的模块对。

## 工具栏



**处理机**：这是结点编辑器中最常使用的模块，处理机的行为可以完全由用户来进行设置，它和其他模块的连接也是任意的。



**队列**：队列提供的功能是处理机的超集，相对处理机，它多了一些属性，如子队列。



**数据包线**：连接两个模块，并且在源和目的之间传输数据包。它代表了在实际的通信结点中的硬件及软件接口。



**统计线**：用于两个模块之间的数值传递。统计线一般是帮助进程来监控设备状态以及性能的变化，创建结点内进程间的简单通信机制。



**逻辑线**：用于指定结点内的两个模块的逻辑关联，如一对收发信机，逻辑线不在模块间传递任何数据。



**点到点发信机**



**总线发信机**



**无线发信机**



**点到点收信机**



**总线收信机**



**无线收信机**



**天线**：用户确定无线收发信机的天线特性。



**外部系统接口**：外部系统接口模块是队列模块的超集，用于和外部系统的接口。

## 进程编辑器

可将与计算机系统和通信网络相关的进程看作对数据进行处理的一系列逻辑操作以及相应的条件。这些进程通过硬件或软件组实现。OPNET 进程模型描述了实际进程中的逻辑，例如：通信协议和算法、共享资源管理、排队原则、专用的业务发生器、统计量搜集机制以及操作系统。

在 OPNET 进程编辑器中，使用了图形和文本的结合。状态转移图（STD，State Transition Diagram）可用于描绘进程模型的总体逻辑构成。STD 内的图标表示了逻辑状态，连线表示了状态间的转移。进程模型所执行的操作用 C 或 C++ 语言进行了描述。进程编辑器内图形

和文本形式的结合有以下两个主要优点：首先，可通过图形直观的查看进程模型及模型间进行控制的流。其次，C 或 C++ 语言的描述可降低模型的复杂性，同时提高仿真的逼真度。状态既可以是强制的（绿色所示）也可以是非强制的（红色所示）。强制状态将依次执行其进入代码和离开代码，然后将控制权转交给下一个状态。非强制状态将会在执行进入代码后暂停，允许仿真过程转向模型中的其他实体和事件，此时该进程进入非强制状态，并等待下一次中断，诸如包到达或计时器超时。

## 工具栏



创建状态：在进程模型中创建新的状态。



创建转移：在状态之间创建转移。



设置初始状态：设置进程模型中的某个状态为初始状态。



状态变量区：定义状态变量，状态变量在不同的进程唤起之间，能够保持值。



临时变量区：定义临时变量，临时变量在不同的进程唤起之间，无法保持其值不变。



头区域：定义常量，宏，头文件，全局变量，数据结构，数据类型，以及函数声明。



函数区域：定义和进程相关的C/C++函数。



诊断区：定义C/C++语句，将诊断信息输出到标准输出设备中。



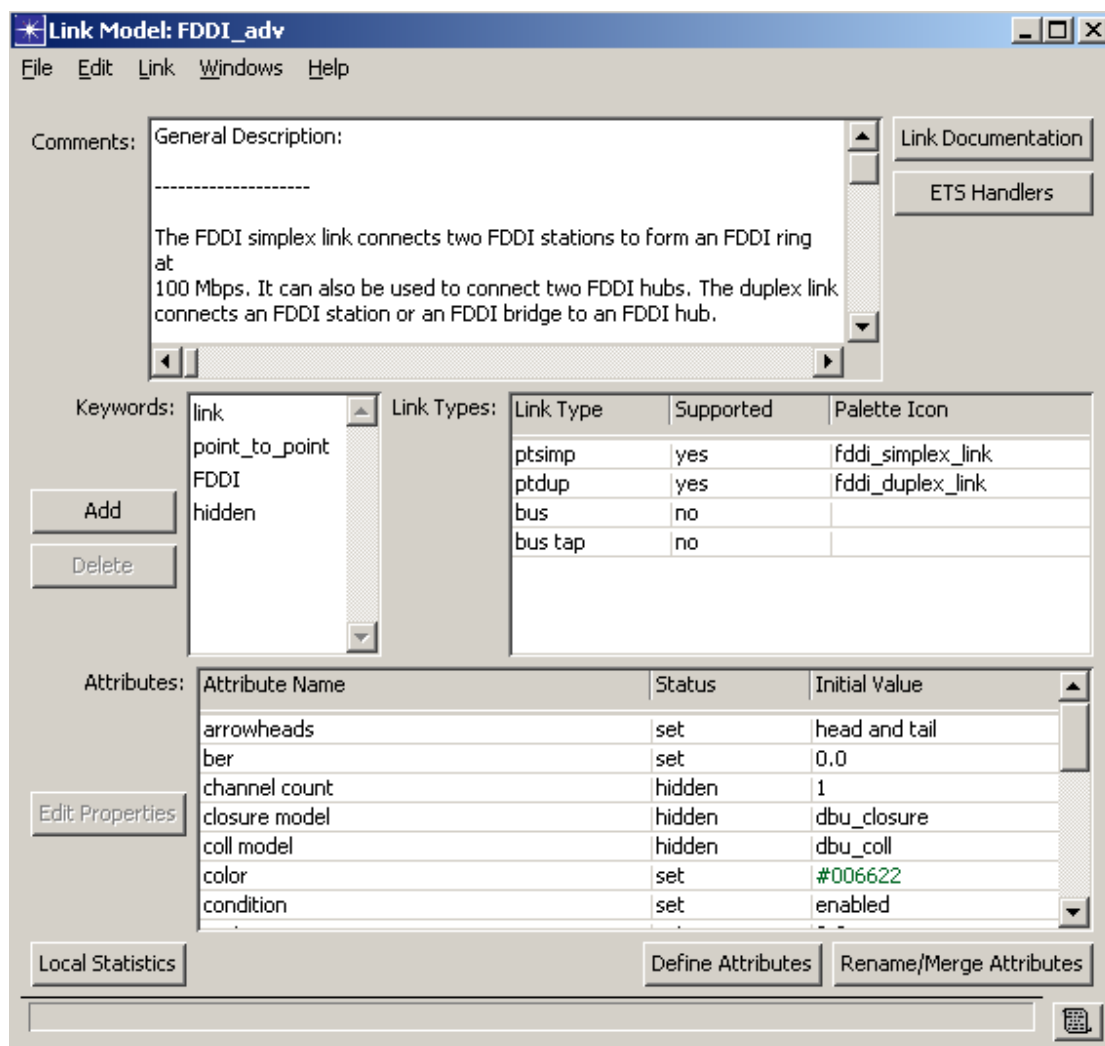
终止区：定义C/C++语句，这些语句将在进程销毁时执行，一般为释放内存等语句。



编译进程：生成进程的 C/C++ 的源代码，以及目标代码。

## 链路编辑器

在链路模型中对各个链路对象进行了说明。对于不同的链路对象，每一类链路都包含了特有的属性接口、注释以及表示方法。在项目编辑器中创建的链路是链路模型的特定实例，因此在对链路模型的属性进行修改时，链路实例会自动的继承修改后的属性。



链路编辑器

在对话框中规定了如下信息：

Link Types（所支持的链路类型）

每一链路模型可以支持四种基本的链路类型中的一种或多种，即ptsimp（点对点双工链路）、ptdup（点对点单工链路）、bus（总线链路）和bus tap（总线分接链路）。请注意，无线链路的设计不包含在链路编辑器中，它是由仿真内核（Simulation Kernel）经过动态定义生成的与结点相对位置、传输及运行环境中诸多因素相关的函数。

Key Words（关键字）

链路模型的关键字允许有选择的在项目编辑器对象面板中显示链路模型。在配置对象面板时，OPNET将关键字与所请求的关键字进行比较，以此决定是否将此模型作为选择。此机制可减少在对象面板中的模型数，而只显示那些与当前应用相关的模型。

Model Comments（模型注释）

在链路模型中包含了一系列注释，这些注释描述了链路的特性、潜在应用和用户可能涉及的任何信息。因为有些用户（例如，IT DecisionGuru用户）无权访问链路模型内部，注释就成为此类用户可利用的主要信息。通过将模型接口的相关文档作为模型自身固有的部分嵌入到模型中，OPNET为用户访问信息提供了便利。

Attribute Interfaces（属性接口）



结点和进程模型可以分别影响结点和模块的属性表达和使用，同样，链路模型为项目编辑器中链路对象的属性提供了规范说明。链路模型和链路之间的关系与进程模型和模块间的相互作用最为类似。和进程模型相似，链路模型中不包含可以提升属性的对象。因而，链路可以提升的唯一属性是“链路模型属性”。和进程模型一样，链路模型可以通过属性预分配、属性隐藏、属性重命名和改变属性优先级为链路对象的内嵌属性规定配置信息。

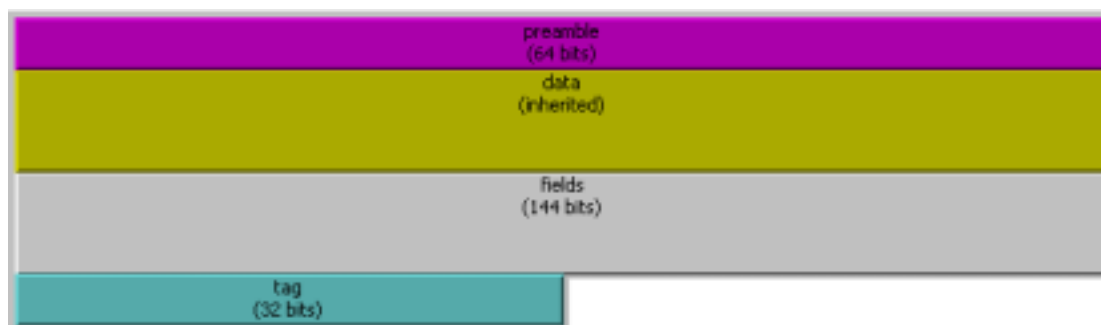
Attribute Specification (属性说明)

用户改变链路属性设置时，OPNET允许改变其默认行为。可以通过单击ETS Handlers按钮规定定制的ETS库和事件句柄。

## 包编辑器

包格式是由字段集合而成的一种结构。包格式规定了每一个字段的名称、数据类型、默认值、大小以及相关注释(可选项)。在核心程序的调用中，可通过在进程编辑器的File: Declare Packet Formats引入某一特定类型的数据包。

在图形环境中，字段被表达成一系列彩色的矩形。矩形大小与size属性中规定的比特数目成正比。



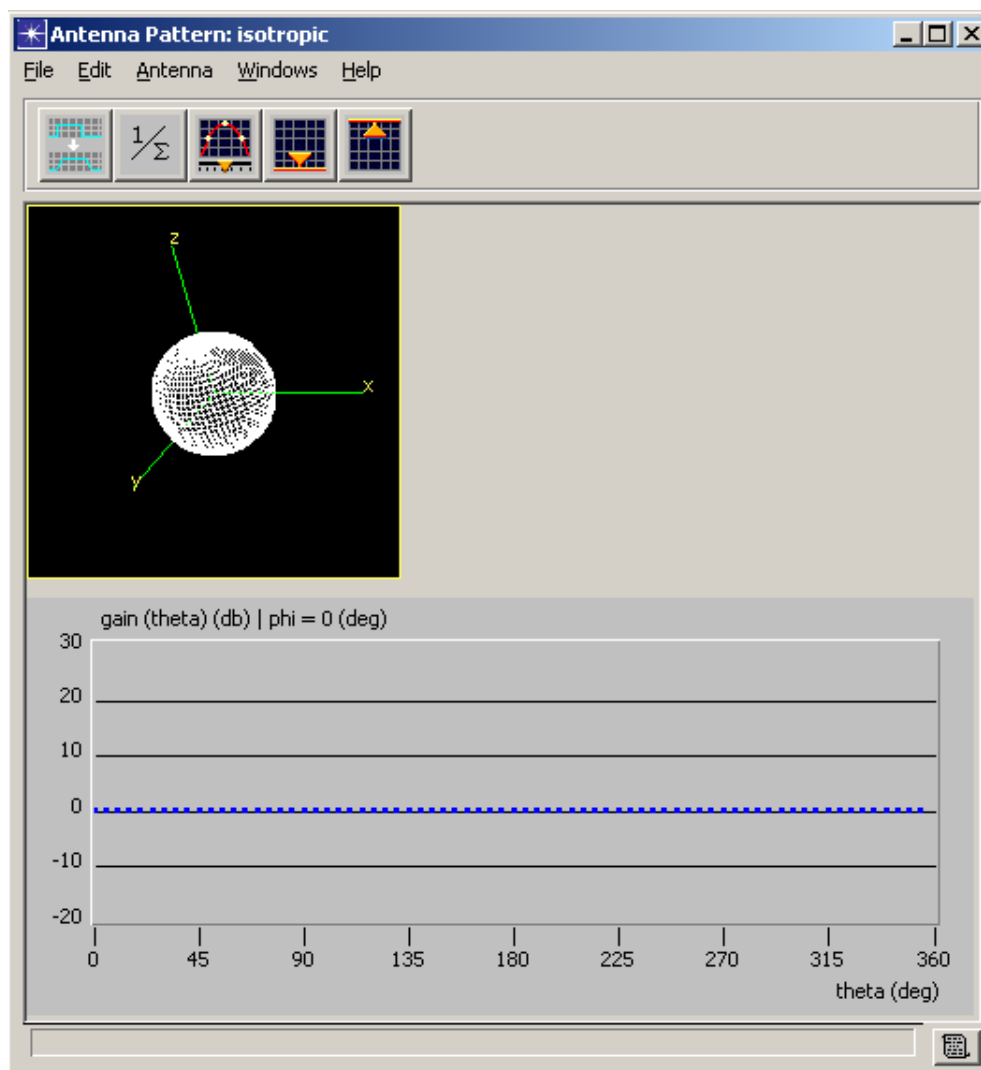
包格式中的字段表示

右击包字段打开该字段的 Attributes 对话框，可对这一字段的属性进行编辑，也可进行相应的注释。字段可以按照任意顺序放置，因为包格式通常通过字段名进行调用。但是，如果模型通过索引引用包字段，那么改变图形表达将会有很大区别。OPNET 把索引 0 分配给左上方的字段，并把最高的索引分配给右下方的字段。

## 天线模型编辑器

信号的接收功率通常是由诸多因素构成的函数计算得到的，这些因素包括天线间的方向矢量和沿着这一方向矢量的每一个天线的增益。在给定结点相对位置的情况下，天线模型编辑器中规定的天线增益模型可被用于提供增益值。





天线模型编辑器

用户可以通过天线模型编辑器的菜单栏访问创建和处理天线模型的相关操作 ,天线模型编辑器的菜单栏包含如下内容：

File——包含与高层功能相关的操作，诸如打开和关闭项目、保存仿真环境、导入模型以及打印图形和报告。

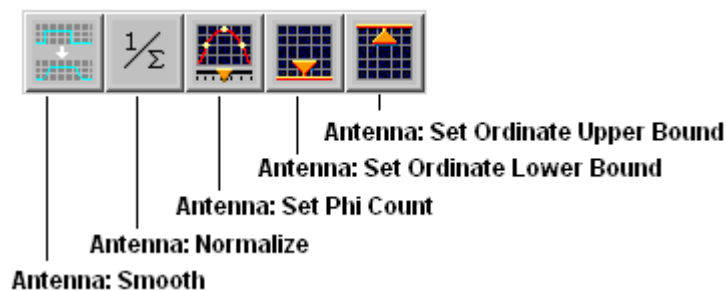
Edit——包含允许编辑控制程序运行的环境属性的操作，也包含维护文本和对象的操作。

Antenna——包含修改天线模型显示的操作。

Windows——列出所有打开的编辑器窗口，并且允许激活其中的一个窗口。

Help——提供对上下文帮助、联机文档以及关于程序的信息的访问。

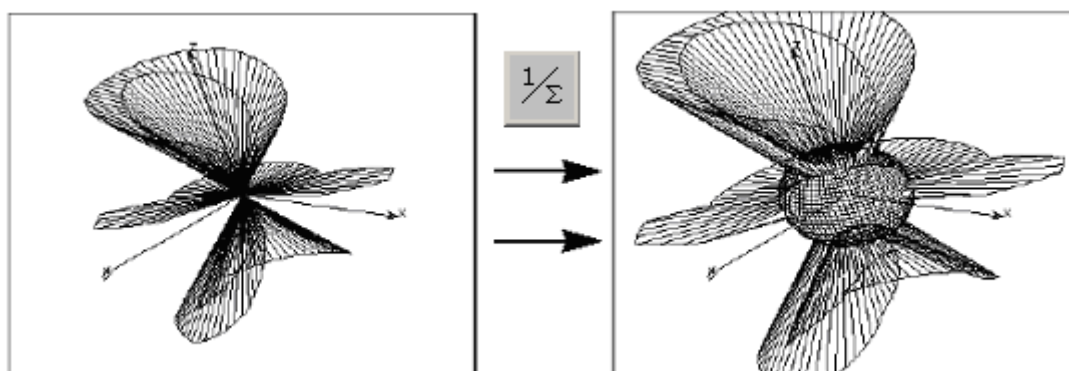
天线模型编辑器为频繁使用的操作提供了快捷按钮。



天线模型编辑器中的快捷按钮

## 1. 规格化

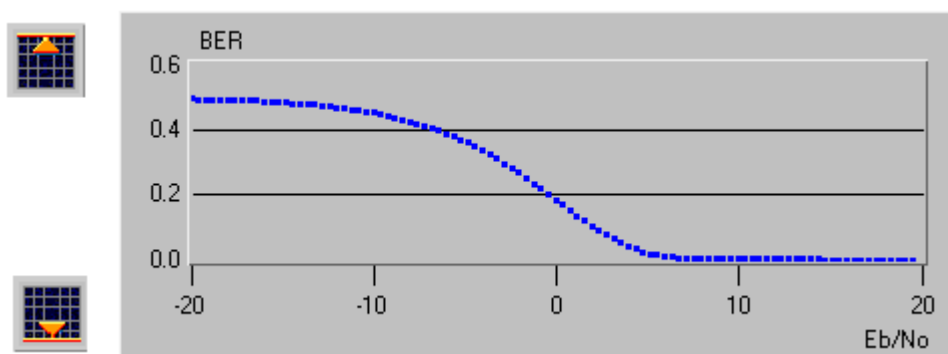
此操作可适当的按比例缩减或增大整个模型，以致整个球型模型上的总增益为零 (0dB)。



规格化的天线模型

## 设置纵坐标约束

设置纵坐标约束 (Set Ordinate Bounds) 操作为天线模型图形的Y轴设置上限和下限。



设置纵坐标约束

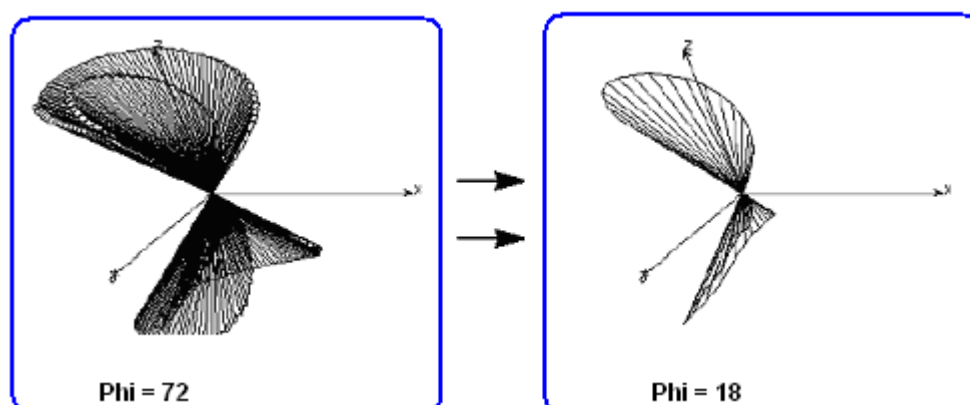
## 设置 Phi 数

此操作为线图或三维图形设置抽样数目。抽样数目决定了在近似构造连续函数时使用多少个样本元素或离散元素。

就天线增益模型而言，抽样数决定了模型的两个特性：用于增益表创建的二维分片数和用于任一增益表中增益分配的离散间隔数。值的数目总是分片数（的离散取值）的二倍。默认抽样数为36，意味着有36个分片。那么每个分片有72个5°间隔的抽样。

此操作既可以在图形方式创建函数前使用，也可以在函数创建/读入后使用。一旦

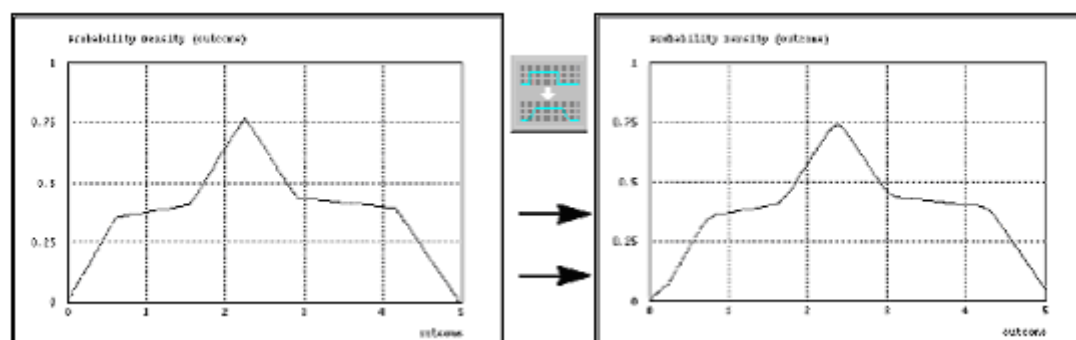
设置了取样点数，OPNET以新的解决方案重绘图形，如下图所示。



带有不同 Phi 数的天线模型

平滑

此操作平滑图形的尖锐边缘和毛刺。平滑算法的实现通过一个沿着 X 轴滑动的窗口实现。



平滑前后的线图

## 编辑天线模型

在天线编辑器中，用户可以创建、编辑以及查看天线的模型。天线模型是一系列以dB为单位建立的三维天线增益的二维函数。

通过EMA编码创建的天线模型表在天线模型编辑器中仍能以图形方式查看。这种方式克服了以图形方式建立天线模型时不可避免的不精确。

使用天线模型编辑器（Antenna Pattern Editor）时，可以执行3类操作。

- （1）使用set plane、increase plane和decrease plane按钮选择天线模型的圆锥分片。
- （2）使用二维图形为当前分片键入增益模型。
- （3）使用缩放和旋转命令维护三维视图。

为尽可能利用天线模型编辑器（Antenna Pattern Editor）的天线模型编辑模型，必须理解解线图和set plane、increase plane以及decrease plane命令如何在三维空间中相互作用。本质上，天线增益模型被分解为一系列锥形部分。您可选择感兴趣的2D切片进行观看或编辑。

注意：天线模型编辑器（Antenna Pattern Editor）的坐标系是左手坐标系。

对调制曲线和天线模型的编辑是在无线链路的仿真中十分常用的。

（1）从文件菜单中选择 New...，从下拉菜单中选择 Antenna Pattern，出现天线模型编辑器视窗。

（2）如果想要改变分片的数目，单击 Set Plane 按钮，然后选择想要的 phi 值。

注意：默认情况下，有36个5°间隔的连续分片。使用Set Phi Count操作来修改分片数和分片之间的相应间隔。此后，OPNET重新绘出图形以便显示所选phi值的图形。

(3) 将增益表示为所选phi值的函数。

(4) 修改分片：

- 设置坐标上限和下限约束，将增益的上限、下限值设置为任意整数。角度域不能被改变，有效范围总是  $-0^\circ$  到  $=360^\circ$ 。可以在定义分片之前或分片后设置上限和下限约束。如果在定义分片之后设置约束，OPNET不改变分片，但会重新绘出线图与坐标的新范围进行匹配。

- 使用平滑操作对分片中的尖锐边缘进行平滑。

- 使用规范化操作，使整个增益模型表(包括所有分片)在所有方向上的增益为0dB。

(5) 通常，完整的天线增益模型需要规定多个分片，也就是说，为多个phi取值绘天线增益表。通过重复上述步骤(2)到步骤(4)，编辑其余的分片。注意：如果编辑相邻的分片，使用Increase Plane和Decrease Plane命令更快更容易。

### 维护天线模型的3D视图

(1) 重复使用 Zoom In 和 Zoom Out 按钮放大或缩小图像。

(2) 使用Rotate X、Rotate Y和Rotate Z按钮获得不同的透视图。使用Rotate Angle按钮可以将每个旋转的幅度变成任意正整数或负整数。

编辑线图操作用于在线图上键入新的数据点(诸如天线模型或PDF曲线)。图形被规定为一系列相互连接的直线段。绘出新线段时，任何占据相同横坐标范围(x坐标的相同范围)的已存在的直线段将自动被替换。

### Modeler 编程

Modeler 的编程并不纯粹是代码的编写，OPNET 的编程包括进程的建模、C/C++代码的书写、以及 OPNET 自身提供的函数的调用等等。

对于一个典型的 Modeler 程序来说，需要进行代码书写的地方有以下几处：进程模型中的状态变量、临时变量、头区域、函数区域、进入和离开代码、转移代码，以及外部的头文件和源文件。

熟悉 OPNET 的编程，一方面要清晰的了解 OPNET 面向对象的三层建模机制，另一方面也要熟悉 OPNET 提供的函数库的使用。

状态变量的定义并没有什么特殊之处，特殊的只是状态变量本身，它可以在进程唤起的过程中保持原来的值，这类似于函数里的静态变量，可以在函数调用的过程中保持原来的值。Modeler 为使用者提供了定义状态的界面，查看其源文件，发现它和我们使用的标准 C/C++ 定义变量并无二异。单击“Edit ASCII”就可以看到定义状态变量的源代码。

### 头区域以及头文件的使用

Modeler 进程模型的头区域功能类似于头文件，用于定义数据结构、宏、常量以及申明函数等等。

除了提供头区域外，Modeler 也支持传统的“.h”头文件，原则上，头文件和头区域的定义是一样的，但是就功能来说，有一些区别。无论是头区域中定义的函数，还是变量，只适用于本进程模型，如果别的进程模型需要同样的定义和申明，则需要在其进程模型的头区域再进行定义。而头文件则不同，大多定义了整个系统或者一些进程需要共用的变量，数据结构以及函数的声明。在不同的进程中使用，只需要在头区域中将该头文件包含即可。

## 函数区域以及外部文件

函数的申明在头区域或者头文件中，相应函数的具体定义就放在了函数区域和外部文件中。在函数区域中定义函数，需要注意的是函数的入口和出口应该使用 FIN 和 FOUT（或者 FRET）。这样利于 OPNET 在编译，以及在运行中发生错误时进行定位。

进程模型的外部文件可由 OPNET 的外部代码编辑器来进行编写和编译。文件名为 xxx.ex.c，编译后的文件名为 xxx.ex.o。

编译后的外部文件只需在进程模型中进行申明即可。

## 进入，离开以及转移代码

进入、离开以及转移代码部分可以自由的添加用户的函数，以及其它代码。

## OPNET 核心函数简介

在OPNET中的代码的编写中，核心函数（Kernel Procedure，KP）起着十分重要的作用。所谓核心函数，指的是一些可以被进程模型和收发信机管道阶段作为中断被调度的C/C++函数，或者普通C/C++函数等所调用的函数。基于核心函数所操作的对象类型，可按功能分为若干类。每一类内的KP统称为一个函数，同一函数内部的函数，其名称拥有相同的函数关键词。例如：对于那些主要用来处理包的KP，它们被归于包函数，使用“pk”作为关键词。

熟练掌握关键核心函数的使用（包括功能、参数、返回值等），了解核心函数分类规则及每一类核心函数主要功能范围，能够迅速查找到所需使用的核心函数并正确应用，是OPNET 编程渐入佳境的关键一步。

### 核心函数命名规则

核心函数命名拥有标准的结构。之所以采用这种结构，主要是为了使得核心函数在C/C++程序中具有更好的可读性，避免与非OPNET函数/变量相混淆。同时也为了在同类型函数名中保持一定的相关性。命名结构拥有以下几个简单的规则：

- 以前缀“op\_”起始：标志该函数为OPNET仿真内核所提供。
- 第二个词为函数名称（小写）：代表所操作的对象缩略名称（如：pk，ici）。
- 第三个词为子包名称：提供了该KP进一步类属信息（如：p\_pk\_nfd\_set()中的nfd）。
- 一般KP要对对象进行操作，被操作对象在具体操作动作之前表示，例如attr\_set和subq\_flush，对象分别为attr和subq，操作分别为set和flush。

### 参数类型

许多核心函数的参数和返回值是用户熟悉的标准C/C++类型（如int，char，double），但除此之外，也还有许多OPNET自定义的数据类型，这些数据类型是通过typedef从OPNET仿真数据结构中继承的。虽然用户经常在编程中与OPNET自定义数据类型打交道，他们也因此对基本的数据类型十分熟悉，但对于每一数据类型内部的确切构造却不必深刻理解。在下文中将会对各种特殊的数据类型加以介绍。

动画（Anim - Animation）函数主要依赖于数字型的“序号”（ID）来指代操作中遇到的动画实体。因为ID对于动画浏览函数op\_vuanim来说，是在超出仿真的范围内通信的，所以采用整数型IDs取代C/C++指针。尽管简单的整型数据可以作为ID，ID也可以被存储于普通的C/C++整型变量中，OPNET还是提供了一些特殊的数据类型来精确的标志ID参数和变量。基于ID的三种OPNET动画实体是：查看器（viewer），宏（macro）和描绘（drawing），它们的相关数据类型如下：

### 三种动画实体及基本数据类型

动画实体	基本数据类型	声明举例
Viewer	Anvid	Anvid vid;
Macro	Anmid	Anmid mid;
Drawing	Andid	Andid did;

布尔类型（Boolean）被OPNET用于确认返回值为真或假，变量的值可以为OPNET常量OPC\_TRUE或OPC\_FALSE。

基本数据类型	声明举例
Boolean	Boolean bool;

完成代码类型（Compcode）常作为许多KP的返回值，用于确认某一操作是否成功。类型值可以为OPNET常量OPC\_COMPCODE\_SUCCESS 和 OPC\_COMPCODE\_FAILURE。

基本数据类型	声明举例
Compcode	Compcode comp_status;

概率分布类型（Distribution）主要是用来描述随机数与特定数值输出间的概率函数PDF对应关系。概率分布类型一般包括一个枚举表，其中包含了对应关系的编码或者指向对应关系算法。对于基于表的概率分布类型，数据目录可以从一个由PDF编辑器编辑好的概率分布函数模型文件中读出。这些结构主要由分布函数KP来操纵。

基本数据类型	声明举例
Distribution	Distribution* dist_ptr;

事件句柄类型（Event Handle）被用来唯一表述一个待决的仿真事件，这种数据结构主要用来被Intrpt包KP调度对应的事件。请注意事件句柄不是简单的整形或者指针，而是数据结构。因此，请勿将其赋值给整形或者指针变量。

基本数据类型	声明举例
Evhandle	Evhandle evh;

统计量句柄类型（Statistics handle）被用来确认动态生成的全局或局部统计量。这种句柄的数据类型叫做Stathandle，获取它们的唯一途径是通过专门的stat包中一个KP来注册某一统计量。每个统计量在注册后都有一个唯一名称，并且同时生成一个输出向量，以便存储结果。全局统计量被多个仿真实体共享，并且输出向量中的值也由这些实体按照一定加权比例决定。

基本数据类型	声明举例
Stathandle	Stathandle stat_handle;

接口控制信息类型（ICI，Interface Control Information）是与仿真中断相关的结构型数据集，它们是进程间通信的手段，特别是在层与层之间传输信息。这些数据结构由ICI包中的KP来使用。

基本数据类型	声明举例
Ici	Ici* ici_ptr;

联系方式：[life\\_beauty@163.com](mailto:life_beauty@163.com)



链表类型 (List) 是数据元素的集合，储存于双向链中，链表中的数据元素可以是简单的C/C++数据类型，也可能是更加复杂的数据结构。链表可以是多样的，包含了各种不同类型的数据，但通常并不这样使用链表。由于数据可以在任意指定位置上加入和删除，链表可以变得非常巨大。这些数据类型由Prg包中的KP处理。

基本数据类型	声明举例
List	List* list_ptr;

对象标识 (Object Id) 可用来唯一的表示某一仿真对象 (序列号)。使用Objid来作为其数据类型，由Id, Ima, Topo, Pk函数中的KP进行处理。

基本数据类型	声明举例
Objid	Objid objid;

流量导入模型 (TIM, Traffic Import Model) 允许OPNET使用OPNET软件外部的数据，共有两种数据类型为此服务。

基本数据类型	声明举例
Tim_Location_ID	Tim_Location_ID location;
Tim_Data data_element	Tim_Data data_element;

包 (Packet) 是描述数据封装和传输的最基本仿真实体，由Pk包中的KP来处理。

基本数据类型	声明举例
Packet	Packet* pkptr;

随着新数据循环产生和销毁，仿真过程可能需要系统动态分配内存以便于仿真过程中数据的存储，并且需要预定每组数据对象的大小，这时使用OPNET仿真内核提供的池存储功能是最有效的。每组同样大小的数据都被看作是一个池，而OPNET内核可以同时为每个池分配大量内存，这种方式大大优于普通的内存分配。每一个池化的数据内存必须通过调用op\_prg\_pmo\_define()来分配，该函数会返回一个池化内存对象句柄 (Pooled Memory Object Handle) 来唯一标识该池，这种句柄的数据类型是Pmohandle。每个池化数据在创建同时会有一个唯一名称，并且通常会被多个仿真实体共享。

基本数据类型	声明举例
Pmohandle	Pmohandle pmh;

当为仿真调试或者结果分析而创建仿真日志时，日志句柄 (log handle) 对于每个日志入口类别都是必需的。

基本数据类型	声明举例
Log_Handle	Log_Handle config_log_hndl;

一些KP用C/C++的指针作为参数，它们使用一种特定的数据类型—Procedure而不是指针来声明这些参数。

基本数据类型	声明举例
Procedure	Procedure proc;

进程句柄 (Process Handle) 用来唯一标识仿真过程中的每个激活的进程。它们由pro包中的KP来操作，请注意和事件句柄一样，进程句柄不是简单的整形或者指针，而是数



据结构。因此，请勿将其赋值给整形或者指针变量。

基本数据类型	声明举例
Prohandle	Prohandle proh;

路由（Routing，RTE）包是几种为路由设计的数据类型的唯一独占用户。路由数据类型中包括从源结点到目的结点路由中所有结点的ID。Route\_set数据类型在源和目的地地址间所有的路由表，而该路由表实际上指向每个路由的Route数据结构。Topology数据类型是RTE包涉及的所有结点和连接及其相关开销所在的复杂数据库。Route\_link是为topology数据库特定连接所提供的句柄。

路由包的基本数据类型

基本数据类型	声明举例
Route	Route* route_ptr;
Route_Set	Route_Set* rs_ptr;
Topology	Topology* topo_ptr;
Route_Link	Route_Link* rl_ptr;

#### 包拆分重组（SAR）

SAR 句柄用来唯一确定SAR缓存，缓存中存储被拆分的包，并且未来可以把它们重新组合。SAR 缓存由进程调用SAR包中的函数产生。SAR 缓存句柄不能赋予整形或者指针变量。

基本数据类型	声明举例
Sbhandle	Sbhandle sbh;

在标志C支持数据类型之外，OPNET支持一种叫做“Vartype”的数据类型，它并不是实际的数据类型，而是声明的一个关键字，以指代一个核心函数的参数可以是多种类型。但并非说Vartype可以接收无数个参数，一次只能传递一个值。

一般的，声明为Vartype的变量可以接收int，double或者指向数据结构的指针。Vartype\*则说明这是一个指向Vartype变量的指针。一般它可以指向int\*,double\*或者指向“指向数据结构的指针”。

#### 函数栈跟踪

函数栈跟踪对于调试来说是非常有用的方法。因为往往一个错误发生的现场并非是导致错误所在，要找到导致错误之处，必须通过反向跟踪。

所有的OPNET程序，包括仿真，都提供了反向跟踪的能力（操作系统自身的调用不包括在内）。如果错误发生，跟踪结果可以通过op\_vuerr打印出来。

下面的例子是一个打印结果。要注意它包括了一系列函数引用，每个引用都包括名称和参数。函数列表从最高层的main()开始，直到最后出错之处。点线将正确和错误之处区分开来，（错误从vos\_ipc\_signal\_trapper()开始），所以可知，导致flm程序出错的是第26行的mmi\_draw\_icon\_spec()。

```
% op_vuerr
extracted the following messages from 'err_log' file:
<<< Program Fault >>>
* Time: Fri Sep 27 12:52:43 1991
```

\* Program: FLM  
 \* Package: Vos (Virtual Operating System) / Ai (Asynchronous Int.)  
 \* Function: vos\_ai\_manager (int\_type, src\_handle)  
 \* Error: program abort -- segmentation violation  
 \* Function call stack: (builds down)

```
-----
0) main (argc, argv, envp)
1) vg_x_event_loop()
2) vg_x_gfx_input_trapper (xevptr)
3) vos_ai_manager (int_type, src_handle)
4) mmi_event_scoop()
5) mmi_event_direct_handler ()
6) Process_Left ()
7) Env_Button_Left ()
8) mmi_execute_button (env_button)
9) lm_rtes_in_use ()
10) Mmi_Reset ()
11) vg_x_gfx_input_trapper (xevptr)
12) vos_ai_manager (int_type, src_handle)
13) mmi_event_scoop()
14) mmi_event_direct_handler ()
15) Process_Left ()
16) Env_Button_Left ()
17) mmi_execute_button (env_button)
18) lm_dmn_up ()
19) Mmi_Reset ()
20) mmi_reset (do_gfx)
21) lm_redraw ()
22) lm_curmod_draw ()
23) lm_dmn_node_model_draw (lmmpttr, org)
24) lm_dmn_node_draw (mptr, i, erase, org)
25) Mmirel_Draw_Icon (icptr, vppttr, org)
26) mmi_draw_icon_spec (icptr, vppttr, pixop)
.....
27) *vos_ipc_signal_trapper (sig, code, scp)
28) vos_ai_manager (int_type, src_handle)
29) Vos_Error_Print (level, package, error0, error1, error2)
-----
```

除了 OPNET 提供的函数外，用户自定义的函数也一样可以跟踪，事实上，大多数用户都会发现这个功能更多用于跟踪自定义函数。为了能够让自定义函数也能被跟踪，其中必须包含特殊的函数栈跟踪代码。通过在函数入口和出口（入口只有一个，出口可能有多个）插入预处理程序声明代码 FIN,FOUT,FRET（分布对应函数进入点，函数退出点，函数返回值点）来实现。下面是例程：

FIN (<function name & arguments>)

FOUT  
FRET (<return type>)

下面是函数书写的规范：

```

sum_three (a, b, c)
int a, b, c;
{
int sum;
FIN (sum_three (a, b, c))
sum = a + b + c;
FRET (sum)
}

pr_banner (string)
char *string;
{
FIN (pr_banner (string))
printf ("-----\n");
printf ("%s\n", string);
printf ("-----\n");
FOUT
}
    
```

所有 OPNET 提供的函数都按照此规范书写，建议用户自定义函数也遵守此规范，如果那样，即便是嵌套调用都可以跟踪。

### 变量命名限制

一般来说，用户可以随便使用语法正确的命名。但是有一个限制：对 C/C++ 编译器和大多数工作站来说，如果全局/局部变量和一个函数名字相同，对函数的声明和调用实际上变成了和变量相关。所以，当这种情况发生时，程序从该变量的地址开始执行，将导致程序中止。

这个问题并不会影响到状态变量，因为它们单独储存在一个数据结构中。但是对于临时变量和头模块中定义的变量就必须小心，不要与函数重名。

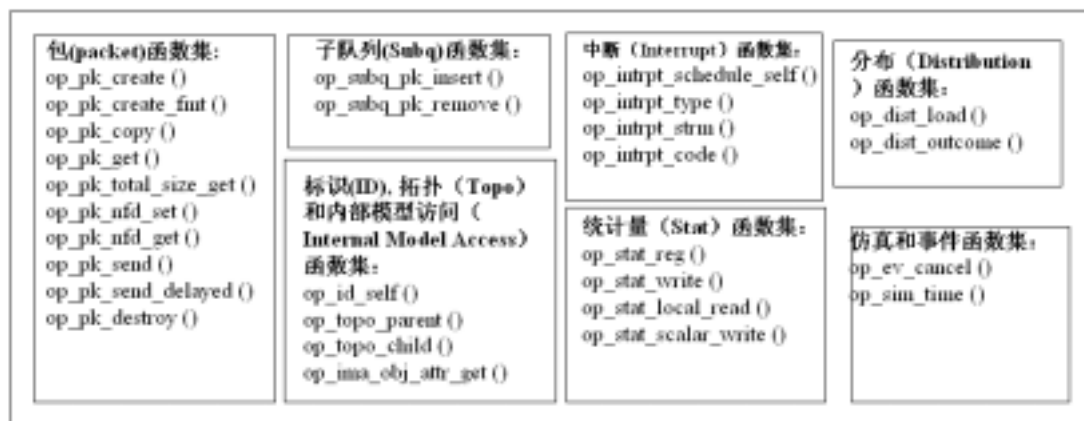
一般来说，并不会因为用户自己定义的变量和自定义函数名冲突，而是自定义变量和一些系统短名称函数冲突。下面是一个可能冲突的函数名列表：更加详尽的列表请参看 UNIX 文档和 WINDOWS\_NT/2000 API 参考。

可能冲突的函数名列表

accept()	kill()	socket()
access()	link()	stat()
audit()	listen()	tell()
bind()	open()	truncate()
clear()	pipe()	unlink()
clock()	poll()	wait()
close()	read()	
connect()	select()	
exit()	send()	
index()	signal()	

### 常用核心函数

OPNET Modeler 提供了 22 个函数，大约 380 多个核心函数。常用的核心函数，大约有二十多个，先熟悉这些函数，就可以开始进行 OPNET 包含 KP 的代码书写了。



常用核心函数

## 包函数

包是 OPNET 仿真中最基本的通信实体，对包进行操作的包函数也是最常使用的函数之一。包函数中包含了对包进行创建，复制，销毁，存取以及获得信息操作的函数。

- 创造、拷贝或者销毁包
  - op\_pk\_create ()
  - op\_pk\_create\_fmt (format\_name)
  - op\_pk\_copy (pkptr)
  - op\_pk\_destroy (pkptr)
- 得到或者发送包
  - op\_pk\_get (instrm\_index)
  - op\_pk\_send (pkptr, outstrm\_index)
  - op\_pk\_send\_delayed ()
- 得到或者设置包内命名字段
  - op\_pk\_nfd\_set (pkptr, fd\_name, value)
  - op\_pk\_nfd\_get (pkptr, fd\_name, value\_ptr)
- 得到包的一定属性
  - op\_pk\_total\_size\_get (pkptr)

<b>op_pk_create_fmt ()</b>			
语法	op_pk_create_fmt (format_name)		
	参数	类型	描述
	format_name	Const char*	即将生成包的类型。
返回值	类型		描述
	Packet*		指向新生成包的指针，如果可恢复错误发生，返回OPC_NIL。
例程	<pre> /* Compose a mac frame from all these elements */ mac_frame_ptr = op_pk_create_fmt ("fddi_mac_fr"); op_pk_nfd_set (mac_frame_ptr, "svc_class", svc_class); op_pk_nfd_set (mac_frame_ptr, "dest_addr", dest_addr); op_pk_nfd_set (mac_frame_ptr, "src_addr", my_address); op_pk_nfd_set (mac_frame_ptr, "info", pdu_ptr); if (svc_class == FDDI_SVC_ASYNC) { op_pk_nfd_set (mac_frame_ptr, "tk_class", req_tk_class); op_pk_nfd_set (mac_frame_ptr, "pri", req_pri); } /* Assign the frame control field... */ op_pk_nfd_set (mac_frame_ptr, "fc", FDDI_FC_FRAME); /* Enqueue the frame at the tail of the queue. */ op_subq_pk_insert (0, mac_frame_ptr, OPC_QPOS_TAIL); </pre>		
功能概述	按照已经定义好的包结构模型生成新包。		
详解	<p>新包由仿真内核生成，在激活此核心函数的处理器或者队列处加入循环。包根据给定格式来建立，所以某些属性可以预先设置。包的最初大小就是预定格式结构大小，并且在初始时刻，附加数据大小为零，以后可通过op_pk_bulk_size_ser()来调整。</p> <p>新建的包将会得到一个唯一的ID，这个ID是在调用这一函数后紧跟的可用ID。包的创造和时间戳以函数调用时的系统时间为准。</p> <p>这个模块是强制串行的。</p>		
开发意图	这一函数提供了按照预设格式创造新包的机制。由于包格式一经设定就不能再更改，所以这种格式化的包可以满足规定格式的通信所需。使用格式化包的好处是每一个字段都由名称来标识，这样就简化了设置时的操作。缺点是所给的字段名字必须和包结构中的字段名相比较，两者相等才有效。		
返回错误值	<ul style="list-style-type: none"> <li>内存分配错误。</li> <li>分段错误，（由错误格式名称参数导致）</li> <li>包格式不可识别</li> <li>非法包格式或字段被赋值，该类型结构字段不可在 创造时赋值。</li> <li>非法包格式或字段被赋值，该类型包字段不可在创造时赋值。</li> </ul>		
相关函数	<ul style="list-style-type: none"> <li><b>op_pk_nfd_set()</b> 给一个格式化包的字段赋值</li> <li><b>op_pk_send()</b> 发送一个包给其他模块</li> <li><b>op_pk_deliver()</b> 发送一个包给其他模块</li> <li><b>op_pk_destroy()</b> 销毁一个不再使用的包</li> </ul>		



<b>op_pk_copy ()</b>			
语法：	<b>op_pk_copy (pkptr)</b>		
	参数	类型	描述
	<i>pkptr</i>	<i>Packet*</i>	指向欲拷贝原始包的指针。
返回值	类型		描述
	<i>Packet*</i>		指向新生成包的指针。
例程	<pre> /* receive a frame from upper layer */ ul_pkptr = op_pk_get (0); /* copy and send to all active network interfaces */ for (i = 0; i &lt; num_interfaces; i++) { if (interface_array [i].active) { cp_pkptr = op_pk_copy (ul_pkptr); op_pk_send (cp_pkptr, i); } } </pre>		
功能概述	生成一个特定原始包的复制品。		
详解	<p>本核心函数产生一个新包，并且把已有原始包的上下文都复制到新包里面。所有格式化和非格式化的包都可以利用这个函数复制，被复制包与原来包具有相同类型。对结构字段（OPC_FIELD_TYPE_STRUCT）的复制涉及到字段相关分配核心函数：op_pk_fd_set()和op_pk_nfd_set()。对包字段（OPC_FIELD_TYPE_PACKET）的复制将导致内含包及其包含字段的递归复制。其他类型的字段都作为普通数据对象复制。对这个核心函数的调用和先调用op_pk_create()，然后调用op_pk_transfer()作用相同。</p> <p>被拷贝的包将会得到一个调用此函数时下一个可用的ID。包的创造和时间戳都以该函数调用时的系统时间为准。新包的时间戳可通过op_pk_stamp()来更改。</p> <p>这个模块是强制串行的。</p>		
开发意图	提供了复制包的机制，主要用于：（1）产生多个相同的包，以供扩散式过程传输使用，（2）备份以便重传。		
返回错误值	<ul style="list-style-type: none"> <li>• 分段错误（由无效包指针或者畸形包/字段数据结构导致）</li> <li>• 总线错误（由进程为包中结构字段分配无效地址导致）</li> <li>• 包指针为空</li> <li>• 包指针指向已销毁包。</li> <li>• 包指针指向不可知包。</li> <li>• 内存分配失败。</li> </ul>		
相关函数	<ul style="list-style-type: none"> <li>• <b>op_pk_destroy()</b> 销毁一个不再使用的包</li> <li>• <b>op_pk_transfer()</b> 传送一个已存在包的上下文给另外一个已存在的包</li> <li>• <b>op_pk_id()</b> 获得一个复制包的包ID</li> <li>• <b>op_pk_create()</b> <b>op_pk_create_fmt()</b> 创造不基于其他包的新包。</li> </ul>		





<b>op_pk_destroy ()</b>			
语法：	<b>op_pk_destroy (pkptr)</b>		
	参数	类型	描述
	<i>pkptr</i>	<i>Packet*</i>	指向欲销毁包的指针。
返回值	类型		描述
	无		
例程	<pre> /* a new packet has arrived; acquire it */ pkptr = op_pk_get (op_intrpt_strm ()); /* insert the new packet at tail of subqueue 0 */ if (op_subq_pk_insert (0, pkptr, OPC_QPOS_TAIL) !=     OPC_QINS_PK) {     /* if the insertion failed, discard the packet */     op_pk_destroy (pkptr); } </pre>		
功能概述	销毁一个特定包，释放其内存。		
详解	<p>当一个包不再被使用时，就可以通过这个核心函数将其销毁，并释放内存以供循环使用。但在销毁时，其相关ICI却不同时销毁，因为ICI可能与其它包或者进程相联系。故ICI需要用op_ici_destroy()单独销毁。</p> <p>op_pk_destroy()会自动将所有字段的内存释放，但如果已经用op_pk_nfd_get()获得其中某个结构字段的内容，则需要用op_prg_mem_free()来释放其结构字段的内存。</p>		
开发意图	<p>提供了将不再需要的包销毁，释放相关内存，以便进行内存循环利用。仿真核心可以自由调度分配释放后的内存。应该在最后一次对该包操作后将其销毁，在这个函数返回后，该包指针就宣告无效。</p>		
返回错误值	<ul style="list-style-type: none"> <li>• 分段错误，（由无效包指针或者畸形包/字段数据结构导致）</li> <li>• 总线错误（由进程为包中结构字段分配无效地址导致）</li> <li>• 包指针为空</li> <li>• 包指针指向已销毁包。</li> <li>• 包指针指向不可知包。</li> <li>• 包指针指向静态包。</li> </ul>		
相关函数	<ul style="list-style-type: none"> <li>• <b>op_pk_create()</b> <b>op_pk_create_fmt()</b> 创造不基于其他包的新包。</li> <li>• <b>op_pk_fd_strip()</b> <b>op_pk_nfd_strip()</b> 从一个包里面溢出一个单独字段。</li> <li>• <b>op_pk_creation_time_get()</b> to 得到一个包的生存时间。</li> </ul>		

<b>op_pk_get()</b>			
语法：	<b>op_pk_get (instrm_index)</b>		
	参数	类型	描述
	instrm_index	int	从环绕模块输入数据流索引。
返回值	类型		描述
	<i>Packet*</i>		从输入流中得到的包指针，如果流中无包，返回OPC_NIL，可以通过使用op_strm_empty()来避免对无包流操作。
例程	<pre> /* a packet is being 'pushed' onto the head of the queue */ pkptr = op_pk_get (op_intrpt_strm ()); /* insert the new packet at head of subqueue 0 */ if (op_subq_pk_insert (0, pkptr, OPC_QPOS_HEAD) !=     OPC_QINS_OK) {     /* if the insertion failed, discard the packet */     op_pk_destroy (pkptr); } </pre>		
功能概述	从给定流中取得一个包，并从流中将其删除。		
详解	<p>这个核心函数可以从流或者远程直接传送的包中得到包指针。这两种传输方式包的获取方式相同，都使用这个函数。</p> <p>被发送或者传递的包都被放入一个输入流中，当它们到来时，将会产生中断，通知接收端接收。如果许多路的“安静”包到达，则会按到达顺序排队，通过这个函数可以得到队首包的指针。（和op_strm_pksize()搭配使用可以获取所有包）</p> <p>这个函数需要进程上下文，所以它只能被进程或者进程调用函数所调用。所有格式化和非格式化的包都可以通过这个核心函数得到。</p> <p>当这个核心函数参数设置成大于已有最大输入流索引时，系统会为其进一步分配一个新流，这个功能可以在无限多的流中获得包。</p> <p>这个核心函数对多线程是安全的。</p>		
开发意图	提供了从周围的模块获得包的机制，这是除了自己创造或者声明包外，进程唯一可以获得包的方式。		
返回错误值	<ul style="list-style-type: none"> <li>核心函数需要进程上下文。</li> <li>内存分配错误</li> <li>输入流索引溢出</li> </ul>		
相关函数	<ul style="list-style-type: none"> <li><b>op_pk_send()</b> 在输出流上将包发送给其他模块</li> <li><b>op_pk_deliver()</b> 将包直接传递给其他模块</li> <li><b>op_pk_send()</b> <b>op_pk_deliver()</b> 可以延时、强制或者“安静”得发送包</li> <li><b>op_strm_empty()</b> <b>op_strm_pksize()</b> 确认一个输入流是否包含包。</li> </ul>		

<b>op_pk_send ()</b>			
语法：	<b>op_pk_send</b> (pkptr, outstrm_index)		
	参数	类型	描述
	<i>pkptr</i>	<i>Packet*</i>	指向相关包得指针。
	<i>outstrm_index</i>	<i>int</i>	输出流的索引值
返回值	类型		描述
	无		
例程	<pre> /* increment the number of attempts made so far */ /* on the current frame */ attempts ++; /* the frame is no longer waiting; notify */ /* the deference process of this transition */ frame_waiting = 0; op_stat_write (FRAME_WAITING_OUTSTAT, frame_waiting); /* record the start time for the transmission attempt */ /* in order to determine if late collisions occur */ /* (these are collisions occurring beyond one slot time) */ frame_start_time = op_sim_time (); /* send frame to bus transmitter */ op_pk_send (tx_frame, LOW_LAYER_OUTPUT_STREAM); </pre>		
功能概述	<p>将给定包通过输出流输出，同时割断原有的本进程和包的所属关系。接收方在发送时刻接收到包。</p>		
详解	<p>这个核心函数提供了将包发送到目的模块的机制。在这一过程中，接收时间与发送时间相同，因此相应的事件会在发送时加入事件列表。如果接收端是进程或者队列，可以通过op_intrpt_type()来确定是否为流中断，如果是，则可以通过op_pk_get()得到该包。</p> <p>一旦包通过这个函数被传递，则原进程与包的所属关系宣告无效，包归属于接收进程，同时阻止了发送方对包的进一步调用。</p>		
开发意图	<p>提供了在模块内部通过数据包流传递包的机制。本核心函数是包传递的最基本函数。其他函数提供如延时、强制、“安静”等包传递功能；还可以在忽视模块之间连接的情况下，用op_pk_deliver()来直接传送包。</p> <p>op_pk_send()允许在处理器或者队列进程之间，以及它们和发射机之间传递。</p>		
返回错误值	<ul style="list-style-type: none"> <li>分段错误，由无效指针或者畸形包引起。</li> <li>包指针为空。</li> <li>包指针指向已销毁包。</li> <li>包指针指向不可知包。</li> <li>包指针指向静态包。</li> <li>在某个对象或者流上包传递失败。</li> <li>核心函数需要进程上下文</li> <li>输出流索引溢出</li> <li>输出流索引未包括</li> </ul>		
相关函数	<ul style="list-style-type: none"> <li><b>op_pk_send_delayed()</b> 在输出流上延时将包发送给其他模块</li> <li><b>op_pk_send_forced()</b> 在输出流上将包发送给其他模块，在接收端立即处理</li> <li><b>op_pk_send_quiet()</b> 在输出流上将包发送给其他模块，不产生接收端中断</li> <li><b>op_pk_deliver()</b> 将包直接传递给其他模块</li> <li><b>op_pk_get()</b> 得到到达包的指针</li> </ul>		



<b>op_pk_send_delayed ()</b>			
语法：	<b>op_pk_send_delayed</b> (pkptr, outstrm_index, delay)		
	参数	类型	描述
	<i>pkptr</i>	<i>Packet*</i>	指向相关包得指针。
	<i>outstrm_index</i>	<i>int</i>	输出流的索引值
	<i>delay</i>	<i>Double</i>	包到达的时延
返回值	类型		描述
	无		
例程	<pre> /* Send the frame once previous transmissions have completed. */ /* Account for propagation delay as well. */ op_pk_send_delayed (pkptr, FDDI_PHY_STRM_OUT, accum_bandwidth +Fddi_Prop_Delay); /* Increment THT emulation variable, and consumed bandwidth accumulator. */ tht_value += tx_time; accum_bandwidth += tx_time; </pre>		
功能概述	将给定包通过输出流输出，同时将包所属关系与本进程割断。接收方在经过一定预定时延后接收到包。		
详解	<p>这个核心函数提供了将包发送到目的模块的机制。接收时间在发送时间的基础上有一定时延，相应的事件在发送时加入事件列表。在发送后到接收前，发送进程可以自由进行各种工作，该包此时已经与发送进程无关。这个核心函数可以用来仿真包传递中的时延。当该事件执行后，接收端会收到流中断，可通过op_pk_get()获得该包。</p> <p>一旦包通过这个函数被传递，则原进程与包的所属关系宣告无效，包归属于接收进程，同时阻止了发送方对包的进一步调用。</p>		
开发意图	<p>提供了在模块内部通过流传递包的机制。本函数是op_pk_send()的衍生函数。其他衍生函数提供如强制、“安静”等包传递功能；还可以在忽视模块之间连接的情况下，用op_pk_deliver()来直接传送包。</p> <p>op_pk_send_delayed()允许在处理器或者队列进程之间，以及它们和发射机之间传递。</p>		
返回错误值	<ul style="list-style-type: none"> <li>• 分段错误，由无效指针或者畸形包引起。</li> <li>• 包指针为空。</li> <li>• 包指针指向已销毁包。</li> <li>• 包指针指向不可知包。</li> <li>• 包指针指向静态包。</li> <li>• 在某个对象或者流上包传递失败。</li> <li>• 核心函数需要进程上下文</li> <li>• 输出流索引溢出</li> <li>• 输出流索引未被包括</li> <li>• 输出时间溢出</li> </ul>		
相关函数	<ul style="list-style-type: none"> <li>• <b>op_pk_send_send()</b> 在输出流上将包发送给其他模块</li> <li>• <b>op_pk_send_forced()</b> 在输出流上将包发送给其他模块，在接收端立即处理</li> <li>• <b>op_pk_send_quiet()</b> 在输出流上将包发送给其他模块，不产生接收端中断</li> <li>• <b>op_pk_deliver()</b> 将包直接传递给其他模块</li> <li>• <b>op_pk_get()</b> 得到到达包的指针</li> </ul>		





<b>op_pk_nfd_set ()</b>			
语法：	<b>op_pk_nfd_set</b> (pkptr, fd_name, value)		
	参数	类型	描述
	<i>pkptr</i>	<i>Packet*</i>	指向相关包得指针。
	<i>fd_name</i>	<i>char*</i>	相关字段的名称
	<i>value</i>	<i>Vartype</i>	赋给相关字段的值
返回值	类型		描述
	<i>Compcode</i>		完成代码用来确认操作是否正确完成，返回值可能为：OPC_COMPCODE_SUCCESS或者OPC_COMPCODE_FAILURE。
例程	<pre> /* create a formatted packet */ f_pkptr = op_pk_create_fmt ("example_pkt"); /* assign integer fields in the packet */ op_pk_nfd_set (f_pkptr, "fd_int_1", 2511); int_value = 5 * 200; op_pk_nfd_set (f_pkptr, "fd_int_2", int_value); /* assign double fields in the packet */ op_pk_nfd_set (f_pkptr, "fd_double_1", 3.1415); db_value = 33.5 / 27.1; op_pk_nfd_set (f_pkptr, "fd_double_2", db_value); /* encapsulate a higher-level packet in the packet */ enc_pkptr = op_pk_create (24); op_pk_nfd_set (f_pkptr, "fd_packet", enc_pkptr); /* send the defined packet out of the processor */ op_pk_send (f_pkptr, OUTSTRM); </pre>		
功能概述	设定给定包的某一字段值，该字段由名称确定。		
详解	<p>这个核心函数中赋值参数类型使用Vartype是因为该值可能为Int，double，指针等，核心函数会根据包内部结构来确定传入值的类型。如果该字段不曾被赋值，核心函数就将据此次赋值来重新确定包的大小。</p> <p>对于integer，double或者包字段，必须确定每一字段的值。当包被封装在字段中后，进程就丧失了对嵌入包的所属权，只有当使用了op_pk_fd_get或者op_pk_nfd_get()将其包取出，这一被封装的包才重新可用。</p>		
开发意图	提供了对包字段赋值的机制。这个函数只能对格式化包使用，否则将出错。因为包已经格式化了，所以赋值时只需考虑值，而不必考虑字段大小等参数。		
返回错误值	<ul style="list-style-type: none"> <li>分段错误，由无效指针或者畸形包引起。</li> <li>包指针为空。</li> <li>包指针指向已销毁包。</li> <li>包指针指向不可知包。</li> <li>给定包中没有这个字段。</li> </ul>		
相关函数	<ul style="list-style-type: none"> <li>使用这个函数的变形来给信息information或者结构structu re字段赋值。</li> <li><b>op_pk_fd_set()</b> 根据字段数字索引，来给其赋值</li> <li><b>op_pk_nfd_get()</b> 从包里面删除一个字段，并得到其大小</li> <li><b>op_pk_nfd_strip()</b> 从包里面删除一个字段，并将其大小从包大小中减去。</li> </ul>		



<b>op_pk_nfd_get ()</b>			
语法：	op_pk_nfd_get(pkptr, fd_name, value_ptr)		
	参数	类型	描述
	<i>pkptr</i>	<i>Packet*</i>	指向相关包得指针。
	<i>fd_name</i>	<i>char*</i>	相关字段的名称
	<i>Value_ptr</i>	<i>Vartype*</i>	指向存放相关字段地址的指针
返回值	类型		描述
	<i>Compcode</i>		完成代码用来确认操作是否正确完成，返回值可能为：OPC_COMPCODE_SUCCESS 或者OPC_COMPCODE_FAILURE.。
例程	<pre> { /* variable declarations */ int i; double d; Packet *pkptr, *enc_pkptr; struct Custom_DS *ds_ptr; /* obtain an arriving packet */ pkptr = op_pk_get (0); /* obtain the simple field types */ op_pk_nfd_get (pkptr, "int_value", &amp;i); op_pk_nfd_get (pkptr, "double_value", &amp;d); printf ("packet contents: integer (%d), double (%g)\n", i, d); /* decapsulate a higher-level packet from the arriving packet */ op_pk_nfd_get (pkptr, "packet_value", &amp;enc_pkptr); op_pk_print (enc_pkptr); /* extract a data structure from a packet field */ op_pk_nfd_get (pkptr, "structure_value", &amp;ds_ptr); op_pk_destroy (pkptr); } </pre>		
功能概述	根据字段名称，读取给定包中一个字段的值，该字段属性将自动变为未设置。		
详解	<p>这个核心函数支持所有类型（ integer ， double ， packet information ， structure ）的字段。简单的类型（ integer ， double ）直接将值复制到相应字段，所以它们所在字段属性也不受这个函数影响，仍然保持为已设置，包的大小也不受影响。</p> <p>Packet ， structure字段也将它们相关的值复制进这个包（如：指向它们的指针）。然而，当调用这个函数时，它们就和包剥离开来；它们所在字段属性变为未设置，也不能通过op_pk_nfd_get()来得到，和包大小不再有任何关系。</p> <p>读取信息字段值并无任何明确目的，因为信息字段本来就没有确定的值，但是读取后该字段就被设成“unspecified”；读取值被赋给一个字符数组（数组大小必须能够容纳其值）。</p>		
开发意图	提供了读取包字段值的机制。这个函数对于利用包中携带的信息，在不同模块进程间通信十分重要。一般在包到达目的端时调用这个函数。		
返回错误值	<ul style="list-style-type: none"> <li>分段错误，由无效指针或者畸形包引起。</li> <li>包指针为空。</li> <li>包指针指向已销毁包。</li> <li>包指针指向不可知包。</li> </ul>		
相关函数	<ul style="list-style-type: none"> <li><b>op_pk_nfd_set()</b> 来给字段赋值。</li> <li><b>op_pk_fd_get()</b> 根据数字索引，来获取字段的值。</li> <li><b>op_pk_nfd_type()</b> 获取字段的类型。</li> <li><b>op_pk_nfd_size()</b> 获取字段大小。</li> <li><b>op_pk_nfd_is_set()</b> 确认字段是否已设置。</li> </ul>		

<b>op_pk_total_size_get ()</b>			
语法：	<b>op_pk_total_size_get (pkptr)</b>		
	参数	类型	描述
	<i>pkptr</i>	<i>Packet*</i>	指向相关包得指针。
返回值	类型		描述
	<i>Double</i>		包的总大小。
例程	<pre> /* acquire the packet */ llc_pkptr = op_pk_get (HIGH_LAYER_INPUT_STREAM); /* llc data larger than the maximum acceptable size will */ /* cause an error and the data will be discarded. */ if (op_pk_total_size_get (llc_pkptr) &gt; MAX_DATA_SIZE) { /* compose and display the error message */ sprintf (err_str, "large packet with (%g) bits", op_pk_total_size_get (llc_pkptr)); op_sim_message ("ethernet mac layer discarding excessively", err_str); /* throw away the packet and return from invocation. */ op_pk_destroy (llc_pkptr); FOUT } </pre>		
功能概述	得到给定包的总大小。		
详解	<p>包的总大小包括每个字段大小和附加字段的大小。包的大小可以通过修改附加字段大小，设置/解设置包的普通字段，以及设置修改字段大小来直接设置。格式化和非格式化的包都可以通过这个函数处理。</p> <p>这个函数可以用于多线程应用。</p>		
开发意图	提供了获取包产生时间的机制，一般都用于计算包传输时延。和 <i>op_pk_creation_mod_get()</i> 结合使用，可以比较来自不同源地址的端到端时延。		
返回错误值	<ul style="list-style-type: none"> <li>分段错误，由无效指针或者畸形包引起。</li> <li>包指针为空。</li> <li>包指针指向已销毁包。</li> <li>包指针指向不可知包。</li> </ul>		
相关函数	<ul style="list-style-type: none"> <li><b>op_pk_total_size_set()</b> 设置包的总大小。</li> <li><b>op_pk_bulk_size_set()</b> 只设置包的附加字段大小。</li> <li><b>op_pk_bulk_size_get()</b> 只得到包的附加字段大小。</li> <li><b>op_pk_fd_set()</b> 修改包的一个字段大小（及其值）</li> <li><b>op_pk_fd_strip()</b> <b>op_pk_nfd_strip()</b> 将一个字段从包中剥离，从而将其大小从总大小中减去。</li> </ul>		

## 队列函数

subq 函数中的函数是用于在队列的进程模型中的。每个子队列都是独立的相互连接的包。

op_subq_pk_insert ()			
语法：	op_subq_pk_insert (subq_index, pkptr, pos_index)		
	参数	类型	描述
	subq_index	Int	相关子队列的索引。（该索引从零开始，以一递增。没有子队列的上限。除了直接使用子队列绝对索引外，还可以使用抽象队列索引，专门指代具有某一特性的队列，如：最大的队列）
	pkptr	Packet*	指向相关包的指针。
	pos_index	Int	在子对列中，将要插入相关包的位置索引。
返回值	类型		描述
	Int		标明是否插入成功的返回代码： OPC_QINS_OK, OPC_QINS_FAIL, OPC_QINS_PK_ERROR, OPC_QINS_SEL_ERROR。
例程	<pre>/* Compose a mac frame from all these elements */ mac_frame_ptr = op_pk_create_fmt ("fddi_mac_fr"); op_pk_nfd_set (mac_frame_ptr, "svc_class", svc_class); op_pk_nfd_set (mac_frame_ptr, "dest_addr", dest_addr); op_pk_nfd_set (mac_frame_ptr, "src_addr", my_address); op_pk_nfd_set (mac_frame_ptr, "info", pdu_ptr); if (svc_class == FDDI_SVC_ASYNC) { op_pk_nfd_set (mac_frame_ptr, "tk_class", req_tk_class); op_pk_nfd_set (mac_frame_ptr, "pri", req_pri); } /* Assign the frame control field... */ op_pk_nfd_set (mac_frame_ptr, "fc", FDDI_FC_FRAME); /* Enqueue the frame at the tail of the queue. */ op_subq_pk_insert (0, mac_frame_ptr, OPC_QPOS_TAIL);</pre>		
功能概述	将包插入给定队列给定位置。		
详解	<p>如果插入位置参数不是系统可以识别的OPC_QPOS_PRIO, OPC_QPOS_HEAD, OPC_QPOS_TAIL，并且该值小于0，则直接将包插入到队列头；如果值大于队列长度，则直接插入队列尾。</p> <p>可通过op_subq_pk_insert()向空的子队列中插入新的包，通过op_subq_pk_remove(), op_subq_flush(), 或op_q_flush()移除包。由于子队列有自己容量的上限，这个核心函数可能无法完成其功能。此时，该包状态不受影响，进程对其应该具有相应异常处理能力，如：直接发送给目的端或者销毁。</p> <p>这个函数使用强制中断。</p>		
开发意图	提供了唯一的将包插入子队列的机制,位置索引参数使得可以随意设置包插入位置，既可以是真实数字，也可以是确定包与其他包关系的预设值。		
返回错误值	<ul style="list-style-type: none"> <li>核心函数需要进程上下文。</li> <li>包指针为空。</li> <li>包指针指向已销毁包。</li> <li>包指针指向静态包。</li> <li>分段错误（由畸形包导致）</li> <li>子队列选择标志不可识别</li> <li>包已经插入该队列。</li> </ul> <p>注意：常数 OPC_QINS_FAIL 的返回意味着子队列或者队列没有足够空间，但这并不是错误。</p>		





<b>op_subq_pk_remove ()</b>			
语法：	<b>op_subq_pk_remove</b> (subq_index, pos_index)		
	参数	类型	描述
	subq_index	Int	相关子队列的索引。
	pos_index	Int	在子对列中，将要移出的包的位置索引。
返回值	类型		描述
	Packet*		指针指向移出的数据包，如果发生了可恢复得错误，则返回OPC_NIL。
例程	<pre> /* a request has been made to forward a packet */ /* over one of the queue's output streams */ /* the packet will be pulled from the subqueue */ /* with the same index as the output stream */ /* determine which subqueue is being accessed */ subq_index = op_intrpt_code (); /* check if it is empty */ if (op_subq_empty (subq_index) == OPC_FALSE) { /* access the first packet in the subqueue */ pkptr = op_subq_pk_remove (subq_index, OPC_QPOS_HEAD); /* forward it to the destination over requested stream */ /* use 'quiet' mode to avoid causing a stream interrupt. */ op_pk_send_quiet (pkptr, subq_index); } </pre>		
功能概述	获得队列中指向某一个包的指针，并把数据包移出。		
详解	<p>该函数只能通过队列模块上运行的进程来唤醒，从处理器和管道阶段唤醒均会导致错误。</p> <p>可通过op_subq_pk_insert()向空的子队列中插入新的包，通过op_subq_pk_remove(), op_subq_flush(), 或op_q_flush()移出。</p> <p>如果位置索引为负，或是超过了队列中包的数目，该函数无法完成其功能。此时，将发生错误，并返回常数：OPC_NIL。</p> <p>这个函数使用强制中断。</p>		
开发意图	<p>可实现包从子队列中无破坏性的移出，其它的KP，如op_subq_flush()和op_q_flush()均会对队列本身造成破坏。位置索引参数使得可以随意设置移出包的位置，既可以是真实数字，也可以是确定包与其他包关系的预设值。</p>		
返回错误值	<ul style="list-style-type: none"> <li>• 核心函数需要进程上下文。</li> <li>• 包指针为空。</li> <li>• 包指针指向已销毁包。</li> <li>• 包指针指向静态包。</li> <li>• 分段错误（由畸形包导致）</li> <li>• 子队列选择标志不可识别</li> <li>• 包已经插入该队列。</li> </ul> <p>注意：常数 OPC_QINS_FAIL 的返回意味着子队列或者队列没有足够空间，但是这并不是错误。</p>		
相关函数	<ul style="list-style-type: none"> <li>• <b>op_subq_pk_access()</b> 从子队列中得到包，而不将其移除。</li> <li>• <b>op_subq_pk_remove()</b> 从子对列中得到包，并溢出。</li> <li>• <b>op_subq_pk_swap()</b> 将子队列中两个包换位。</li> <li>• <b>op_subq_flush()</b> 将子队列中所有包移除。</li> </ul>		

## 标识，拓扑和内部模型访问函数

仿真中所有的对象都有标识与之对应。大多数对仿真对象进行操作的核心函数都需要知道操作对象的标识。

拓扑函数是用来获得模型中对象的数量以及标识。

内部模型访问函数主要用于对模型以及对象的属性进行操作和维护。

<b>op_id_self ()</b>			
语法：	<b>op_id_self ()</b>		
	参数	类型	描述
	无		
返回值	类型		描述
	<i>Objid</i>		本处理器或队列的对象ID。如果这个核心函数从一个非进程上下文（如：非处理器或者队列的上下文）运行，则将返回 OPC_OBJID_INVALID
例程	<pre>/* initially the server is idle */ server_busy = 0; /* get queue module's own object ID */ own_id = op_id_self (); /* get assigned value of server processing rate */ op_ima_obj_attr_get (own_id, "service_rate", &amp;service_rate);</pre>		
功能概述	得到包含本进程处理器或者队列的对象ID。		
详解	<p>如果这个核心函数从一个非进程上下文（如：非处理器或者队列的上下文）运行，则将返回OPC_OBJID_INVALID。通过 op_intrpt_schedule_call()接口安排好的过程调用，将从调用它们的进程中继承上下文，因此，通过op_id_self()，它们可以返回有效值。然而，pipeline阶段并不包含进程上下文，所以它们调用这个函数将会返回 OPC_OBJID_INVALID。</p> <p>可用于多线程的应用。</p>		
开发意图	提供了得到自身ID的机制，之后仿真可以通过 op_ima_obj_attr_get()来获取对象的属性。通常和op_topo_parent()结合使用，获得上一级的对象ID。		
返回错误值	无		
相关函数	<ul style="list-style-type: none"> <li>• <b>op_topo_parent()</b> 得到父对象的ID</li> <li>• <b>op_topo_child()</b> 得到子对象的ID</li> <li>• <b>op_ima_obj_attr_get()</b> 得到某个对象属性值</li> <li>• <b>op_ima_obj_attr_set()</b> 设置某个对象属性值</li> </ul>		

<b>op_topo_parent ()</b>
--------------------------

语法：	<b>op_topo_parent (child_objid)</b>		
	参数	类型	描述
	<i>child_objid</i>	<i>Objid</i>	子对象的对象ID。（对象ID可以通过多个核心函数得到： <i>op_id_self()</i> , <i>op_id_from_sysid()</i> , <i>op_id_from_userid()</i> , 和 <i>op_id_from_name()</i> . 对象ID也可以通过拓扑包中的核心函数 <i>op_topo_parent()</i> , <i>op_topo_child()</i> , <i>op_topo_assoc()</i> , 和 <i>op_topo_connect()</i> 得到）
返回值	类型		描述
	<i>Objid</i>		父对象的对象ID。如果子对象已经是最高一级子网，则返回OPC_OBJID_INVALID。如果可恢复错误发生，将会返回OPC_OBJID_INVALID
例程	<pre> /* Obtain self ID */ myid = op_id_self (); /* Obtain the attribute values. */ op_ima_obj_attr_get (myid, "address", &amp;my_address); op_ima_obj_attr_get (myid, "average_interarrival_time", &amp;ia_time); op_ima_obj_attr_get (myid, "average_service_time", &amp;service_time); op_ima_obj_attr_get (myid, "ack_service_time", &amp;ack_service_time); /* Obtain the parent object id and name. */ ppid = op_topo_parent (myid); op_ima_obj_attr_get (ppid, "name", name); </pre>		
功能概述	得到特定对象父对象的对象ID。		
开发意图	<p>当一个低层对象的对象ID已知，并且需要对其高一层对象进行操作时，本核心函数提供了向上提升层次的机制。一般的，这个核心函数用于得到包含该进程处理器或队列的结点（或者再进一步：子网）的对象ID。得到了这些对象ID后，就可以通过<i>op_ima_obj_attr_get()</i>来得到它们的属性值。</p>		
返回错误值	<ul style="list-style-type: none"> <li>对象ID溢出</li> <li>对象ID指向受限对象</li> </ul>		
相关函数	<ul style="list-style-type: none"> <li><b><i>op_id_self()</i></b>得到包含该进程的处理器或者队列对象ID。</li> <li><b><i>op_topo_child()</i></b> 得到另外一个对象子对象的对象ID</li> <li><b><i>op_ima_obj_attr_get()</i></b> 得到对象属性值</li> <li><b><i>op_ima_obj_attr_set()</i></b> 设置对象属性值</li> </ul>		

<b>op_topo_child ()</b>			
语法：	<b>op_topo_child</b> (parent_objid, child_type, child_index)		
	参数	类型	描述
	parent_objid	Objid	相关父对象的对象ID。(对象ID可以通过多个核心函数得到：op_id_self(), op_id_from_sysid(), op_id_from_userid(), 和 op_id_from_name(). 也可以通过拓扑包中的核心函数op_topo_parent(), op_topo_child(), op_topo_assoc(), 和 op_topo_connect()得到)
	child_type	Int	相关子对象的类型
	child_index	Int	相关子对象的数字索引(子索引从零开始，最大到该类型子对象数目减一)
返回值	类型		描述
	Objid		特定子对象的对象ID，如果发生可恢复错误，将返回特定值OPC_OBJID_INVALID
例程	<pre> /* Set unique user ID's of the nodes in parent subnet. */ /* First obtain the subnet object ID. */ node_id = op_topo_parent (op_id_self ()); subnet_id = op_topo_parent (node_id); /* Determine the number of children that are nodes. */ node_count = op_topo_child_count (subnet_id, OPC_OBJMTYPE_NODE); /* Loop through the child nodes and set unique user ID's. */ for (i = 0; i &lt; node_count; i++) { node_id = op_topo_child (subnet_id, OPC_OBJMTYPE_NODE, i); op_ima_obj_attr_set (node_id, "user id", i); } </pre>		
功能概述	获得一个拥有要求类型和索引子对象的对象ID		
详解	<p>子对象包含在父对象之中，子网包含子对象有：子网、结点，链路和接头；结点包含子对象有：模块、流和统计线；队列包含子对象有：子队列复合属性；接收机包含子对象有：接收信道复合属性对象；发射机包含子对象有：发射信道复合属性对象；复合属性包含子对象有：类属子对象、子队列、接收信道、发射信道。</p> <p>通过以下两个参数来确定子对象：子对象类型和子对象索引。</p> <p>对本函数成功的调用可以实现层次的下降，在下面这个例程里面，共有5级对象层次，顶层的子网(对象ID为零)可以通过下述步骤一直转换到子队列的对象ID：</p> <pre> subnet_objid = op_topo_child (top_subnet_objid, OPC_OBJTYPE_SUBNET_FIX, 0); node_objid = op_topo_child (subnet_objid, OPC_OBJTYPE_NDFIX, 0); </pre>		

	<pre>queue_objid = op_topo_child (node_objid, OPC_OBJTYPE_QUEUE, 0); subq_comp_attr_objid = op_topo_child (queue_objid, OPC_OBJTYPE_COMP, 0); subqueue_objid = op_topo_child (subq_comp_attr_objid, OPC_OBJTYPE_SUBQ, 0);</pre> <p>注意：subq_comp_attr_objid是系统内建复杂属性，对于用户自己定义的复杂属性，该语句改为：</p> <pre>generic_objid = op_topo_child (comp_attr_objid, OPC_OBJTYPE_GENERIC, 0);</pre> <p>路径不包含任何子对象，也不属于任何子网，所以如果用此核心函数去获取与路径相关子对象的话，将会返回空句柄。</p> <p>本函数使用强制串行。</p>
开发意图	<p>当已知高层某一对象的ID，并且需要对其低层对象进行操作时，提供了从对象树高层向下下降的机制。这个核心函数经常被用来获得某个发射/接收机中特定信道的对象ID。以便于通过op_ima_obj_attr_get() 或 op_ima_obj_attr_set() 来设置或者读取信道参数。一个可选的本函数使用方法是用来获得某不知名结点内某模块的ID，不过一般这种情况都是使用op_id_from_name()来直接获得给定名字的子对象ID。这个函数也经常用来获取用户定义的复杂属性和子对象ID。</p>
返回错误值	<ul style="list-style-type: none"> <li>• 对象索引溢出</li> <li>• 对象ID溢出</li> <li>• 对象ID指向受限对象</li> <li>• 对象元类型(&lt;child_type&gt;)不可识别</li> <li>• 对象无子对象</li> <li>• 特定类型对象索引溢出</li> </ul>
相关函数	<ul style="list-style-type: none"> <li>• <b>op_topo_parent()</b> 获得一个对象的父对象的对象ID</li> <li>• <b>op_topo_object_count()</b> 获得特定类型对象数目</li> <li>• <b>op_ima_obj_attr_get()</b> 获得一个对象的属性值</li> <li>• <b>op_ima_obj_attr_set()</b> 设置一个对象的属性值</li> <li>• <b>op_topo_object()</b> 获得一个对象的对象ID</li> <li>• <b>op_id_from_name()</b> 通过对象名字获得它的对象ID</li> </ul>

<b>op_ima_obj_attr_get()</b>			
语法：	<b>op_ima_obj_attr_get(objid, attr_name, value_ptr)</b>		
	参数	类型	描述
	Objid	objid	相关对象的Object ID ( Object ID可以从Topo和ID包中的一些KP中获得，例如： op_topo_parent(),op_id_self(),op_topo_child(), op_id_from_sysid(), op_id_from_userid(), op_id_from_name()等。 )

	<i>attr_name</i>	<i>const char*</i>	相关属性名称( 这个属性必须是特定对象预先定义好的，否则将产生错误 )
	<i>value_ptr</i>	<i>Vartype*</i>	指向存储该属性信息变量的指针。( 这个参数需要在程序中通过 <i>Vartype*</i> 先行定义，可以是 string,int *,double *, 或者指向复杂对象的 objid *。如果是 string 型，则该数组长度必须能够容纳获得的属性值。如果属性类型是 toggle，返回值是 OPC_BOOLINT_ENABLED OPC_BOOLINT_DISABLED。
返回值	类型		描述
	Compcode		完成代码用来确认操作是否正确完成 ,返回值可能为：OPC_COMPCODE_SUCCESS 或者 OPC_COMPCODE_FAILURE。
例程	<pre> /* determine id of own processor to use in finding attrs */ my_id = op_id_self (); /* determine address range for uniform destination assignment */ op_ima_obj_attr_get (my_id, "low dest address", &amp;low_dest_addr); op_ima_obj_attr_get (my_id, "high dest address", &amp;high_dest_addr); /* determine object id of connected 'mac' layer processor */ mac_objid = op_topo_assoc (my_id, OPC_TOPO_ASSOC_OUT, OPC_OBJTYPE_QPS, MAC_LAYER_OUT_STREAM); /* determine the address assigned to it */ /* which is also the address of this station */ op_ima_obj_attr_get (mac_objid, "station_address", &amp;station_addr); /* setup a distribution for generation of addresses */ dest_dist_ptr = op_dist_load ("uniform_int", low_dest_addr, high_dest_addr); Additional example of obtaining and using a character string variable: /* Obtain the name of the process (the "process model" attribute of the module. */ /* proc_model_name was previously declared in the temporary variable block as: */ /* char proc_model_name [128]; */ op_ima_obj_attr_get (op_id_self (), "process model", proc_model_ </pre>		
功能概述	得到特定对象的某一属性值。		
详解	对象属性可以从网络域对象（子网，结点，连接），结点域对象（模块）或者在使用网络、结点、连接等前已经定义的复杂属性中取得。特别要注意的是，那些被提升的属性可以从包含该属性定义所在对象的高层对象中获得（例如：从处理器根进程定义并提升的属性可以在包含处理器的子网中获得）。对于那些提升的属性，需要加一个前缀（前缀.属性）才能得		

	<p>到。比如，我们在结点处想得到其中一个进程的某属性，需要在该属性前加上包含该进程模块的名字和一个“.”。如果不考虑属性所在层次的话，将会找到同名属性。</p> <p>并不是所有的属性值都可以这样得到，任何将Simulation/Access 字段设成N/A 的属性都无法通过op_ima_obj_attr_get()获得</p>
开发意图	<p>提供在仿真过程中动态获得属性的一种方法。处理器和队列可由此得到它们自身属性值或者网络中其他结点属性值。比如：天线方向或者发射信道的属性可以通过这个函数得到。当和op_ima_obj_attr_set()结合使用时，实际上提供了一种在进程间不使用包、接口控制信息或中断的通信机制。</p>
返回错误值	<ul style="list-style-type: none"> <li>• 分段错误（由无效指针，指向错误内存指针或者无效地址引起）</li> <li>• 属性名字(&lt;attr_name&gt;) 对该对象不可识别</li> <li>• 对象 ID (&lt;objid&gt;) 溢出</li> <li>• 对象 ID (&lt;objid&gt;) 指向一个受限对象。（由引用对象属于某个受保护模型引起）</li> </ul>
相关函数	<ul style="list-style-type: none"> <li>• op_ima_obj_attr_set()：设置对象属性值</li> <li>• op_ima_sim_attr_get()：得到仿真属性值</li> <li>• op_ima_obj_svar_get()：得到进程状态变量值</li> <li>• op_ima_obj_command()：发送一个命令给对象</li> <li>• op_id_self()：得到本处理器或者队列的对象ID</li> <li>• op_id_from_name()：得到特定名称对象的对象ID</li> </ul>

## 中断函数

<b>op_intrpt_schedule_self ()</b>			
语法：	<b>op_intrpt_schedule_self (time, code)</b>		
	参数	类型	描述
	<i>Time</i>	Double	调度中断发生时间。（绝对时间，不是从现在时刻算起的相对时间）
	<i>Code</i>	Int	与中断相联系，用户自定义的数值代码。（该代码完全由用户自行定义，以便于在接收时进行认证，通过op_intrpt_code()接收）
返回值	类型		描述
	<i>Evhandle</i>		被调度中断的事件句柄。这个返回值可以存在状态变量中，以便于当使用op_ev_cancel()取消时验证用。
例程	<pre>/* get a handle on packet at head of subqueue 0 */ /* (this does not remove the packet) */ pkptr = op_subq_pk_access (0, OPC_QPOS_HEAD); /* determine the packet's length (in bits) */ pk_len = op_pk_total_size_get (pkptr); /* determine the time required to complete */</pre>		



	<pre> /* service of the packet */ pk_svc_time = pk_len / service_rate; /* schedule an interrupt for this process */ /* for the time when service ends. */ op_intrpt_schedule_self (op_sim_time () + pk_svc_time, 0); /* the server is now busy. */ server_busy = 1; </pre>
功能概述	在特定时间为激活进程调度一个中断。
详解	<p>调用这个函数，将在事件表中插入一个事件，该事件代表了在特定时间发生的自中断。只要该事件一被调度，这个核心函数就将控制权交回给激活进程。在这段期间，激活进程可以自由的进行任何操作。这个中断将在所有比其级别高的中断之后按时发生。当所有排在它前面的事件都已完成，这个事件就排到了事件表的前头，在设定时间进行。该事件将导致对激活进程的自中断。进程可以通过op_intrpt_code得到其设定代码。</p> <p>当自中断被用来表示时间溢出时，经常需要取消该中断。比如：一个计时器需要重起，因为已经收到了ACK信号。）为了能够取消中断，需要我们将此核心函数返回的事件句柄储存于状态变量中，以便op_ev_cancel()使用。例如：</p> <pre> state variable declaration: Evhandle \evh; self interrupt scheduling: evh = op_intrpt_schedule_self (sch_time, 5); self interrupt cancelling: op_ev_cancel (evh); </pre> <p>这个核心函数采用强制串行。</p>
开发意图	提供了安排未来自中断的机制。这一般用来限制进程某种状态的持续时间。对于模拟超时现象，当前状态一直在等待某个包的到来，而等待时间结束时，自中断就会发生。对于仿真处理时延，当前状态表示正在处理，而“处理结束”自中断的发生将意味着工作的结束，以及资源的释放。在时间间隔结束时，进程一般进入闲置状态，或者重新处理其他等待处理的事务。
返回错误值	<ul style="list-style-type: none"> <li>• 核心函数需要进程上下文（如果从处理器或者队列调用）。</li> <li>• 内存分配失败。</li> <li>• 给定时间小于当前时间</li> <li>• 给定时间值非法</li> </ul>
相关函数	<ul style="list-style-type: none"> <li>• <b>op_intrpt_schedule_remote()</b> 调度一个远程中断</li> <li>• <b>op_intrpt_force_remote()</b> 强制执行远程中断</li> <li>• <b>op_intrpt_schedule_call()</b> 调度一个过程中断</li> <li>• <b>op_ev_cancel()</b> 取消一个即将到来的自/远程/过程中断</li> <li>• <b>op_intrpt_code()</b> 得到与中断相联系的代码</li> </ul>

### op\_intrpt\_type ()



语法：	<b>op_intrpt_type ()</b>		
	参数	类型	描述
	无		
返回值	类型		描述
	<i>int</i>		当前中断的类型。
例程	<pre> type = op_intrpt_type (); switch (type) { case (OPC_INTRPT_STRM): /* handle stream interrupts... */ break; case (OPC_INTRPT_SELF): /* handle self interrupts... */ break; case (OPC_INTRPT_STAT): /* handle statistic wire interrupts... */ break; default: op_sim_end ("err: link mac: unexpected intrpt", "", "", ""); } </pre>		
功能概述	得到激活进程当前中断的类型。		
详解	<p>这个核心函数返回一个可以与以下类型常数相比较的值：</p> <p>OPC_INTRPT_FAIL 结点或链路失败中断</p> <p>OPC_INTRPT_RECOVER 结点或链路恢复中断</p> <p>OPC_INTRPT_PROCEDURE 过程中断</p> <p>OPC_INTRPT_SELF 自中断</p> <p>OPC_INTRPT_STRM 流中断</p> <p>OPC_INTRPT_REGULAR 常规中断</p> <p>OPC_INTRPT_STAT 统计中断</p> <p>OPC_INTRPT_REMOTE 远程中断</p> <p>OPC_INTRPT_BEGSIM 仿真起始中断</p> <p>OPC_INTRPT_ENDSIM 仿真结束中断</p> <p>OPC_INTRPT_ACCESS 访问中断</p> <p>OPC_INTRPT_PROCESS 进程中断</p> <p>OPC_INTRPT_MCAST 广播中断</p> <p>这个核心函数使用强制中断。</p>		
开发意图	使激活进程了解是何原因使其被“唤醒”。这个核心函数是非常基本的，因为几乎每个进程都要使用它。许多拥有多个分支状态的进程模型都要保留这个返回值，以便选择下一状态。		
返回错误值	无		

相关函数	<ul style="list-style-type: none"> <li>• <b>op_intrpt_code()</b> 得到与中断相联系的代码</li> <li>• <b>op_intrpt_strm()</b> 得到和当前中断相关的流索引</li> <li>• <b>op_intrpt_stat()</b> 得到和当前统计中断相关的统计索引</li> <li>• <b>op_intrpt_ici()</b> 得到和当前中断相关的接口控制信息</li> </ul>
------	---

<b>op_intrpt_strm ()</b>			
语法：	<b>op_intrpt_strm ()</b>		
	参数	类型	描述
	无		
返回值	类型		描述
	<i>int</i>		当前中断的流索引。当一个包通过 op_pk_deliver()或者op_pk_send()传入，或者通过op_strm_access()读取时，该值被明确设定。
例程	<pre> /* if the event was an arrival from the physical */ /* layer, accept the packet and decapsulate it */ if (op_intrpt_type() == OPC_INTRPT_STRM &amp;&amp;     op_intrpt_strm () == LOW_LAYER_INPUT_STREAM) {     eth_mac_phys_pk_accept (); } /* otherwise, if the event is an arrival from the */ /* higher layer, accept the packet and enqueue it */ else if (op_intrpt_type () == OPC_INTRPT_STRM &amp;&amp;     op_intrpt_strm () == HIGH_LAYER_INPUT_STREAM) {     eth_mac_llc_pk_accept (); } </pre>		
功能概述	得到激活进程当前中断相关的流索引。		
详解	<p>当发生流中断后，调用这个函数只返回一个有效值。和流有关的两个中断是：流中断和访问中断。前者相关的流是包到来的输入流，后者相关的流是访问的输出流。</p> <p>这个核心函数采用强制串行。</p>		
开发意图	通过该核心函数，进程可以确定当前是哪一个输入数据包流。（这个信息在接下来通过op_pk_get()接收包时可以用来确定确切的流索引），对于访问中断，该函数可以确定选择哪一个输出流。		
返回错误值	<ul style="list-style-type: none"> <li>• 核心函数需要进程上下文</li> <li>• 当前中断不是流中断或者访问中断</li> </ul>		
相关	• <b>op_intrpt_code()</b> 得到与中断相联系的代码		

函数	<ul style="list-style-type: none"> <li>• <b>op_intrpt_stat()</b> 得到和当前统计中断相关的统计索引</li> <li>• <b>op_intrpt_type()</b> 获得当前中断类型</li> <li>• <b>op_pk_send...()</b> <b>op_pk_deliver...()</b> 发送或者传递包给一个模块的输入流。</li> <li>• <b>op_strm_access()</b> 访问一个输出流</li> </ul>
----	---

<b>op_intrpt_code ()</b>			
语法：	<b>op_intrpt_code ()</b>		
	参数	类型	描述
	无		
返回值	类型		描述
	<i>Int</i>		和当前中断相关的数字代码。
例程	<pre> switch (op_intrpt_type()) { case OPC_INTRPT_SELF: { /* Test if the timer that expired is 'T13' */ if (op_intrpt_code () == X25C_T13) { /* If this is the first expiration, retransmit the clear indication */ /* to the DTE and increment the expiration counter. */ if (retry_count [X25C_T13] == 0) { x25_pk_send_local (chan_vars-&gt;data_link, chan_vars-&gt;number, X25_GFI_BASE, X25C_PT_CLEAR_INDICATION, X25C_RC_LOC_PROC_ERROR, 50); retry_count [X25C_T13]++; x25_dce_trace_msg ("Timer T13 expired; resend clear indication to DTE"); } } else { } } </pre>		
功能概述	得到和当前中断相关的数值代码。		
详解	<p>只有当一个中断有相关的数值代码时，此函数才有效。自中断，远程中断和进程中断支持数值代码。</p> <p>这个核心函数使用强制串行。</p>		

开发意图	便于从中断中取得用户自定义的代码，从而可以确认该中断的目的。当有几个不同类型的中断存在，这就非常必要。例如：多种超时自中断。
返回错误值	无
相关函数	<ul style="list-style-type: none"> <li>• <b><i>op_intrpt_type()</i></b> 确定当前接口控制信息类型</li> <li>• <b><i>op_intrpt_strm()</i></b> 获得当前中断的相关流索引</li> <li>• <b><i>op_intrpt_stat()</i></b> 获得当前统计中断相关的输入统计索引</li> <li>• <b><i>op_intrpt_source()</i></b> 获得中断源</li> <li>• <b><i>op_intrpt_ici()</i></b> 获得中断携带接口控制信息</li> </ul>

## 统计量函数

	<b>op_stat_reg ()</b>		
语 法：	<b>op_stat_reg (stat_name, stat_index, type)</b>		
	参数	类型	描述
	<i>group_dot_stat_name</i>	<i>const char*</i>	统计量所在组及其名字，中间用“.”隔开。
	<i>stat_index</i>	<i>Int</i>	相关统计量的数字索引。若它没有维数，使用 OPC_STAT_INDEX_NONE,
	<i>type</i>	<i>Int</i>	OPC_STAT_GOLBAL：全局统计量 OPC_STAT_LOCAL：局部统计量
返回值	类型		描述
	<i>Stathandle</i>		稍后在 <i>op_stat_write()</i> 中使用的进程句柄。
例程	<pre> /* register HTTP-related statistics. */ app_mgr_state_ptr-&gt;HTTP_NUM_PAGES = op_stat_reg("Client Http.Downloaded Pages", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL); app_mgr_state_ptr-&gt;HTTP_NUM_OBJECTS = op_stat_reg("Client Http.Downloaded Objects", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL); app_mgr_state_ptr-&gt;HTTP_ABORT_ACCOUNT = op_stat_reg("Client Http.User Cancelled Connections", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL); /* The page response stat has been added recently to measure the time elapsed */ /* in completing the page request and getting its response. Earlier only the */ /* object response time was measured. */ app_mgr_state_ptr-&gt;HTTP_PAGE_RESPONSE_TIME = op_stat_reg("Client Http.Page Response Time(sec)", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL); app_mgr_state_ptr-&gt;GLOBAL_HTTP_PAGE_RESP_TIME = op_stat_reg("Http.Page Response Time (sec)", OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL); </pre>		
功能概述	返回在进程内部指向结点或模块统计量的句柄。可通过 <i>op_stat_write()</i> 和 <i>op_stat_write_t()</i> 来向其中写入值。		
详解	<p>这个函数和<i>op_stat_obj_reg()</i>类似，区别是它在进程内部使用，而前者在管道（pipeline）中使用。</p> <p>仿真核心管理所有的全局和本地统计量。所属组、名称、索引、类型以及输出向量都被保存。在使用<i>op_stat_reg()</i>时，用户可以确定前四项。稍后使用组和名称来使用统计量的函数将会据此找到已经注册的统计量。这允许多个进程使用同一统计量，只要它们有能够同步名称的机制即可。一个模块内部的进程可以共享局部统计量，所有的模块可以共享全局统计量。</p> <p>多维统计量提供了统计值。统计量句柄通过索引来指向具体的值。一维统计量使用OPC_STAT_INDEX_NONE。</p> <p>一个使用全局统计量的典型应用是计算端到端包传输时延。</p> <p>对统计量的注册包含对已注册统计量的搜索，以免出现重名。为了提高效率，一个统计量只应该注册一次，并且每个统计量用户都应该将其缓</p>		



<i>op_stat_write ()</i>			
语法：	<b>op_stat_write</b> (stat_handle, value)		
	参数	类型	描述
	<i>stat_handle</i>	<i>Stathandle</i>	即将写入统计量的统计量句柄。
	<i>value</i>	<i>Double</i>	统计值。
返回值	类型		描述
	无		
例程	<pre> /* obtain the arriving packet */ pkptr = op_pk_get (INPUT_STRM); /* determine the bulk size of the packet */ bulk_size = op_pk_bulk_size_get (pkptr); /* If the statistic handle for received packet size has not yet */ /* been initialized, do so now. */ if (pk_size_init == OPC_FALSE) {     pk_size_shandle = op_stat_reg ("Packet Size (bits)",     OPC_STAT_INDEX_NONE,     OPC_STAT_GLOBAL);     pk_size_init = OPC_TRUE; } /* write out the bulk size of the packet as a global statistic */ op_stat_write (pk_size_shandle, (double) bulk_size); /* destroy the now useless packet */ op_pk_destroy (pkptr); </pre>		
功能概述	将给定值和当前仿真时间写入到统计量中。		
详解	<p>这个函数可向输出矢量中写入一对新的数据（值，时间），统计量只能通过统计量句柄来访问。如果句柄错误，将导致可恢复性的错误。</p> <p>如果使用相同的统计量句柄，不同的进程可以向同一个统计量中写入数据。op_stat_reg()可以使用组和名称来确定统计量。</p> <p>多个进程使用这个函数时，是按顺序写入，而不是相互覆盖。统计量只能通过探针编辑器或者直接输出为输出向量矢量来分析。输出文件名字由仿真的探针属性决定。为了记录所有的全局统计量，需要设置record_gstats为真。</p> <p>这个核心函数使用强制中断。</p>		
开发意图	<p>将一个新数据（包括值和时间）写入统计量中。</p> <p>如果统计量句柄指向一个与已连接统计线相关的统计量，则当触发条件匹配时，会在另一端模块触发一个中断。</p>		
返回错误值	<ul style="list-style-type: none"> <li>分段错误（由无效统计量句柄导致）</li> <li>尝试使用未注册句柄。</li> </ul>		
相关函数	<ul style="list-style-type: none"> <li><b>op_stat_obj_reg()</b> 获得对任何链路，路径，模块或者子模块的统计量访问权。</li> <li><b>op_stat_reg()</b> 注册一个统计量，得到统计量句柄。 联系方式：<a href="mailto:lile_beauty@163.com">lile_beauty@163.com</a></li> <li><b>op_stat_writet()</b> 写入值到of命令输出统计量，包括当前的仿真时间。</li> <li><b>op_stat_scalar_write()</b> 写入值到一个统计标量中，最后将和其他仿真得到的结果合并。</li> <li><b>op_stat_local_read()</b> 读取输入统计的当前值。</li> </ul>		

## 分布函数

<b>Op_dist_load()</b>			
语法	<b>op_dist_load</b> (dist_name, dist_arg0, dist_arg1)		
	参数	类型	描述
	dist_name	const char*	将被载入的分布名称，（这个值一般是常数，例如：position）
	dist_arg0	double	分布的附加参数0
	dist_arg1	double	分布的附加参数1
返回值	类型		描述
	Distribution*		类型：指向载入的分布的指针。如果发生了可恢复的错误，OPC_NIL常数将被返回。函数的返回值一般储存在Distribution*类型的状态变量中，再传递给相关的op_dist_outcome()函数。
例程	<pre>/* Load the distribution function for the packet interarrival times. */ next_dist = op_dist_load ("exponential", ia_time, 0.0); if (next_dist == OPC_NIL)     jsd_gen_error ("Unable to load packet arrival distribution."); /* Load the distribution function for the job types. */ job_type_dist = op_dist_load ("uniform_int", 1, job_type_range); if (job_type_dist == OPC_NIL)     jsd_gen_error ("Unable to load job type distribution.");</pre>		
功能概述	得到一随机分布以便产生随机值。		
详解	<p>分布由名称唯一确定，分布可以通过系统预设，或者用户自行使用PDF编辑器及外部模块访问设定。预设的分布在此核心函数调用时可以看作是一个使用本函数参数的一个算法。用户自行定义两种分布一般是列表形式数据文件（.pd.s）的，当调用此函数时载入内存。要注意，预设的分布也在models/std/base文件夹内存在对应的数据文件(.pd.s)，但是这些文件都是零字节的，它们的存在只是为了在菜单中能够显示对应的分布，没有任何真实数据在其中。当使用用户自定义分布时，本核心函数两个参数都无效。</p> <p>这个核心函数对多线程来说是安全的。</p>		
开发意图	提供一种机制，以便于按照一定的分布来产生随机值。本函数最普通的用途就是确定包产生间的随机间隔。此外，任何需要随机确定的仿真参数都可以通过它得到。最终的实际值是通过调用op_dist_outcome()得到。		
返回	<ul style="list-style-type: none"> <li>分段错误(由无效分布名称指针，引起)</li> </ul>		



错误值	<ul style="list-style-type: none"> <li>内存分配错误</li> <li>分布名字(&lt;dist_name&gt;) 对该对象不可识别</li> </ul>
相关函数	<ul style="list-style-type: none"> <li><b>op_dist_outcome()</b> <b>op_dist_outcome_ext()</b> 得到分布对应的随机值</li> <li><b>op_dist_uniform()</b> 得到均匀分布的随机值</li> <li><b>op_dist_unload()</b> 分布所占有内存</li> </ul>

<b>Op_dist_outcome()</b>			
语法：	<b>op_dist_outcome</b> (dist_ptr)		
	参数	类型	描述
	dist_ptr	Distribution*	将被载入的分布名称（这个值一般是常数，例如：position）
返回值	类型		描述
	double		根据特定分布得到的随机值。如果发生可恢复的错误，将会返回OPC_DBL_INVALID。
例程	<pre> /* Determine the time at which to schedule the interrupt. */ next_pk_arrvl_time = op_sim_time () + op_dist_outcome (int_arrival_distptr); /* Check to verify that this time is within the end time */ /* specification for this module. */ if ((next_pk_arrvl_time &lt; gen_end_time)    (gen_end_time == FRMSC_APPL_END_OF_SIM)) { /* Schedule the self-interrupt. */ op_intrpt_schedule_self (next_pk_arrvl_time, FRMSC_FR_APPL_TRAF_GEN); } </pre>		
功能概述	基于特定分布，产生一个浮点随机值。连续多次调用本函数，将会基于该分布产生一系列随机值。		
详解	<p>特定的分布可以是使用op_dist_load()预先载入的系统预设算法，或者是用户自定列表。这个核心函数将会进行查表或者使用该算法进行运算（如：指数分布）。</p> <p>本核心函数使用的随机数流是通过BSD random()函数来得到的，这个数流也被所有其他分布类核心函数和包产生器使用。随机数流基于一个初始种子数，而该种子数可以通过仿真环境属性"seed"改变。如果仿真输入相同（包括seed），结果是可重复的。</p> <p>这个核心函数对多线程来说是安全的。</p>		
开发意图	实现了按照特定的分布得到随机值。本函数最普通的用途就是确定包产生间的随机间隔。此外，任何需要随机确定的仿真参数都可以通过它得到。		
返回	<ul style="list-style-type: none"> <li>分段错误（由无效分布名称指针，引起）</li> </ul>		

错误值	<ul style="list-style-type: none"> <li>分布指针为空。</li> </ul>
-----	---

## 事件和仿真函数

<b>op_ev_cancel()</b>			
语法：	<b>op_ev_cancel(evhandle)</b>		
	参数	类型	描述
	evhandle	Evhandle	识别需取消事件的事件句柄（此句柄可以通过诸如op_ev_seek_time() op_ev_next() op_intrpt_schedule_self()等函数得到）。
返回值	类型		描述
	Compcode		完成代码用来确认操作是否正确完成，返回值可能为：OPC_COMPCODE_SUCCESS 或者OPC_COMPCODE_FAILURE。
例程	<pre> /* Obtain the currently executing event (it must be at this module, */ /* since this module woke up.) and the one after it. */ this_event = op_ev_current (); next_event = op_ev_next_local (this_event); /* Loop through all of the events scheduled for this module */ /* and cancel any that are retransmission timers. */ while (op_ev_valid (next_event)) { if ((op_ev_type (next_event) == OPC_INTRPT_SELF) &amp;&amp; (op_ev_code (next_event) == RETRANS_TIMER)) { retrans_timer = next_event; next_event = op_ev_next_local (retrans_timer); op_ev_cancel (retrans_timer); } else /* Obtain the next event; if there are no more events, op_ev_next_local () */ /* will return an invalid event handle and the loop will terminate. */ next_event = op_ev_next_local (next_event); } </pre>		
功能概述	取消已经预先安排好的事件		
详解	<p>取消一个已经处于事件列表内但尚未执行的事件。对已经交付或者正在执行的事件使用此函数会发生错误，只有还没有执行的事件才能取消。使用这个函数取消事件不会带来效率上的降低，因为仿真核心根据它</p>		

	们的事件句柄来得到信息，很快可以在事件表中定位，不需要对整个表进行搜索。所以在事件没有执行前将其取消比让其发送到目的地而忽略要节省资源得多。 这个函数使用强制串行。
开发意图	提供了一种取消无意义事件的机制。一般都是用来取消用 <code>op_intrpt_schedule_self()</code> 设定的时间溢出事件，也就是说，如果一个包在定时器溢出前已经到达目的地，那么定时器溢出这一事件就变得没有必要，通常通过这个函数取消，并且重新设置定时器。
返回错误值	<ul style="list-style-type: none"> <li>• 分段错误Segmentation violation (由畸形事件句柄引起)</li> <li>• 事件句柄指向了一个无效或者已经还没有安排的事件。</li> <li>• 当前事件不能被取消。</li> <li>• 正在进行中的事件不能取消。</li> <li>• 不能取消未定位事件。</li> </ul>
相关函数	<ul style="list-style-type: none"> <li>• <b><code>op_ev_pending()</code></b> 决定一个事件是否可以取消</li> <li>• <b><code>op_intrpt_schedule_self()</code></b> 安排一个本进程自中断</li> <li>• <b><code>op_intrpt_clear_self()</code></b> 取消本进程中所有待决中断</li> </ul>

<b><code>op_sim_time ()</code></b>			
语法：	<b><code>op_sim_time ()</code></b>		
	参数	类型	描述
	无		
返回值	类型		描述
	<i>double</i>		当前仿真时间。双精度浮点数代表了当前的仿真时间，单位为秒，从begin simulation interrupt起计数，在该时刻返回值为0。
例程	<pre>/* get a handle on packet at head of subqueue 0 */ /* (this does not remove the packet) */ pkptr = op_subq_pk_access (0, OPC_QPOS_HEAD); /* determine the packets length (in bits) */ pk_len = op_pk_total_size_get (pkptr); /* determine the time required to complete */ /* service of the packet */ pk_svc_time = pk_len / service_rate; /* schedule an interrupt for this process */ /* at the time where service ends. */ op_intrpt_schedule_self (op_sim_time () + pk_svc_time, 0); /* the server is now busy. */ server_busy = 1;</pre>		
功能概述	得到当前仿真时间		
详解	当前仿真时间是一个双精度浮点数，即从仿真开始到当前事件所经过的时间。在一个进程仿真过程中，仿真时间并不是时刻变化，而是当一个		

	旧的事件完成（时间T）而下一个新的事件（时间T + dT）发生才变化。因为时间按照这种不规则模型可以间断性的跳跃，我们把这种模型叫做事件驱动方式。 这个核心函数使用强制串行。
开发意图	提供得到当前仿真时间的机制：以供：1、计算绝对时间，以便未来安排事件，2、确定从过去某事件到当前的时延，3、在输出设备或者输出文件中打印基于时间的诊断结果。
返回错误值	无
相关函数	<ul style="list-style-type: none"> <li>• <code>op_intrpt_schedule_self()</code>, <code>op_intrpt_schedule_remote()</code>, 或 <code>op_intrpt_schedule_call()</code> 来安排未来的中断</li> <li>• <code>op_pk_creation_time_get()</code> 或 <code>op_pk_stamp_time_get()</code> 来得到每个包的时间戳，以便和当前时间比较</li> <li>• <code>op_q_wait_time()</code> 或 <code>op_q_insert_time()</code> to 从队列模块中存储的包来得到队列待时延</li> </ul>

## OPNET Modeler 中的无线建模

对于无线系统而言，和有线网络最大的区别是无线信道的广播和时变特性，以及结点的移动性。无线信道需要被合理的模拟，无线信道的频率，功率，视距，以及干扰等等都需要在仿真中体现。如果无线信道刻画的不准确，将直接影响到高层的性能以及仿真的精确性。Modeler 的无线模块提供了模拟无线网络的功能，可以很好的模拟结点的移动性，并通过收发信机管道来刻画无线信道。

Modeler 的无线模块包括：

- 移动结点和子网
- 卫星结点和子网
- 天线模型编辑器
- 调制曲线编辑器
- 收发信机管道阶段
- 并行仿真功能

## OPNET Modeler 对移动性的支持

无线模块增加了两种结点和子网：其一为移动结点和移动子网（移动结点和移动子网统称为移动站），其二为卫星结点和卫星子网（卫星结点和卫星子网统称为卫星站）。移动结点可以基于预先定义的移动轨迹或者进程模型中的程序控制来进行移动，卫星结点则是根据设置的卫星轨迹来进行运动。

对于使用无线方式进行通信的系统来说，距离，视距传输以及其他与位置相关的信息对于系统的性能是十分重要的。因此对于无线通信系统的仿真来说，建立恰当的结点或子网的移动性模型至关重要。

### 运动轨迹

运动轨迹就是移动站在仿真过程中运动的路线。运动轨迹可以基于段，也可以基于向量。最常用的是基于段的运动轨迹，在这里只叙述基于段的运动轨迹。

#### 1. 基于段的运动轨迹

基于段的运动轨迹通过预先定义好的一系列点来进行定义。基于段的运动轨迹包含定长或变长的时间间隔（也就是两个位置点之间的运动时间），以及一系列的三维坐标（x,y 以及高度）。运动轨迹文件存在 ASCII 码的文本文件中，文件后缀是.trj。通过编辑结点或者子网的 Trajectory 属性来为其分配运动轨迹。

仿真中，移动结点根据定义的运动轨迹，从一个点移动到下一个点。OPNET 通过在两点间进行线性的插值，如果仿真时间超过基于段的运动轨迹所设定的最大时间，则该对象仍然停留在运动轨迹的末端。

基于段的运动轨迹分为定长时间间隔和不定长时间间隔两种。对于定长时间间隔来说，两点之间的运动时间是相等的，无论这两点的距离是多少，而对于不定长时间间隔来说，每个点都有其独自的高度，等待时间，以及运动时间（前一点到当前点的运动时间），等待时间就是结点在向下一点运动前，在该点停留的时间。

两种类型的运动轨迹都是基于 ASCII 码格式的文件(.trj)来确定运动轨迹，可以通过文本编辑器来编辑.trj 文件，也可以在项目编辑器里采用图形化的方式来定义运动轨迹。

下面讲述如何在项目编辑器中定义结点的运动轨迹。

（1）打开项目编辑器，选择Topology->Define trajectory...，出现对话框，可以选择定长时间间隔的，也可以选择不定长时间间隔的。

（2）在Altitude或者Initial Altitude字段中，输入高度或者初始高度值。

（3）（只适用于变长时间间隔运动轨迹）在Initial Wait Time字段中填入初始等待时间。

（4）（只适用于定长时间间隔运动轨迹）在Time Step域内设置每段的运动时间。

（5）设置Coordinates are relative to object's position，如果该框被选中，则x, y坐标是相对于初始位置的偏移。如果该框没有被选中，则该坐标被视为绝对值。

（6）单击Define Path按钮。

（7）定义运动轨迹上的点，如下所示：

定长时间间隔的轨迹：在一些点上，单击鼠标左键，完成定义整个运动轨迹后，单击鼠标的右键，命名定义的轨迹文件的名字。

不定长时间间隔的轨迹：单击鼠标左键在第一个要定义的位置，出现运动轨迹段信息对话框，在该对话框中输入相应的信息。在 Traverse segment in/at 字段中输入上一点到该点的运动时间或者速度，然后输入该点的高度以及在该点的等待时间。

（8）最后，如果还想编辑定义的运动轨迹，单击鼠标右键，编辑运动轨迹，出现如下对话框。

（9）选中结点，单击右键，选择Edit Attribute，在Trajectory一项，选择定义好的运动轨迹。

## 卫星轨道

卫星结点依靠卫星轨道来确定其在仿真中的位置。卫星轨道通过卫星轨道文件来定义，卫星轨道文件是由前卫星轨道仿真软件 STK 生成的。Modeler 的用户可以通过导入卫星轨道文件来将 STK 生成的轨道文件导入。在仿真运行的时候，当需要卫星的位置信息时，仿真核心将基于卫星轨道文件中的点来计算卫星的当前位置，卫星的位置是通过在轨道文件中的抽样值进行线性插值来计算的。

导入卫星轨道后，就可以使用 OPNET 的 op\_vuorb 来进行查看卫星的运行轨迹。

### 1. 卫星工具包 (Satellite Tool Kit)

无线模块可以使您输入 STK 创建的卫星轨道，STK 标准是免费的，可以从以下 Analytical Graphics 的主页获得：[www.stk.com](http://www.stk.com)

### 2. 导入 STK 轨道

在 STK 中创建了轨道文件之后，可以有两种方法来导入 STK 的文件。两种方法

联系方式：[life\\_beauty@163.com](mailto:life_beauty@163.com)

都是将 STK 创建的.v 或.sa 文件格式转化成 OPNET 的.orb 文件。

注意：只有 STK3.0 以上的版本生成的.v 或者.sa 文件才能转化成.orb 文件。

第 1 种方法：

- (1) 选择 topology 菜单下的 Import STK orbit，弹出对话框
- (2) 确定要导入的文件。
- (3) 单击 OK。

第 2 种方法：

- (1) 右键单击卫星结点，选择 Edit Attributes，弹出对话框，选中 Advanced。
- (2) 单击 Import STK Orbit 按钮，确定导入的 STK 轨道文件名。
- (3) 单击 OK。

第 2 种方法和第一种方法不同的是，将导入的轨道加到选中的卫星结点。

## 通过进程模型来控制位置

如果没有为移动结点定义轨迹，则可以通过进程模型中的代码来直接控制移动结点的移动。移动结点具有 x position, y position 以及 altitude 这些属性，在进程模型中可以直接去修改这些属性。这些属性的改变会直接影响移动结点的位置。使用进程模型来控制位置分为两种，一种为分布式的控制，一种为集中式的控制。所谓分布式的位置控制就是每个移动结点自己控制自己的位置属性，如下面一段代码所示：

所谓集中式的位置控制，就是由一个进程文件，集中的更新各个移动结点的位置，如下面代码所示：

## 无线收发信机管道建模

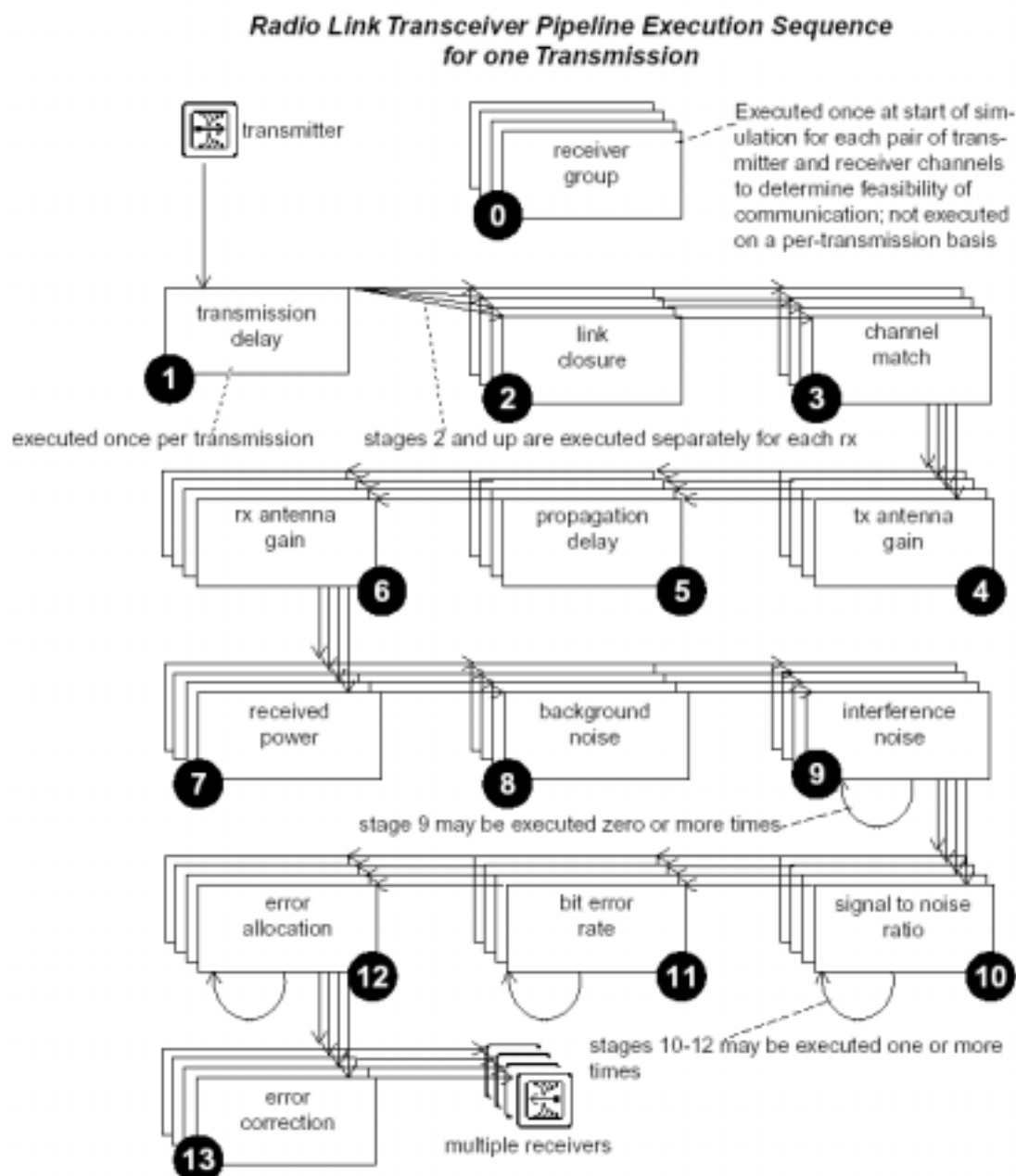
### 无线收发信机管道

由于无线链路是一种广播媒介，每一次传输都可能影响到整个网络模型中的多个接收机。另外，对于某个特定包的发送，每个接收机的无线链路可能呈现不同的行为和定时特征。因此每个接收机都必须执行不同的管道阶段。

无线收发管道包括 14 个管道阶段，大多数必须在每个接收机中执行。然而，第一个阶段(接收机组)在每对收发机中只执行一次，以建立发送与接收机信道之间的静态绑定。第二个阶段(发送时延)用来计算一个对所有目的结点都适用的数值，因此每次发送可以只执行一次。最后，每个管道序列可以也可完全不需要执行，这依赖于第三个阶段(闭合阶段)的结果，因为该阶段负责判断收发机之间是否可以通信。类似的，第四个阶段(信道匹配)考虑收发信机之间的信道匹配的影响，进行接收可达性的判断，这样避免了管道序列必须执行到最后一个管道。

必须注意到无线链路最后的几个阶段对于每个接收机可能执行多次，这是由于可能有多个同时包发送。为了检测在同一接收机信道的多个包的接收，仿真内核为每个无线接收机信道保留了两个“当前”数据包的表单。第一个表只包含信道匹配阶段检测是“有效”的数据包；第二个表包含了“无效”的数据包。通常，这要求发射与接收机管道之间存在匹配的特性，也可以是接收机信道能够与数据包信号保持同步。在不同的表单中保留“无效”和“有效”包，主要是因为这样使得仿真内核可以只对“有效”的包执行特定的管道阶段和计算特定信道的统计数据。例如，对于接收机不接收的包来说，就不需要计算信噪比、比特错误率或者错误分布参数。通常，接收功率阶段之后的所有的阶段都只适用于“有效”的数据包。

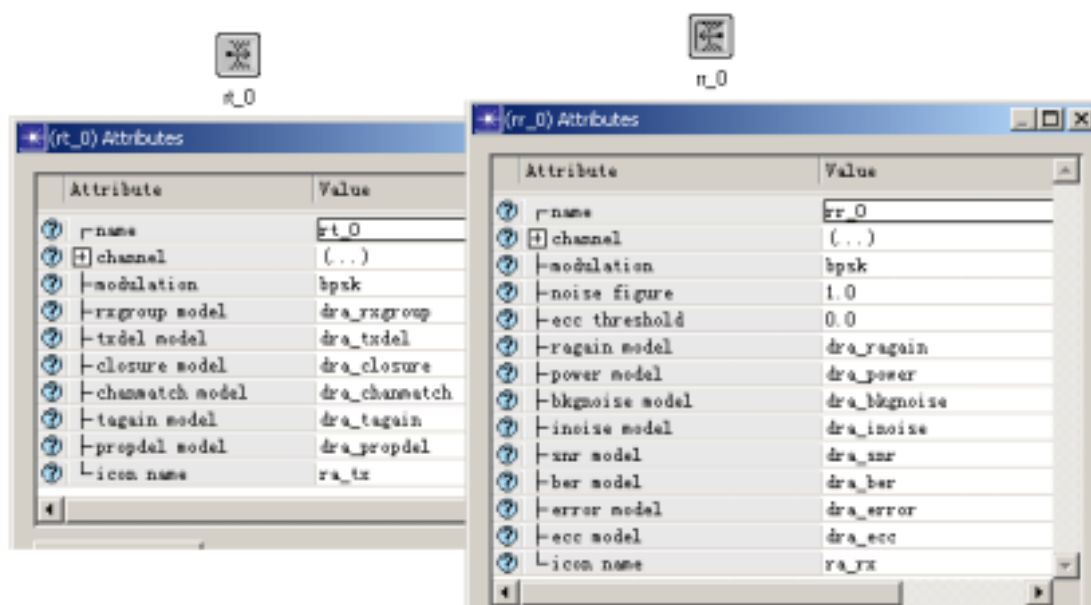




### 无线链路中的管道阶段

管道阶段 9 到阶段 12 用来估计链路的状况，以及相应信号状态的变化。通常至少调用阶段 10 到阶段 12 其中之一来估计有效数据包持续时间内的性能。然而，当干扰包到达时，需要调用阶段 9 到 12 之间的每个阶段，以计算新的信号状况。

由于无线链路不作为物理对象而存在，所以用来进行特定的无线传输的管道阶段必须与形成链路的发射机与接收机联系起来。某些阶段与发射机联系，其余的与接收机联系，如下所示：



无线发送机和接收机的属性

### 1. 保留的发送数据属性

仿真内核预留了很多 TDA，管道阶段可以访问最少的一组标准值，也可在其自身与管道阶段以及各阶段之间进行通信。为此而保留在包中的 TDA 如下表所示。为了简洁起见，表示 TDA 索引值的符号常数表示成短的形式，没有通常的前缀。当在实际中应用时，每个符号常数以字符序列“ OPC\_TDA\_RA ”追加在名称前。每种 TDA 的数据类型在符号常数名称旁表示：“ D ”代表一个双精度浮点值，“ I ”代表整数。

## 阶段 0：接收机组

接收机组阶段实际上并不是处理发送的动态管道的一部分。但是，由于它能计算出结果，影响无线传输的行为特征，它被认为是部分管道，由无线发射机的“ rxgroup model ”属性表示。

当无线发送开始时，仿真内核通过在发送信道与一组接收机信道之间实现多个无线链路，来模拟无线链路的广播特性。每个发送机信道都保留着其自身的信道“接收机组”，这是可能的接收传输的候选信道。接收机组阶段的目的是为每个发射机信道创建一个初始的接收机组。仿真内核检查每种可能的发送接收机信道对，并为每个发射机信道构建一个接收机组。

然而这样引发了一个问题：由于收发机特性在仿真中动态变化，通常在开始时很难判定哪个接收信道与发送信道匹配对应。因此，接收机组阶段包括了一个接收信道，除非它能够事先判定发送信道永远不与该接收信道对应。仿真过程中，随后的管道阶段可以动态的判定接收信道能否接收传输的包。仿真中使接收信道不能接收的可能原因包括：

(1) **分离的频带**。频带由其基频和带宽描述。如果发送信道与接收信道的频带不交叠，那么发送信道的发送不会影响接收信道 - 或者作为有效的信号或者当成噪声。

(2) **物理分离**。发送信道与接收信道之间可能相隔很远，而建立一条无线链路将没有足够的信号强度。无线链路的建立也取决于诸如这些因素如发射机接收机天线高度、发送信号功率以及频率等。

(3) **天线失效**。管道阶段分别在阶段 4 和阶段 6 对发射机与接收机天线的增益模型进行建模。当应用方向性天线时，这些阶段计算出的结果可能极大的衰减信号功率 以至

联系方式：[life\\_beauty@163.com](mailto:life_beauty@163.com)



于仿真可以合理的忽略发送对接收信道的影响。

在某些网络模型中，仿真内核可以判定发送接收信道对之间是完全不能进行通信的。这种情况下，将接收信道从发射机信道的接收机组中移去可以加速仿真，因为这样减少运行管道阶段，而不产生影响。虽然接收机组在仿真开始之前被调用，但是结果不应取决于可能在仿真中变化的因素。这种潜在的“iffy”标准的例子如两个移动阶段之间最初的距离以及分配给发送和接收信道初始的频率(假设模块能动态改变这些频率)。缺省情况下，这些情况下 OPNET 包含接收信道，所有的接收机组在整个仿真中保持不变；随后的管道阶段于是可以利用动态标准来估计发送接收机信道之间的连接性。

仿真内核要求结束机组阶段程序分别接收发送和接收信道的“对象 ID”。程序应该返回一个整数值(OPC\_TRUE 或 OPC\_FALSE)给内核；该数值表明了接收机信道是否为一个合适的目的端，而且应当包含在接收机组中。

可以利用核心函数(KP)中的无线函数包来改变缺省的接收机组性质，以及针对仿真事件动态的改变和重新计算接收机组。例如，如果在仿真进程中接收机结点被屏蔽掉，可以从接收机组中移去接收信道。同样可以使用函数 `op_radio_txch_rxgroup_compute()`，在仿真中的任何时间计算或者重新计算给定信道接收机组(本质上，这是重新调用接收机组管道机阶段)。甚至可以完全“跳过”接收机组阶段，接着按需要构造并更新接收机组。(跳过这个阶段涉及到设置“rxgroup model”为“dra\_no\_rxgroup”而不是缺省值“dra\_rxgroup”，这样就为所有的发送信道创建了空接收机组)。动态的更新接收机组可以进行更快速的仿真，尤其是对高无线通信量的网络模块。“跳过”管道阶段以及按需要创建接收机组也可加速有大量收发信道的网络的仿真。

## 阶段 1：发送时延

发送时延阶段数值上是无线管道的第二个阶段，但当发送新的数据时却是首先执行的。它由无线发射机的“txdel model”属性描述，开始发送数据包时立即执行。这是唯一的阶段，只执行一次而对所有的管道都适用。

调用该管道来计算整个包完成发送所需要的时间。这个时间是第一个 bit 开始发送的时间与最后一个 bit 完成发送之间的仿真时间。仿真内核利用该阶段返回的结果在发送包的发送信道中调度一个结束发送事件。当该事件发生时，发射机可以在信道内部的队列中开始发送下一个包；否则发送信道进入空闲状态。另外，发送时延与传播时延结合起来计算包被链路目的模块接收的时间(例如，最后一个比特到达的时间就是它发送完成的时间加上在链路上的传播时延)。

仿真内核要求发送时延阶段函数将包地址作为其唯一的参数。计算出的发送时延由内核符号常量 OPC\_TDA\_RA\_TX\_DELAY 表示。分配值应当是非负的双精度浮点值。不允许取别的值。

## 阶段 2：闭合阶段

闭合阶段是管道的第三个阶段，由无线发射机的“closure model”属性描述。它在发送信道目的信道设置中所指定的接收信道中只执行一次，在发送时延阶段返回后立即执行，期间没有仿真时间过去。该阶段的目的是判断一个特定的接收信道是否受发送的影响。发送到达接收信道称之为发送信道与接收信道之间的闭合，正如该阶段的名字。

注意到闭合阶段的目的是不是判断是否某个传输对特定的信道是有效的或恰当的，而是发射信号能否得到候选的接收信道，通常，该阶段所进行的计算大多数基于物理上的考虑，例如障碍物、地球的表面等。

仿真内核要求该阶段得到一个整数值。内核利用其判断是否执行下面的无线管道阶段。如果该整数值等于符号常量“OPC\_TRUE”，那么就建立闭合，信号在发送与接收机之

间的传送就可能；否则符号常量 OPC\_FALSE 用以表明不可以。

仿真内核要求该阶段将包地址作为其唯一的参数。闭合标志由内核符号常量 OPC\_TDA\_RA\_PROP\_CLOSURE 表示，不允许取别的数值。

### 阶段 3：信道匹配

信道匹配阶段是管道的第四个阶段，由无线发射机的“chanmatch model”属性指定。它在每个满足链路闭合的接收机信道中都执行一次。在链路闭合阶段返回后立即触发该阶段，期间没有仿真时间过去。该阶段的目的是根据接收信道对发送进行分类。三种可能的种类之一必须指定给包，如下所示：

(1) 有效性。该类的包被认为是与接收信道匹配，可以被接收并转发到接收结点中的别的模块，只要其不受过多错误的影响。将数据包分类为有效的通常取决于是否发送接收机之间关键参数有至少一项符合。

(2) 噪声。这种分类用来表示数据内容不能接收的包，但对接收信道产生干扰而影响其性能。发射与接收信道配置的不匹配，就将包分类为有噪声的。

(3) 忽略。如果一次发送决定对接收信道的状态或性能都不产生影响，那么它应被分为此类。仿真内核将停止执行发送信道与接收信道之间的管道阶段(未来的信道之间的传输将不会被阻止)。

仿真内核要求信道匹配阶段函数将包地址作为其唯一的参数。分类由内核符号常量 OPC\_TDA\_RA\_MATCH\_STATUS 表示。符号常量 OPC\_TDA\_RA\_MATCH\_VALID, OPC\_TDA\_RA\_MATCH\_NOISE, 或 OPC\_TDA\_RA\_MATCH\_IGNORE 分别用来表示有效、噪声和忽略种类。不允许别的值。

### 阶段 4：发射天线增益

发射天线增益是无线收发管道的第五个管道，由无线发射机的“tagain model”属性指定。其对每个目的信道分别执行，除了那些链路闭合阶段失败以及那些信道匹配阶段将传输分类为“忽略”的信道。发射天线增益阶段在信道匹配阶段返回后立即调用，期间没有仿真时间过去。发射天线增益阶段的目的是计算发射机关联的天线的增益，基于发射机到接收机向量的方向。仿真内核自己不用该结果，当其通常被分解用以阶段 7 计算接收功率。

天线增益刻画了发射信号能量放大或衰减的现象。发射功率的“整形”是基于天线结构的物理特性以及可能的在某个天线上的相位。

在任何方向上都没有增益的天线称为“全向”天线，因为其对于所有可能的信号路径都有最优的对称性。特定方向上的天线增益测量用来与全向天线做比较。它定义为给定距离的天线的信号功率与同样距离的全向天线之比。增益是无单位的量，通常表示为分贝的形式(dB)。

通过测量或者计算一系列方向上的增益，可以构建三维的天线增益模型，它描述了不同发送方向上的天线的效果。OPNET 不提供计算天线增益模型的工具，但是天线编辑器可以根据经验形式得到天线模型的数据(例如，模型必须已经知道，并且可以表示为数字形式)。另外，EMA 程序接口可以用 C 语言程序描述天线模型，允许用分析的方法形成模型，或者是数据库的检索(例如，别的面向天线的程序产生模型数据，这些数据能被 OPNET 所利用)。不考虑用来捕获天线模型数据的方法，天线编辑器还可以展示 3D 的天线模型，如下面的例子所示：

通常，发射天线增益阶段首先计算方向向量分离发射机与接收机，接着利用天线模型的信息来决定发射的天线增益。该管道阶段也指明了天线的方向。方向由天线的一组属性控制，包括“pointing ref. phi”和“pointing ref. theta”，这两个属性指明了天线模型的一个特定部分在空间中指向一个选定的点。管道阶段可以通过 TDA：

联系方式：[life\\_beauty@163.com](mailto:life_beauty@163.com)

OPC\_TDA\_RA\_TX\_BORESIGHT\_PHI 和 OPC\_TDA\_RA\_TX\_BORESIGHT\_THETA 来取得发射天线的这些属性。另外,天线属性“ target latitude ”;“ target longitude ”以及“ target altitude ”属性定义了空间中天线的参考点。管道阶段如果需要的话可以直接从天线对象获得这些属性值。然而,仿真内核提供了来源于这些属性的信息,在大多数情况下对天线增益的计算是足够的; 这些 信息 包含 在 TDA “ OPC\_TDA\_RA\_TX\_PHI\_POINT ” 和 “ OPC\_TDA\_RA\_TX\_THETA\_POINT ” 中,它们表示分离发射机和目标位置的向量的方向角  $\varphi$  和  $\theta$ 。这些角度与用来定义天线模型的  $\varphi$  和  $\theta$  没有混淆。相反的,这些目标方向角是相对于笛卡儿坐标系而定义的。

仿真内核保留了一个 TDA,由符号常量 OPC\_TDA\_RA\_TX\_GAIN 表示,来存储发射天线增益阶段的结果以便后面的管道阶段使用。仿真内核要求阶段的函数将包地址作为其唯一的参数。该阶段的语法要求如下面的函数模板所示:

### 阶段 5 : 传播时延

传播时延阶段是无线收发机管道的第六个阶段,由无线发射机的“ propdel model ”属性表示。每个接收信道成功通过链路闭合和信道匹配阶段后调用该阶段。该阶段的调用发生在发射天线增益阶段返回后,期间没有仿真时间过去。该阶段的目的是计算包信号从无线发射机到接收机所需要的时间。内核利用这个时间在接收信道中设置一个开始接收事件。另外,传播时延值用来与发送时延结合起来计算包接收完成所需的时间(例如,最后一比特到达的时间就是包开始发送的时间加上传播时延与发送时延之和)。

仿真内核要求传播时延阶段函数将包地址作为其唯一的参数。计算的传播时延由内核符号常量 OPC\_TDA\_RA\_START\_PROPDEL 和 OPC\_TDA\_RA\_END\_PROPDEL 表示。指定的值应当是非负的双精度浮点值。不允许取别的值。该阶段的语法要求如下面的函数模板所示:

### 阶段 6 : 接收天线增益

接收天线增益阶段是无线收发机管道的第七个阶段。它是与无线接收机而不是发射机关联的最早阶段,由接收机的“ ragain model ”属性描述。每个符合条件的目的信道分别执行该阶段,发生在包到达接收机时(传播时延过去之后)。

接收天线增益阶段的目的是计算接收机所关联的天线增益,基于从接收机到发射机向量的方向。仿真内核自己不应用该计算结果,但通常将其分解用于阶段 7 计算接收功率。

接收天线增益的概念与发射天线增益的概念完全相同,除了接收天线增益是与接收机所关联的天线的物理配置及实现有关的。

仿真内核保留了一个 TDA,由符号常量 OPC\_TDA\_RA\_RX\_GAIN 表示,来存储该阶段的数值结果以为后来的管道阶段使用。仿真内核要求该阶段的函数将包地址作为其唯一的参数。该阶段的语法要求如下面的函数模板所示:

### 阶段 7 : 接收功率

接收功率阶段是无线收发机的第八个阶段,由无线接收机的“ power model ”属性描述。对每个合格的目的信道分别调用该阶段;调用发生在接收天线增益返回后,期间没有仿真时间过去。该阶段的目的是计算到达数据包信号的接收功率(瓦)。

对分类为有效的包而言,接收功率值在判定接收机是否正确的获取包中的信息时是一个关键的因素。对于分类为噪声的包,通常必须估计接收功率以计算有效的和噪声的包的相对强度。

通常,计算接收功率是基于诸如这些因素如发射功率,发射机与接收机之间的距离,

发射频率，发射和接收天线增益等。仿真内核要求接收功率阶段函数将包地址作为其唯一参数。计算的接收功率由内核符号常量“OPC\_TDA\_RA\_RCVD\_POWER”指定。指定值应当是一个非负的双精度浮点值，不允许别的值。

## 阶段 8：背景噪声

背景噪声阶段是无线收发机阶段的第九个阶段，由无线接收机的“bkgnoise model”属性描述。它在接收功率阶段返回后立即执行，期间没有仿真时间过去。该阶段的目的是表示所有噪声源的影响性，除了别的同时到达的传输包—由干扰噪声阶段表示。预期的数值是其余的噪声源功率之和，这些功率是在接收机处和接收信道通带处测量。典型的背景噪声源包括了从临近电子元件或者未建模型的无线电发射的热噪声或射电噪声(例如，商业无线电台，业余无线电，电视，这取决于频率)。

仿真内核自身不利用该阶段的数值结果，但它却保留一个 TDA—由符号常量 OPC\_TDA\_RA\_BKGNOISE 表示—来存储数值结果并传递给随后的管道阶段。通常，在信噪比阶段，背景噪声值是后来加到别的噪声上来计算总的噪声。内核要求背景噪声阶段将包地址作为其唯一的参数。

## 阶段 9：干扰噪声

干扰噪声阶段是无线收发机的第九个阶段，由无线接收机的“inoise model”属性描述。对于数据包而言，有两种情况可能需要执行该阶段：(1)包是有效包并且到达目的信道同时另外一个包已经接收到了；(2)包是有效包并且已经被接收到了，当另外一个包(无论有效还是无效)到达时。很明显，大多数情况下对每个包出现第一种情形，第二种情形可能出现多次，这取决于模块中别的发射机的传输状况。注意到如果两个包都是有效的，这两个包可以共享一次调用干扰噪声阶段(调用的语法如下所示，提供两个包的地址给干扰噪声阶段以估计共同的影响)。

该阶段的目的是说明同时到达同一接收信道的发送之间的影响。仿真内核保留了一个 TDA—由符号常量 OPC\_TDA\_RA\_NOISE\_ACCUM 表示—来存储当前从所有的干扰传输来的噪声值。由于通常不需要对有噪数据包进行链路状况估计，所以只是对有效的包(由信道匹配阶段决定)才保留这样的累加器。这样，干扰噪声阶段利用干扰包的接收功率在每一有效包中增加该累加器的值。当一个包接收完成时，内核自动地在仍旧在信道中的包的噪声累加器中减去接收功率值，累加器只反映了当前地噪声值。

内核要求干扰噪声阶段函数将两个包地址作为参数。第一个包地址代表最早到达的包，第二个表示新到的并触发干扰噪声阶段的包。

除了 OPC\_TDA\_RA\_NOISE\_ACCUM 这个 TDA 外，内核还提供了一个 TDA 来记录每个包所经历的碰撞次数。这个 TDA 由符号常量 OPC\_TDA\_RA\_NUM\_COLLIS 表示，在开发面向应用的管道阶段时提供了便利，因为当判定在最后一个管道阶段中是否接收或丢弃包时需要用到该结果。虽然干扰阶段是更新这个 TDA 的一个合适的位置，但内核并不应用这个 TDA，因此也不需要保证其精确性。

## 阶段 10：信噪比

信噪比(SNR)阶段是无线收发机管道的第十一个阶段，由无线接收机的“snr model”属性描述。对于有效包，在三种情形下被执行：(1)包到达目的信道；(2)一个包已经接收到而另外一个(无论有效还是无效)到达时；(3)一个包已经接收到而另外一个(无论有效还是无效)完成接收。很明显，第一种情形对每个包只出现一次，第二和第三个可能发生任意多次，取决于模型中其余的发射机传输状况。这三种类型的调用描述了包的平均 SNR 被认为是常数的时间间隔(当然，当存在移动性时这只是一个近似，因为 SNR 将会联系变化)。



SNR 阶段的目的是计算到来的数据包 SNR 值，通常它是基于早期阶段获得的数值，包括了接收功率、背景噪声以及干扰噪声。包的 SNR 值是一个重要的性能量度，用来判定接收机是否正确的收到包的内容。内核利用该阶段计算的结果来更新接收信道的输出值，通常管道后面的阶段也用到该值。

仿真内核要求 SNR 阶段函数将包地址作为其唯一的参数。计算的 SNR 值由内核符号常量 OPC\_TDA\_RA\_SNR 来表示。指定的值应当是一个双精度浮点值，以分贝(dB)表示。不允许别的数值。

### 阶段 11：误比特率

误比特率(BER)阶段是无线收发信机管道的第 12 个阶段，由无线接收机的“ber model”属性描述。对有效包（由信道匹配阶段决定）在 3 种情形下可能调用该阶段：(1) 包在目的信道完成接收；(2) 一个包已经接收到而另外一个包（无论有效还是无效）到达时；(3) 一个包已经接收到而另外一个包（无论有效还是无效）完成接收时。这几种情形对应于包的 SNR 值为常数。

BER 阶段的目的是从过去的 SNR 值为常数的阶段中得到比特错误概率。这不是根据经验的比特错误率，而是基于 SNR 的预期的值。通常，该阶段的比特错误率也是用于发送信号的调制类型的函数。

仿真内核要求 BER 阶段函数将包地址作为其唯一的参数。计算出的 BER 由内核符号常量 OPC\_TDA\_RA\_BER 表示。指定的值应当是一个介于 0 和 1（包括 1）之间的双精度浮点值，不允许取别的数值。

### 阶段 12：错误分布

错误分布阶段是无线收发信机的第十三个阶段，由无线接收机的“error model”属性描述。通常在比特错误率阶段返回后立即执行。错误分布阶段的目的是估计数据包中的一段的比特错误数目，这一段的比特错误概率已经计算并且为常数。如果比特错误概率在包的接收过程中没有变化，那么段可以是整个包。比特错误数目的估计通常基于比特错误概率（从阶段 11 中获得）和段的长度。

仿真内核要求错误分布阶段函数将包地址作为其唯一的参数。内核只在包中保留一个比特错误累加器 TDA；因此，管道阶段将新发生的错误数目加到整数 TDA 的已存在的数值中，用符号常量 OPC\_TDA\_RA\_NUM\_ERRORS 表示。增加的数值应当介于 0 和受影响的段的长度之间。另外，内核要求错误分布阶段在数据包段上提供经验式的比特错误率；这个错误率可以通过将段中的比特错误数除以段的大小获得。经验的比特错误率应当置于一个双精度的 TDA 中，它由符号常量 OPC\_TDA\_RA\_ACTUAL\_BER 表示。内核利用这个数值来更新接收信道的比特错误率。

### 阶段 13：错误纠正

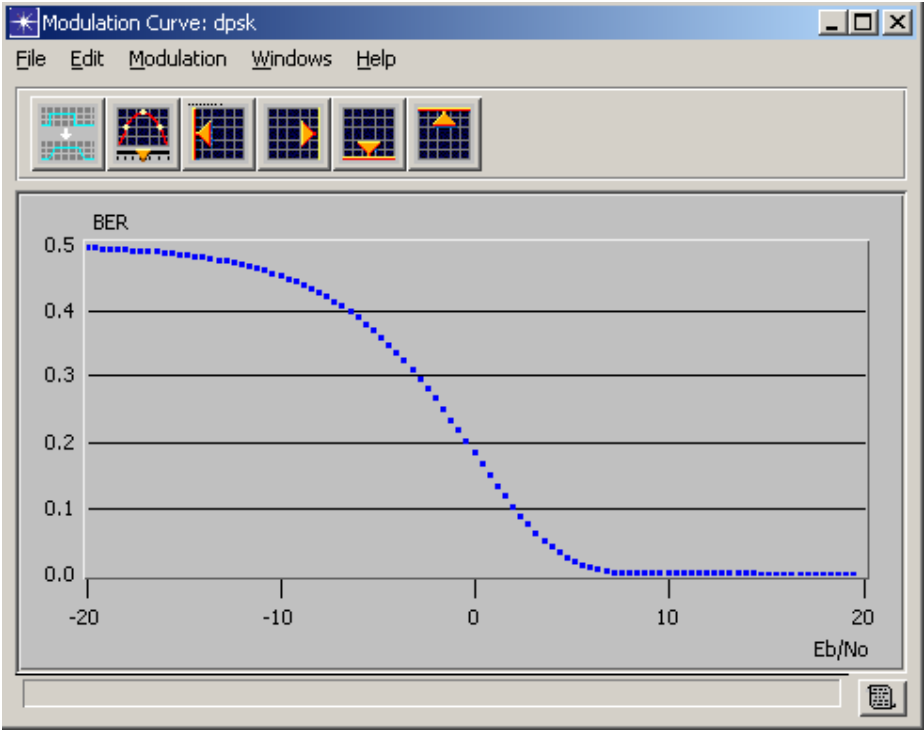
错误纠正阶段是管道的第十四个阶段，由无线接收机的“ecc model”表示。当包完成接收，在错误分布阶段返回后立即被触发执行，期间没有仿真时间过去。对每个被认为是有效的数据包(由信道匹配阶段判定)只调用该阶段一次。该阶段的目的是判定是否接收到达的包并通过信道对应的输出流转发到接收机相邻的模块中，这通常取决于是否包经历了碰撞、在错误分布阶段所计算的结果以及接收机纠正错误的能力。依据该阶段所作出的判断，内核或者销毁该包，或者允许其继续发送到目的阶段。另外，它也影响接收信道收集的错误和吞吐量结果。

仿真内核要求错误纠正阶段函数将包地址作为其唯一的参数。判断接收或丢弃包由符号常量 OPC\_TDA\_RA\_PK\_ACCEPT 表示。所指定的值应当是一个等于常量 OPC\_TURE

的整数，表示接收；否则应当指定整数值 OPC\_FALSE，表示丢弃。不允许取别的数值。

调制曲线

在无线模块中，调制曲线用于对信息编码和调制机制的描述，将误码率（BER，Bit Error Rate）作为信噪比的函数（ $E_b/N_o$ ）。



调制曲线编辑器

在调制曲线编辑器的菜单中，Modulation 是特殊的一项。在下文中对 Modulation 子菜单进行介绍。

Modulation 子菜单中的操作针对调制曲线的显示，如下表所示。

Modulation 菜单选项

Modulation 菜单	
菜单选项	描述
Smooth	平滑图形中的尖锐边缘和毛刺
Set Sampling Resolution	设置在函数中采用多少个离散的抽样点
Set Abscissa Lower Bound	设置在显示的图像中 X 轴的下限
Set Abscissa Upper Bound	设置在显示的图像中 X 轴的上限
Set Ordinate Lower Bound	设置在显示的图像中 Y 轴的下限
Set Ordinate Upper Bound	设置在显示的图像中 Y 轴的上限

除了菜单之外，在调制曲线编辑其中还包括一系列快捷按钮。如下所示：



平滑图形中的尖锐边缘和毛刺



设置在函数中采用多少个离散的抽样点

联系方式：[life\\_beauty@163.com](mailto:life_beauty@163.com)



设置在显示的图像中 X 轴的下限



设置在显示的图像中 X 轴的上限



设置在显示的图像中 Y 轴的下限

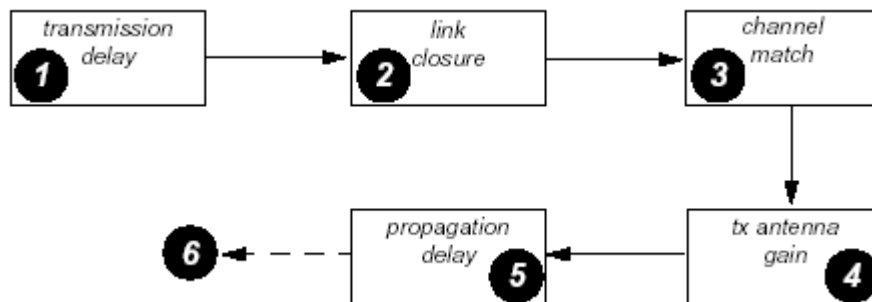


设置在显示的图像中 Y 轴的上限

调制曲线反映的是误码率和无线传输过程中信噪比之间的关系。但图形化的定义方式不够精确，因此在 OPNET Modeler 中，您可以通过 EMA 代码来创建调制曲线。

## 并行仿真

在 Modeler 无线模块中，提供了并行仿真的功能，也就是允许仿真在多台计算机的多个处理器上同时运行。由于并行仿真的出现，OPNET 可以同时执行多重计算，加快了多径无线链路的仿真速度，缩短了仿真的时间。无线发送机的管道阶段如下图所示，这是在发送包时经历的 5 个阶段。



无线发送机的管道阶段

## 并行仿真的执行

您可以在项目编辑器或仿真顺序编辑其中设置并行仿真，如下所述：

1. 在项目编辑器中设置并行仿真
  - (1) 打开项目编辑器
  - (2) 选择 Simulation/Configure Simulation 或点击相应的快捷按钮
  - (3) 在仿真设置对话框中，选中 Parallel Simulation，并说明在仿真中使用的处理器的数目。如果您选择了 Maximum，OPNET 将会在可能的情况下选择处理器的最大数目。
2. 在仿真顺序编辑器中设置并行仿真
  - (1) 打开仿真顺序编辑器
  - (2) 在工作空间中放置一个仿真图标，并打开的属性对话框。
  - (3) 选择 Parallel Simulation，说明处理器的数目。

## 并行建模和编程

仿真内核只在处理无线发送时执行并行操作，下面将对这一过程进行详细描述。

### 1. 无线管道阶段

在无线发送中，包含了 5 个阶段：发送时延，闭合阶段，信道匹配，天线增益和传播时延。在发送的过程中，发送延时只计算一次，因此不涉及在并行仿真中。一旦发送延时阶段结束后，将会为每一个潜在的接收端产生复制的包，并且每一个包都将经历后续的四个阶段。使用无线模块中的并行仿真功能，内核可以利用多 CPU 来并行的处理多个包的接收阶段。

对包的处理通过多个线程共同完成。线程的数目用 `parallel_sim.num_processors` 来说明。如果在管道阶段中没有强烈的 I/O 活动（例如对数据库的搜索），所使用的线程数目不能超过 CPU 的物理允许范围，否则将会导致性能的下降。

### 2. 多线程的安全设计

为了确保包的并行传输能够顺利而快速的进行，需要确保多线程设计的正确性，这就要保证以下条件的成立：

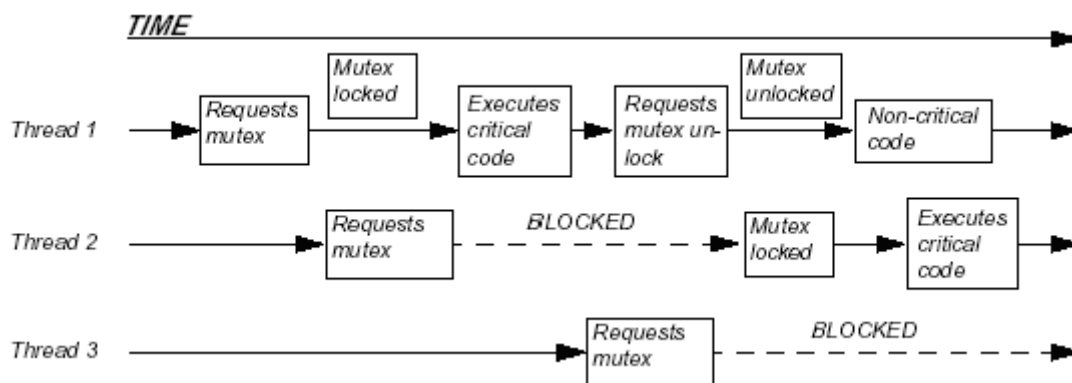
- (1) 多线程可以在同一时刻工作；
- (2) 最都有一个线程在执行过程中使用互斥的机制；
- (3) 在进程的执行过程中，有一些是可以并行的，由一些是线性执行的。

对于那些可以并行执行的代码，它们中间所涉及的元素都是这本线程可以访问的，例如在函数声明的本地变量和函数。在管道阶段中，包指针要参考被传送的复制包。如果这里的计算只包含了包和本地变量，则这一管道阶段就是可以并行的。但如果代码访问了全局的变量，公用的文件，以及其他的元素时，则可能会发生对这种共享资源的访问冲突。这时您需要确保可以获得所有所需的数值。一般来说，如果访问操作只是单纯的读操作，则不会发生问题。如果有写操作存在，则最好禁止对资源的同时访问。这时可以只用执行锁和互斥的机制。

### 3. 互斥的使用

在不能同时执行多个代码的情况下，您可以使用多种机制来确保程序运行的正确性，例如互斥机制，是在处理同类问题时一种常见的方法。

如下图所示，当线程需要访问核心的部分时，锁住可能会造成冲突的其他线程。锁打开后，其他的线程才能被处理。



多线程处理

### 4. 死锁的产生

当使用多重互斥机制时，有可能造成死锁的出现，也就是说每一个线程都处于锁定状态，等待某一线程中锁定某一资源被释放。因此在设计过程中，要避免这种情况的出现。