

Discrete-event simulation system

# Delsi

Version 1.1

Copyright © 1999 Softland (Rivne, Ukraine)

## Programmer's Guide

Rivne 1999

## CONTENTS

Introduction .....	3
Registration .....	3
Installation .....	3
1. Simulation logic .....	4
1.1. Object of <i>Delsi</i> .....	4
1.2. The main simulation algorithm .....	6
1.3. Advantages of <i>Delsi</i> simulation logic .....	7
2. Model time .....	7
3. Transactions .....	8
4. <i>Delsi</i> components .....	9
4.1. TModel .....	9
4.2. TScheduler .....	11
4.3. TBlock .....	12
4.4. TGenerator .....	14
4.5. TQueue .....	15
4.6. TStack .....	17
4.7. TQueuePrtY .....	18
4.8. TServer .....	20
4.9. TStorage .....	22
4.10. TStoragePrtY .....	24
4.11. TTerminator .....	26
4.12. TDivider .....	27
4.13. TAssembler .....	28
4.14. TCreator .....	29
4.15. TGate .....	30
4.16. TTabulator .....	31
4.17. TMultiRand .....	32
References .....	34
How to contact Softland? .....	35

## INTRODUCTION

The main idea of *Delsi* modeling is that your queuing formalization may be performed as an oriented graph with the nodes, which corresponds to some processing objects (such as generators, queues, servers etc). The arcs of the graph correspond to streams of transactions. Some user algorithms may control processing objects and transaction routing.

The simulation system is designed by use of object-oriented approach. The whole model and processing objects (so called blocks) are implemented as Borland® *Delphi*™ components. The transactions are implemented as objects.

The user algorithms are implemented as a reaction on events such as entering block, routing, exit from block and others. During the simulation process, transactions are being passed from one block to another. By use of the methods and properties of components it is possible to control behavior of the model and obtain necessary statistical results.

From the end-user's point of view, *Delsi* allows him to use all the power of Delphi (GUI, OOP, and components) as an environment for developing a wide variety of simulation models, implemented as end products. *Delsi*, in combination with a standard PC and *Delphi* creates a non-expensive simulation workplace for developing valuable applications.

## REGISTRATION

After purchasing *Delsi*, please fill registration form at [http:// www.softland.rv.ua/delsi/register.htm](http://www.softland.rv.ua/delsi/register.htm) or print file *register.pdf*, fill the form and send it to Softland. After registration you will get an opportunity for the technical support, which include:

- free fixing updates
- free upgrade from *Delsi* for Delphi™ 3.0 to the same *Delsi* version for Delphi™ 4.0 on your demand
- free downgrade from the *Delsi* for Delphi™ 4.0 to the same *Delsi* version for Delphi™ 3.0 on your demand
- free consultation concerning *Delsi* via e-mail (up to 3 messages per month and up to 3 ours of our job for answering) during one year since purchasing.
- information about new releases

Registered users may apply to Softland for developing new components. This wishes will be taken into consideration. If the component suggested by user will be implemented, this user will receive it for free. **Softland do not promise to develop the components on your wishes or suggestions.**

## INSTALLATION

How to install *Delsi*?

- Extract *delsi\_\*.zip* into some directory, for instance, C: \DELPHI\DELSI
- Start Delphi
- Choose Component\Install Packages
- Click "Add"
- For Delphi 3.0, type the full file name C: \DELPHI\DELSI\delsi.dpl
- For Delphi 4.0, type the full file name C:\DELPHI\DELSI\delsi.bpl
- Click "Open"
- Click "Ok"

You should have now a new page on your component palette called *Delsi* with new components.

## 1. SIMULATION LOGIC

### 1.1. Objects of model

*Delsi* is based on the Theory of aggregate models found in the beginning of 70's by Ukrainian Academician N.P.Buslenko. The basic unit of an aggregate model is Piecewise Linear Aggregate (PLA).

**Aggregate** is an abstract object, which functions in time. It is capable to perceive entering signals  $X$ , to produce output signals  $Y$  and to be in each instant in a condition  $Z$ . The dynamics of PLA has an "event" essence. Two key operations correspond to *Delsi* aggregates: *activation* and *odering next activation time*.

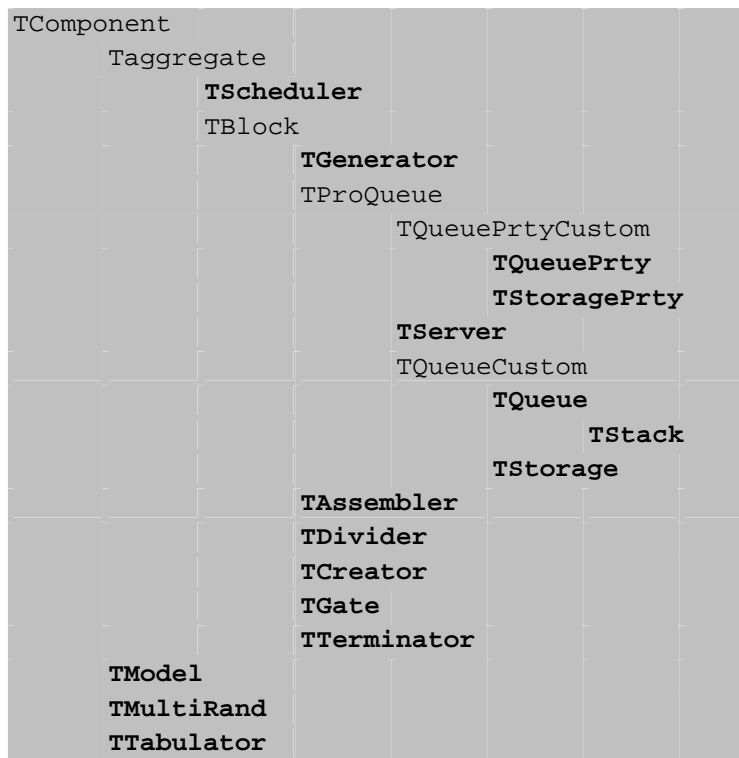
There are two kinds of aggregates in *Delsi*: *schedulers* and *blocks*.

**Scheduler** is an aggregate, which just handles some user -processed event. It does not operate with transactions.

**Block** is an aggregate which passes and receives transactions.

Each block is characterized by two states: *Ready/not ready to receive transaction* and *Ready/not ready to give transaction*. These states are necessary conditions for transaction transfer. Generally, the behavior of each block is described as a reaction to the following events: *OnEnter* (transaction is entering the block), *OnRouting* (the block is *Ready to give* and it needs to pass the exiting transaction to another block), *OnExit* (transaction leaves the block during the pass). For some blocks there are additional specific events which determine their behavior.

The inheritance of aggregates has the following structure.



**Activation** is an event initiated by changes of model time. The activation of scheduler is just initiating user-processed event. For the generator, activation is the producing of a new transaction. For servers, queues, storages it is the finishing of the waiting period for a certain transaction.

**Ordering next activation time** takes place during the handling of the certain aggregate events. For example, when the transaction leaves the generator we order next activation time (the time interval till the next generation) handling *OnExit* event. When transaction enters into the server or storage we order the next activation time (the service time) by handling *OnEnter* event. During ordering next activation time the record with the following fields will be inserted into the system calendar (List of Future Events): Block, Transaction/**nil**, Activation Time.

**FIFO rule of activation.** If several aggregates should be activated at the same time they will be activated with FIFO rule (First orderd - first activated).

**Transaction** is a dynamic object, which moves through the fixed structure of blocks. They resemble GPSS transactions but they can have an arbitrary structure.

*Delsi* has the following limitation: One transaction may be only in one block at the same time.

## 1.2. The main simulation algorithm

The simulation logic is controlled by the main simulation algorithm, which is performed in pseudo-programming language as shown below. The main cycle of the simulation algorithm scans the system calendar. In the case of *Delsi*, the system calendar is called the List of Future Events (LFE). From theoretical point of view LFE keeps the interior events of aggregates.

```

repeat
  Take the next element from the List of Future Events;
  ModelTime:=NewTime;
  Model.OnNewTime;                                // You can check what has happened
  If Aggregate is Scheduler then
    Activate(Scheduler)
  else
    begin
      Activate (Block, Transaction);
      Insert this Block into the List of Current Events;
      Using List of Current Events do all possible transaction passes;
    end;
until ModelTime<LimitTime;

```

List of Current Events (LCE) includes all blocks which are in the state *Ready to Give*. Implementation of all possible passes is controlled by the following algorithm.

```

repeat
  for all blocks in the LCE do
    begin
      Block.OnRouting;                                // Pass transaction to another block
      if not Block.ReadyToGive then
        Remove Block from the LCE;
    end;
until at least one pass have been done;

```

The user can route the transaction in the routine of *OnRouting* event in the following way:

```

if Condition then
  ThisBlock.PassTo(Block1)
else
  ThisBlock.PassTo(Block2);

```

The procedure *PassTo* does the following:

```

if AnotherBlock.ReadyToReceive(PassedTransaction) then
begin
  ThisBlock.ExitFromBlock;    // The transaction leaves this block
  ThisBlock.OnExit;           // User-defined handling
  AnotherBlock.OnEnter;       // User-defined handling
  AnotherBlock.EnterToBlock;  // The transaction enters to another block
  Model.OnAfterPass;          // User-defined handling
end;

```

### 1.3. Advantages of *Delsi* simulation logic

#### **Main simulation algorithm does not know about transactions**

As we have seen, the main simulation algorithm operates with states and events of blocks. Transactions are generated, moved and terminated during different operations with blocks. The main simulation algorithm knows almost nothing about the transactions. Thus, the speed of the main simulation algorithm does not depend on the number of transactions in the system.

#### **Direct access to transactions in the block**

Independently of the number of transactions in the block, the search and removal of certain transaction from the block is an one -step operation. Let's imagine the storage with 100,000 transactions. In some moment of the model time one of them has to be removed. The node of List of Future Events stores the address of the transaction and address of the storage. Because transactions are stored in the storage in a bidirectional list, this particular transaction will be found and removed directly. That is why the term of activation and exiting does not depend on the number of transactions in storage. The same rule applies to queues with limited waiting time. Generally, for all blocks of *Delsi* there are no cycles during entering, activation or exiting. So, the number of transactions in the model does not influence the performance.

#### **All time dependences are concentrated in LFE**

The blocks place transactions independently of their activation time. The List of Future Events is a single data structure where all time dependences are maintained. In simulation systems with intensive load, the length of LFE may reach hundreds of thousands of nodes. That is why LFE maintenance plays such a decisive role in the performance. *Delsi* is designed in such way that I/O performance of LFE has binary -logarithmic dependence on its length.

## 2. MODEL TIME

The model time is a time of model functioning. It is not physical time. Before simulation start -up and after model reset the model time is equal to 0.

```
function ModelTime: real;
```

Returns current model time.

### 3. TRANSACTIONS

#### Declaration

```
TTransaction = class(TObject)
```

#### Methodes

```
procedure TTransaction.SetPrty(Prty: byte);
```

Sets priority level *Prty* for the transaction

```
procedure TTransaction.SetPreempt;
```

Makes the transaction preemptive.

```
procedure TTransaction.SetNonPreempt;
```

Makes the transaction non -preemptive.

```
function TTransaction.GetPrty: byte;
```

Returns the priority of the transaction.

```
function TTransaction.IsPreempt: boolean;
```

Returns *True* if the transaction is preemptive. Otherwise returns *False*.

```
function TTransaction.GetTransID: longint;
```

Returns the transaction ID. IDs of all “alive” transactions are unique but newly generated transaction may repeat the ID of terminated one.

#### Inheritance

It is very important that you can redefine *TTransaction* to obtain the new customized fields. For instance,

```
MyTransaction = class (TTransaction);
public
    MyField1: integer;
    MyField2: real;
end;
```

When you redefine *TTransaction* you have to inform the internal simulation manager about that new redefinition. Do this by calling method *TModel.SetTransactionClass* before start of simulation:

```
Model.SetTransactionClass(MyTransaction);
```

When you need to use your own fields, do this in the following way.

```
procedure TForm1.EntranceExit(Sender: TBlock; Trans: TTransaction);
begin
    . . .
    (Trans as MyTransaction).MyField2:=ModelTime;
end;
```



## 4. *Delsi* COMPONENTS

### 4.1. TModel

#### Declaration

```
TModel = class(TComponent)
```

#### Purpose

*TModel* is a basic component, which is responsible for the whole model functioning. With help of *TModel* we can start and stop simulation the process or clear statistics collected in blocks. One more useful feature of *TModel* is the possibility of registration of transaction passes and fixing the moments of time changes. There can be only one *TModel* component in an application.

#### Properties

```
property HeapSize: byte;
```

Before simulation start-up the application allocates a cut of memory. In *HeapSize* you can set the size of allocated memory in Megabytes. The range of possible values is from 1 to 128. The default value is 4.

```
property PageSize: integer;
```

All dynamic objects of *Delsi*, like transactions, elements of LFE and LCE are stored in memory pages. This property sets the page size in bytes. The range of possible values is from 1024 to 32768. The default value is 1024.

#### Methods

```
procedure ClearStatistics;
```

Clears statistics of all blocks. As a rule, stationary stochastic systems pass a phase of transient in their initial period of time. If the contributor wants to evaluate parameters of a system in its stable state, the transitional process brings an error to outcomes. We can cut off the errors by clearing statistics after the transient phase.

```
procedure Reset;
```

This procedure resets the model into its initial state so, that it becomes ready for the next run. Use this procedure when you need several simulation runs with changing input parameters.

```
Procedure SetTransactionClass(TransClass: Tclass);
```

You need this method when you redefine class TTransaction to obtain your own customized structure of transaction. Use this procedure before simulation run (i.e. before first calling *TModel.Simulate*).

```
procedure Simulate(TLimit: real);
```

Starts simulation run. The simulation will proceed until reaching a value *TLimit* by model time.

```
procedure Stop;
```

Stops simulation run.

## Events

```
TAfterPassEvent = procedure(Sender: TBlock;    // sending block
                             Receiver: TBlock; // receiving block
                             Trans: TTransaction // passed transaction
                             of object;
property OnAfterPass: TAfterPassEvent;
```

The event occurs after the transaction is passed from one block to another.

```
TBeforeModelGoOnEvent = procedure of object;
property OnBeforeTimeGoOn: TBeforeModelGoOnEvent;
```

This event occurs before model running and before blocks' events OnBeforeTimeGoOn. So, it's the first Delsi event at all.

```
TDeadlock = procedure of object;
property OnDeadlock: TDeadlock;
```

This event is initiated after detection of deadlock in the simulated system. Deadlock in *Delsi* means the List of Future Events is empty.

```
TNewTimeEvent = procedure(Aggregate: Taggregate; // activated aggregate
                           Trans: TTransaction) // nil or activated transaction
                           of object;
property OnNewTime: TNewTimeEvent;
```

The event occurs when model time changes the value. Parameter *Aggregate* refers to activated aggregate (block or scheduler). *Trans* is the transaction, which should leave the block (server, queue or storage). If the activated aggregate is a generator or scheduler, *Trans* is equal to **nil**.

## 4.2. TScheduler

### Declaration

```
TScheduler = class(TAggregate)
```

### Purpose

*TScheduler* is intended to initiate the events through some intervals of simulation time. We need it to control the simulation process. By use of this event, the user can change the parameters of the model, stop simulation run, clear statistics or lock/unlock gates.

### Methods

```
procedure NextTime(ActivationTime: real);
```

Sets the time to the next activation.

### Events

```
TBeforeTimeGoOnEvent = procedure(Sender: TAggregate) of object;  
property OnBeforeTimeGoOn: TBeforeTimeGoOnEvent;
```

The event occurs before simulation run. We can use it to do any user -defined action before simulation, for example, to initialize some parameters. This event is especially useful for planning the first user-defined procedure used during simulation run. We plan the next event using method *NextTime*.

```
TPlannedEvent = procedure(Sender: TAggregate) of object;  
property OnPlanned: TPlannedEvent;
```

This event is a result of activation, which happens after expiration of *ActivationTime*. By handle of this event we can do any possible changes for the model. If you need to order the next activation, use *NextTime* again.

### 4.3. TBlock

#### Declaration

```
TBlock = class(TAggregate)
```

#### Purpose

TBlock is an ancestor of all *Delsi* blocks. Here we describe methods and events common for all blocks.

#### Methods

```
procedure ClearStatistics;
```

Clears accumulated statistics in the block. When *TModel.ClearStatistics* is being executed, it calls the method *ClearStatistics* for all blocks of a model.

```
function Count: longint;
```

Returns the current number of transactions in the block.

```
function Entries: longint;
```

Returns the total number of transactions, which have entered into the block.

```
function Exits: longint;
```

Returns the total number of transactions, which have exited from the block.

```
function IsReadyToReceive(Trans: TTransaction): boolean;
```

Returns *True* if the block is ready to receive transaction pointed by *Trans*.

```
function Pass(Block: TBlock): boolean;
```

Passes current transaction to the block *Block*. If transaction is successfully passed the function returns *True*. Use this method only when you handle *OnRouting*, *OnPreempt*, *OnRelease* and *OnTimeFinish* events.

#### Events

```
TOnEnterToBlock = procedure(Sender: TBlock;      // This block
                           Trans: TTransaction) // Entering transaction
                           of object;
```

```
property OnEnter: TOnEnterToBlock;
```

The event occurs when transaction enters into the block. It is not applicable for *TGenerator*.

```
TOnExitFromBlock = procedure(Sender: TBlock;      // This block
                             Trans: TTransaction) // Exiting transaction
                             of object
```

```
property OnExit: TOnExitFromBlock;
```

The event occurs when transaction leaves the block. It is not applicable for *TTerminator*.

---

```
TOnRoutingEvent = procedure(Sender: TBlock;      // Sending block (this block)
                             Trans: TTransaction// Passed transaction
                             of object;  
property OnRouting: TOnRoutingEvent;
```

Processing this event you can route transaction to another block with the help of *Pass* method. By using Delphi's power you can create very sophisticated procedures of routing.

## 4.4. TGenerator

### Declaration

```
TBlock = class(TBlock)
```

### Purpose

TGenerator produces one transaction during the activation. It's commonly used source of transactions in the model. We use it when we need to simulate arrival of served elements. Usually, the arrivals have some intensity described by probability distribution. For simulation of sequential arrivals, when transaction leaves the generator we need to order next activation, i.e. next transaction generation. Appropriating a value *ActivationTime* random numbers, we simulate statistical properties of arrival intensity.

### Methods

```
function AverageTime: extended;
```

Returns the average time between the moments when transactions leave the generator.

```
function DeviationTime: extended;
```

Returns the standard deviation of time between the moments when transactions leave the generator.

```
procedure NextTime(ActivationTime: real);
```

Sets the time to the next activation (generation of transaction).

### Events

```
TOnAfterGenerationEvent = procedure(Sender: TBlock; Trans: TTransaction) of object;  
property OnAfterGeneration: TOnAfterGenerationEvent;
```

The event occurs right after generation of transaction. It's a good time to set the transaction properties like priority, preemptive property or fields defined by user.

```
TBeforeTimeGoOnEvent = procedure(Sender: TAggregate) of object;  
property OnBeforeTimeGoOn: TBeforeTimeGoOnEvent;
```

The event occurs before simulation run. We use it to order the time of generation of the first transaction. Certainly, we do it with help of *NextTime* method.

```
property OnExit: TOnExitFromBlock;
```

The event occurs when transaction leaves the block. It is a good moment to order the next transaction generation by use of *NextTime* method.

```
property OnRouting: TOnRoutingEvent;
```

See 3.3. Tblock

## 4.5. TQueue

### Declaration

```
TQueue = class(TQueueCustom)
```

### Purpose

This component simulates simple FIFO queue. It handles transactions in accordance with the rule “First input – first output”. The queue may be limited by capacity.

Another useful feature is the possibility to limit a waiting period in the queue. You can do this by the ordering the activation when a transaction enters the queue.

If the number of transactions in the queue is less than its capacity, the queue is in the state of “*Ready to Receive*”. If the queue keeps one or more transactions, it is in the state of “*Ready to Give*”.

### Properties

```
property Capacity: longint;
```

*Capacity* is the greatest possible number of transactions in the queue. If the number of transactions is equal to capacity, the queue is “*Not Ready to Receive*”. If Capacity is equal to 0, the possible number of transactions in the queue is unlimited.

### Methods

```
function AverageCount: extended;
```

Returns the average number of transaction in the queue (average length).

```
function AverageTime: extended;
```

Returns the average time spent in the queue.

```
function Count: longint;
```

See 3.3. TBlock

```
function DeviationTime: extended;
```

Returns the standard deviation of the time spent in the queue.

```
function IsReadyToReceive(Trans: TTransaction): boolean;
```

Returns *True* if the queue is “*Ready to Receive*” transaction *Trans*.

```
function MaxCount: longint;
```

Returns maximal number of transactions in the queue (maximal length).

```
procedure NextTime(ActivationTime: real);
```

Limits the waiting time for entering transaction by the value of *ActivationTime*. Use this method only when handle *OnEnter* event.

```
function SAverageTime: extended;
```

Returns average non-zero time spent by transactions in the queue.

```
function SDeviationTime: extended;
```

Returns the standard deviation of the non -zero time spent by transactions in the queue.

```
function TimeLimitExits: longint;
```

Returns the number of transactions, which have abandoned the queue because of the end of waiting period.

```
function Usage: extended;
```

Returns a relative part of the time when the queue was in use.

```
function ZeroEntries: longint;
```

Returns the number of transaction entries into the queue, when the queue was empty.

## Events

```
property OnEnter: TOnEnterToBlock;
```

See 3.3. TBlock. If you want to limit the waiting time in the queue, do this by processing this event with the help of *NextTime* method.

```
property OnExit: TOnExitFromBlock;
```

See 3.3. TBlock.

```
property OnRouting: TOnRoutingEvent;
```

See 3.3. TBlock

```
TOnTimeFinish = procedure(Sender: TBlock;           // This queue
                           Trans: TTransaction) // The transaction whose time is finished
                           of object;
property OnTimeFinish: TOnTimeFinish;
```

This event occurs when admissible latency period for the transaction is finished. We call this moment “activation of the queue”. Here you can determine what will happen further with the transaction. One of alternatives is to pass the transaction to another block. If you do n't pass the transaction to another block, or if you don't handle this event at all, the transaction will be terminated.



## 4.6. TStack

### Declaration

```
TStack = class(TQueue)
```

### Purpose

This component simulates simple LIFO queue. It handles transactions in accordance with the rule “Last input – first output”. Besides, this component repeats all properties, methods and events of *TQueue*.

## 4.7. TQueuePrty

### Declaration

```
TQueuePrty = class(TQueuePrtyCustom)
```

### Purpose

*TQueuePrty* differs from *TQueue* by the discipline of transaction keeping. It keeps them in accordance with the rule “First input – first output in its priority level”. So, the transactions with higher priority will leave the queue first.

Another distinguishing feature of *TQueuePrty* is the following. Let's imagine that the queue is full, i.e. the number of transactions is equal to the queue capacity, and a transaction tries to enter into the queue. Assume that this transaction has preemptive priority with the level higher than the lowest priority of transactions in the queue. In this case high -priority preemptive transaction will displace the lowest - priority transaction. This operation is called *preempting*. Preempted (displaced) transactions may be passed to the other blocks or terminated depending on handling procedure of *OnPreempt* event.

### Properties

```
property Capacity: longint;
```

See 3.5. TQueue

```
property Preemptive: boolean;
```

The preempting is possible if *Preemptive* is *True*.

### Methods

```
function AverageCount: extended;
```

See 3.5. TQueue

```
function AverageTime: extended;
```

See 3.5. TQueue

```
function Count: longint;
```

See 3.3. TBlock

```
function DeviationTime: extended;
```

See 3.5. TQueue

```
function IsReadyToReceive(Trans: TTransaction): boolean;
```

Returns *True* if the queue is “*Ready to Receive*” transaction *Trans*. It takes place in the following cases:

- The capacity is unlimited;
- The number of transactions in the queue is less then its capacity;
- Entering transaction pointed by *Trans* has preemptive priority higher than the priority of at least one transaction in the queue.

**function** MaxCount: longint;

See 3.5. TQueue

**procedure** NextTime(ActivationTime: real);

See 3.5. TQueue

**function** PreemptExits: longint;

Returns the number of transactions, which have abandoned the queue due to priority preempting.

**function** SAverageTime: extended;

See 3.5. TQueue

**function** SDeviationTime: extended;

See 3.5. TQueue

**function** TimeLimitExits: longint;

Returns the number of transactions, which have abandoned the queue because of the end of admissible waiting period.

**function** Usage: extended;

See 3.5. TQueue

**function** ZeroEntries: longint;

See 3.5. TQueue

## Events

**property** OnEnter: TOnEnterToBlock;

See 3.3 TBlock, 3.5 TQueue

**property** OnExit: TOnExitFromBlock;

See 3.3. TBlock

TOnPreempt = **procedure**(Sender: TBlock;                    *// This queue*  
                            Trans: TTransaction)       *// Preempted transaction*  
                            **of object;**

**property** OnPreempt: TOnPreempt;

Handling this event, you can determine what will happen with the preempted low -priority transaction. You can pass preempted transaction to another b lock. If you don't pass the transaction or if you don't handle this event at all, the transaction will be terminated.

**property** OnRouting: TOnRoutingEvent;

See 3.3. TBlock

**property** OnTimeFinish: TOnTimeFinish;

See 3.5. TQueue

## 4.8. TServer

### Declaration

```
TServer = class(TProQueue)
```

### Purpose

*TServer* simulates a serving process. In every moment of time only one transaction can be served. The transaction is being served during the time defined by *NextTime* method in the procedure of handling *OnEnter* event.

As *TQueuePrty* component, *TServer* can handle preempting. Preemptive high -priority transaction preempts the service of low -priority transaction. The preempted transaction may be passed to another block or terminated. The third alternative is that the serving preempted transaction may be *postponed*. When a high -priority transaction leaves the server, the postponed transaction will be reset on serving for the rest of service time. The server can store only one postponed transaction per priority level. Postponed transactions are kept in the stack ordered by priority. Note, there is a difference between the number of transactions which is being served (may be 0 or 1) and the number of transactions in the server (may be equal up to number of priority levels in your model).

### Properties

```
property Preemptive: boolean;
```

The preempting is possible if *Preemptive* is *True*.

### Methods

```
function AverageCount: extended;
```

Returns the average number of transactions in the server.

```
function AverageTime: extended;
```

Returns the average time spent by transactions in the server.

```
function Count: longint;
```

See 3.3. TBlock

```
function DeviationTime: extended;
```

Returns the deviation of time spent by transactions in the server (including the time of postponing).

```
function IdleForExit: boolean;
```

Returns *True* if the current transaction in the server is already served. In this case the transaction is just standing idle to exit.

```
function IsReadyToReceive(Trans: TTransaction): boolean;
```

Returns *True* if the server is “*Ready to Receive*” transaction *Trans*. It takes place in the following cases:

- The server is empty;
- The entering transaction pointed by *Trans* has preemptive priority higher than the priority of the transaction, which is served.

```
function MaxCount: longint;
```

Returns the maximal number of transactions in the server.

```
procedure NextTime(ActivationTime: real);
```

Sets the service time for the transaction entering into the server. The service time is equal to *ActivationTime*. Use this method while handle *OnEnter* event.

```
procedure Postpone;
```

Postpone the service of preempted transaction. Use this method only handling *OnPreempt* event.

```
function PreemptExits: longint;
```

Returns number of transactions, which have abandoned the server due to priority preempting.

```
function Usage: extended;
```

Returns a relative part of the time when the server was in use.

## Events

```
property OnEnter: TOnEnterToBlock;
```

See 3.3. TBlock. The event occurs when transaction enters into the block. It's the time to define the service time using *NextTime* method.

```
property OnExit: TOnExitFromBlock;
```

See 3.3. TBlock

```
TOnPreempt = procedure(Sender: TBlock;           // This server
                       Trans: TTransaction)      // Preempted transaction
of object;
```

```
property OnPreempt: TOnPreempt;
```

Handling this event, you can determine what will happen with the preempted low -priority transaction. You can pass transaction to another block or postpone its service. If you don't do it or if you don't handle this event at all, the transaction will be terminated.

```
property OnRouting: TOnRoutingEvent;
```

See 3.3. Tblock

## 4.9. TStorage

### Declaration

```
TStorage = class(TQueueCustom)
```

### Purpose

*TStorage* may be described as a server able to serve several transactions simultaneously. It is necessary to notice, that *TStorage* does not support preempting.

### Properties

```
property Capacity: longint;
```

*Capacity* is the greatest possible number of transactions, which could be served in the storage. If the number of transactions is equal to capacity, the storage is “*Not Ready to Receive*”. If *Capacity* is equal to 0, the possible number of transactions in the storage is unlimited.

### Methods

```
function AverageCount: extended;
```

Returns the average number of transactions in the storage. You can determine the average filling of the storage as a ratio of *AverageCount* to *Capacity*.

```
function AverageTime: extended;
```

Returns the average time spent by transactions in the storage.

```
function Count: longint;
```

See 3.3. TBlock

```
function DeviationTime: extended;
```

Returns the deviation of time spent by transactions in the storage.

```
function IsReadyToReceive(Trans: TTransaction): boolean;
```

Returns *True* if the storage is “*Ready to Receive*” transaction *Trans*. The function returns *True* if the *Capacity* is 0 or the number of transactions is less than *Capacity*.

```
function MaxCount: longint;
```

Returns the maximal number of transactions in the storage.

```
procedure NextTime(ActivationTime: real);
```

Sets the service time for the transaction entering the storage. Use this method handling *OnEnter* event.

```
function Usage: extended;
```

Returns the relative part of time when the storage was in use.

## Events

**property** OnEnter: TOnEnterToBlock;

The event occurs when transaction enters into the storage. It's the time to define the service time by use of *NextTime* method.

**property** OnExit: TOnExitFromBlock;

See 3.3. TBlock

**property** OnRouting: TOnRoutingEvent;

See 3.3. TBlock

## 4.10. TStoragePrty

### Declaration

```
TStoragePrty = class(TQueuePrtyCustom)
```

### Purpose

Additionally to the possibilities of *TStorage*, this component supports priority preempting. The preempting in *TStoragePrty* is similar to preempting in *TQueuePrty* and *TServer*. In contrast to *TServer*, it does not support the postponed service.

### Properties

```
property Capacity: longint;
```

See 3.9. TStorage

```
property Preemptive: boolean;
```

The preempting is possible if *Preemptive* is *True*.

### Methods

```
function AverageCount: extended;
```

See 3.9. TStorage

```
function AverageTime: extended;
```

See 3.9. TStorage

```
function Count: longint;
```

See 3.3. TBlock

```
function DeviationTime: extended;
```

See 3.9. TStorage

```
function IsReadyToReceive(Trans: TTransaction): boolean;
```

Returns *True* if the queue is “*Ready to Receive*” transaction *Trans*. The function returns *True* in the following cases:

- The capacity is unlimited;
- The number of transactions in the storage is less then its capacity;
- The entering transaction pointed by *Trans* has preemptive priority higher than priority of the transaction, which is being served.

```
function MaxCount: longint;
```

See 3.9 TStorage



```
procedure NextTime(ActivationTime: real);
```

See 3.9. TStorage

```
function PreemptExits: longint;
```

Returns the number of transactions, which have abandoned the storage due to priority preempting.

```
function Usage: extended;
```

See 3.9. TStorage

## Events

```
property OnEnter: TOnEnterToBlock;
```

See 3.3. TBlock, 3.9. TStorage

```
property OnExit: TOnExitFromBlock;
```

See 3.3. TBlock

```
TOnPreempt = procedure(Sender: TBlock;           // This storage
                       Trans: TTransaction)       // Preempted transaction
                       of object;
```

```
property OnPreempt: TOnPreempt;
```

Handling this event, you can determine what will happen with the preempted low -priority transaction. You can pass preempted transaction to another block. If you don't pass the transaction or if you don't handle this event at all, the transaction will be terminated.

```
property OnRouting: TOnRoutingEvent;
```

See 3.3. TBlock

## 4.11. TTerminator

### Declaration

```
TTerminator = class(TBlock)
```

### Purpose

Transactions enter into this block to be terminated. When you handle *OnEnter* event, the transaction is still accessible and you have a possibility to get some information about it. *TTerminator* is always “*Ready to Receive*”.

### Events

```
property OnEnter: TOnEnterToBlock;
```

See 3.3. TBlock.

## 4.12. TDivider

### Declaration

```
TDivider = class(TBlock)
```

### Purpose

Splits the arrived transaction into several transactions. The new transactions have the same priority and ability to preempt just as their parent. When divider emits transactions, parent transaction goes last. This block is “*Ready to Receive*” and “*not Ready to Give*” if it is empty. If it is not empty, it is “*not Ready to Receive*” and “*Ready to Give*”.

With the purpose of routing you may need to distinguish parental and generated transactions. You can do this by handling *OnExit* or *OnRouting* events. If *TDivider.Count* returns 1, it's a parental transaction.

### Methods

```
function IsReadyToReceive(Trans: TTransaction): boolean;
```

See 3.3. TBlock

### Properties

```
property Capacity: longint;
```

The number of the output transactions obtained in the outcome of splitting.

### Events

```
property OnEnter: TOnEnterToBlock;
```

See 3.3. TBlock

```
property OnExit: TOnExitFromBlock;
```

See 3.3. TBlock

```
property OnRouting: TOnRoutingEvent;
```

See 3.3. Tblock

### 4.13. TAssembler

#### Declaration

```
TAssembler = class(TBlock)
```

#### Purpose

TAssembler assembles several arrived transactions into one. Actually, the first entering transaction remains alive. The rest of the transactions will be terminated.

This block is “*Ready to Receive*” and “*not Ready to Give*” until the resulting transaction is assembled. When transaction is assembled, TAssembler is “*not Ready to Receive*” and “*Ready to Give*”.

#### Properties

```
property Capacity: longint;
```

Sets the number of the input transactions necessary to obtain the assembled transaction.

#### Methods

```
function IsReadyToReceive(Trans: TTransaction): boolean;
```

See 3.3. TBlock

#### Events

```
property OnEnter: TOnEnterToBlock;
```

See 3.3. TBlock

```
property OnExit: TOnExitFromBlock;
```

See 3.3. TBlock

```
property OnRouting: TOnRoutingEvent;
```

See 3.3. Tblock

## 4.14. TCreator

### Declaration

```
TCreator = class(TBlock)
```

### Purpose

Creates the transactions on demand. After creation of transactions the component is “*Ready to Give*” as long as transactions are being emitted. In a combination with *TScheduler* you can simulate the generator, which produces portions of the transactions.

### Methods

```
procedure Generate(Number: longint);
```

Immediately generates *Number* transactions and order the next activation time equal to 0. Thus, after the generation simulation manager immediately tries to move generated transactions through the model.

### Events

```
property OnAfterGeneration: TonAfterGenerationEvent;
```

The event occurs right after generation of transaction. It's a good time to set the transaction properties like priority, preemptive property or fields defined by user. (See 3.4. TGenerator.)

```
property OnExit: TOnExitFromBlock;
```

See 3.3. TBlock

```
property OnRouting: TOnRoutingEvent;
```

See 3.3. TBlock

## 4.15. TGate

### Declaration

```
TGate = class(TBlock)
```

### Purpose

This block is a simple gate. You can lock, unlock and inverse the gate. If gate is unlocked it works like simple server with zero service time. By locking the gate you discontinue promoting of the transactions that go through it.

### Methods

```
procedure Inverse
```

Locks the gate if it is unlocked and vice -verse.

```
function IsLocked: boolean;
```

Returns *True* if the gate is locked and vice -verse.

```
procedure Lock;
```

Sets the gate into the “*not Ready to Receive*” state. If there is transaction in the gate in the locking moment, this transaction will leave the gate whenever it will be possible.

```
procedure Unlock;
```

Unlocks the gate.

### Events

```
property OnEnter: TOnEnterToBlock;
```

See 3.3. TBlock

```
property OnExit: TOnExitFromBlock;
```

See 3.3. TBlock

```
property OnRouting: TOnRoutingEvent;
```

See 3.3. Tblock

## 4.16. TTabulator

### Declaration

```
TTabulator = class(TComponent)
```

### Purpose

This component is not an aggregate. It is designed for statistical purposes only. If you need to build histogram of some output parameter, this component helps you to calculate the hits of its values in numerical intervals. Beside that, *TTabulator* calculates the average value of the parameter and its standard deviation.

### Properties

```
property Interval: real;
```

Width of a numerical interval.

```
property IntervalCount: byte;
```

Number of intervals. The range of admissible values is from 2 to 254.

```
property LowerBound: real;
```

Lower bound of the first interval.

### Methods

```
function Count: longint;
```

Returns the total number of tabulated values.

```
function Deviation: real;
```

Returns the standard deviation of tabulated values.

```
function Hits(Index: byte): longint;
```

Returns the number of hits in *Index* interval. If *Index* is equal to 0, the function returns the number of hits lower than *LowerBound*. If *Index* is equal to *IntervalCount*+1, the function returns the number of hits higher than the upper bound of the last interval.

```
function Mean: real;
```

Returns the average value of tabulated values.

```
procedure PutValue(Value: real);
```

Inputs value for tabulation.

```
procedure Resets;
```

Resets all accumulated data.

## 4.17. TMultiRand

### Declaration

```
TMultiRand = class(TComponent)
```

### Purpose

This component is intended to generate random variates of different distributions.

### Properties

The basement for obtaining all distributions is uniform distribution from 0 to 1. The prime modulus multiplicative congruential random number generator (PMMCG) is used to obtain uniformly distributed values. The algorithm of PMMCG is the following.

```
MODULUS: = 2147483647;      // 2^31-1
Seed:= (Multiplier * Seed) mod MODULUS;
Result:=Seed / MODULUS;
```

```
property Seed: comp;
```

This property sets the initial seed for PMMCG. The value of *Seed* should be in the range from 2 to 2147483646. The default value is 1000000000.

```
property Multiplier: comp;
```

This property sets the multiplier for PMMCG. The value of *Multiplier* should be in the range from 2 to 2147483646. The default value is 950706376.

Fishman and Moore (1986) recommend the best values of *Multiplier*.

```
950,706,376
742,938,285
1,226,874,159
62,089,911
1,343,714,438
```

### Methods

```
function Beta(ShapeAlpha: real; ShapeBeta: real;
              LowerBound: real; UpperBound: real): real;
```

Returns Beta distributed variates with the following parameters and limitations.

*ShapeAlpha* –  $\alpha$ -shape, *ShapeAlpha* > 0;

*ShapeBeta* –  $\beta$ -shape, *LowerBound* > 0;

*LowerBound* – Lower bound of distribution, *LowerBound* >= 0;

*UpperBound* – Upper bound of distribution, *UpperBound* > 0, *LowerBound* <= *UpperBound*.

Method: Transformation of random variates where sample from Beta distribution is some ratio of two Gamma-distributed samples [4].



```
function Gamma(Mean: real; Alpha: real): real;
```

Returns Gamma-distributed variates.

*Mean* – the mean of distribution,  $Mean \geq 0$ ;

*Alpha* – the  $\alpha$ -shape parameter,  $Alpha > 0$ .

Method: For  $\alpha < 1$ , Jonk's method is used [7]. For  $\alpha > 1$  the function uses combination of Gamma distribution for  $\alpha < 1$  and Erlang distribution. This method is suggested by *Valeria Derevenskaya*,

```
function Erlang(Mean: real; M: integer): real;
```

Returns variates with Erlang distribution. Distribution has the following parameters.

*Mean* – the mean of distribution,  $Mean \geq 0$ ;

*M* – the shape parameter,  $M > 0$ .

Method: Summarizing *M* exponential variates; each of them is exponentially distributed with the mean equal to  $Mean/M$  [7].

```
function Exponential(Mean: real): real;
```

Returns variates exponentially distributed with the mean *Mean*, where  $Mean \geq 0$ .

Method: Inverse transformation [7].

```
function Lognormal(Mean: real; Deviation: real): real;
```

Returns variates with lognormal distribution. *Mean* and *Deviation* parameters are the mean and standard deviation of the distribution; both parameters should be greater than 0.

Method: For the equation  $L = \exp(N)$ , if *N* is normal distributed variates then *L* is lognormal distributed [2].

```
function Normal(Mean: real; Deviation: real): real;
```

Returns variates with normal distribution. The mean and standard deviation of the variates are set in the parameters *Mean* and *Deviation*; both parameters should be greater than 0.

Method: Transformation of random variables with a selective truncation was used for obtaining normal variates [1,3].

```
procedure Reset;
```

Resets the component into initial state. It sets *Seed* into initially assigned value and resets some internal triggers.

```
function Triangular(Min: real; Mode: real; Max: real): real;
```

Returns value with triangular distribution. Distribution has the following parameters.

*Min* – minimal value,  $Min \geq 0.0$ ;

*Max* – maximal value,  $Max \geq 0.0$ ;

*Mode* – mode of distribution,  $Mode \geq 0.0$ ,  $Min \leq Mode \leq Max$ .

Method: Inverse transformation [6].

```
function Uniform(LowerBound: real; UpperBound: real): real;
```

Returns variates uniformly distributed between *LowerBound* and *UpperBound*.

Both parameters should be equal or greater than 0 and  $LowerBound \leq UpperBound$ .

```
function Weibull(Alpha: real; Scale: real): real;
```

Returns variates with Weibull distribution, which has the following parameters.

*Alpha* – shape,  $Alpha > 0$

*Scale* – scale,  $Scale > 0$

Method: Inverse transformation [5].

## References

1. Ahrens, J.H. and U.Dieter, "Computer Methods for Sampling from the Exponential and Normal Distributions", *Comm. ACM*, Vol. 15, 1972, pp. 873-882.
2. Aitchison, J. and J.A.C. Brown, *The Lognormal Distribution*, Cambridge Press, 1957
3. Box, G.E.P. and M.A.Miller, "A Note on the Generation of Random Normal Deviates", *Annals of Math Stat.*, Vol. 29, 1958, pp. 610-611.
4. Fishman, G.S., *Principles of Discrete Event Simulation*, John Wiley, 1978.
5. Hahn, G.J. and S.S.Shapiro, *Statistical Models in Engineering*, John Wiley, 1967
6. Pritsker, A.A.B., *The GASP IV Simulation Language*, John Wiley, 1974
7. Pritsker, A.A.B., *Introduction to simulation and SLAM II*, John Wiley, 1984

**How to contact Softland?**

If you have any questions or comments, please, feel free to contact us.

e-mail: [german@rivne.com](mailto:german@rivne.com)

Web: <http://www.softland.rv.ua> or <http://www.softland.rovno.ua>

Address: Softland  
Soborna Street, 1  
Rivne  
Ukraine 33028

Tel: +380 362 261570

Fax: +380 362 265441