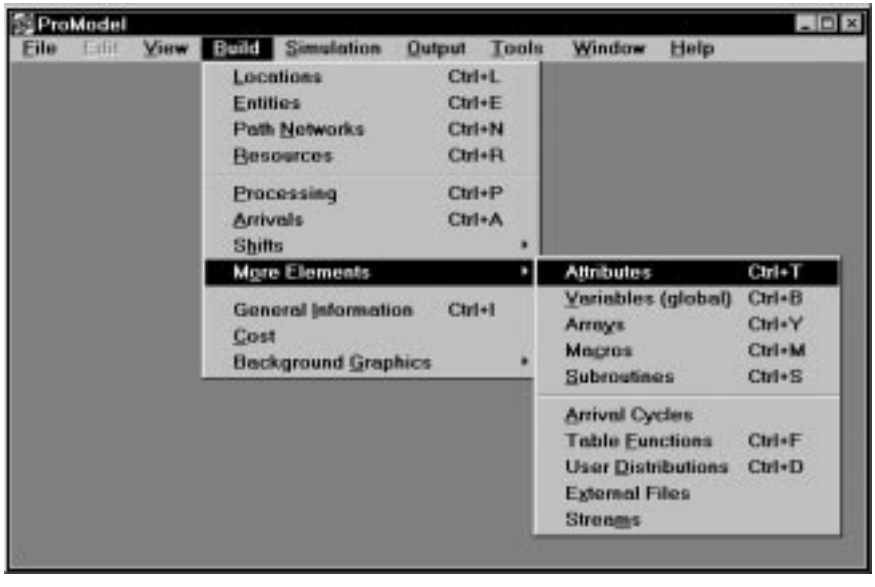
Chapter 8 Defining Advanced Elements

CHAPTER CONTENTS

8.1 Attributes

Attributes are place holders similar to variables, but are attached to specific locations and entities and usually contain information about that location or entity. Attributes may contain integers or real numbers. You may also assign model element names (e.g., StationA) to an attribute, which is stored as the element's index number but may be referenced by name. Attributes are defined through the Build menu as shown below.



How To **Create and edit attributes:**

1. Select **More Elements** from the **Build** menu.
2. Select **Attributes**.

8.1.1 Attribute Types

Attributes are classified as follows:

Entity Attributes

Entity attributes are place holders assigned to an entity and contain numerical information about that entity. An entity attribute is identified by its name and may be assigned a value or model element name stored as a value. An entity attribute may be examined and acted upon in any of the following places:

- Arrival logic
- Operation logic
- Move logic which refers to the attribute of the entity being routed
- Min or Max attribute rules for locations and resources
- Routing quantity
- Routing destination priority
- Resource pick up and drop off times
- Entity speed
- Debug condition

Location Attributes

Location attributes are place holders assigned directly to a location and contain numerical information about that location. A location attribute is identified by its name and may be assigned a value or model element name stored as a value. A location attribute may be examined and acted on in any of the following places:

- Arrival logic
- Operation logic
- Move logic
- Min or Max attribute for selecting incoming entities as a location rule
- Routing quantity
- Routing destination priority
- Resource pick up and drop off times
- Location down time logic
- Conveyor speed
- Debug condition

8.1.2 Memory Allocation for Attributes

Because locations always exist during a simulation, location attributes always use memory. However, attributes for entities are not created until the first time any of the entity's attributes are examined or set. At that time, ProModel allocates enough memory for all of the entity's attributes. When the entity exits the system or is absorbed by another entity through a COMBINE or JOIN statement, ProModel automatically frees the memory allocated for its attributes.

8.1.3 Attributes vs. Local Variables

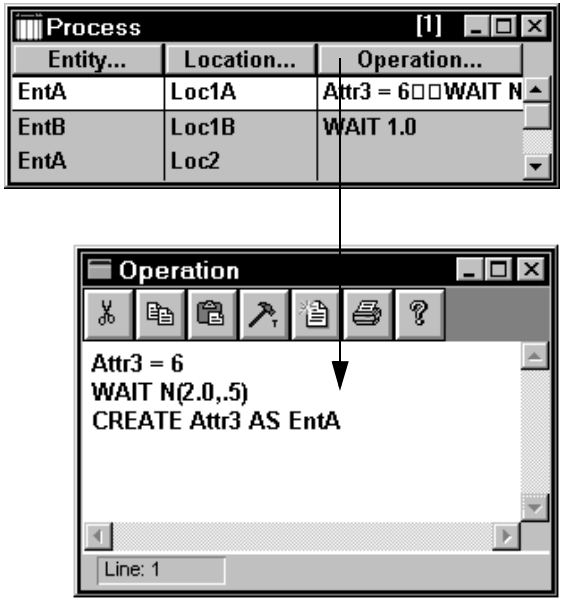
Attributes are primarily useful where the value of the attribute is assigned in one logic section and evaluated in another logic section or field, perhaps at a different location. If, however, an attribute is assigned a value and evaluated within the same logic section (e.g., an operation logic), it would be more appropriate to use a local variable. Local variables act like temporary attributes and are valid only within the logic in which they are defined (see *Variables* on page 423).

8.1.4 Cloning Attributes

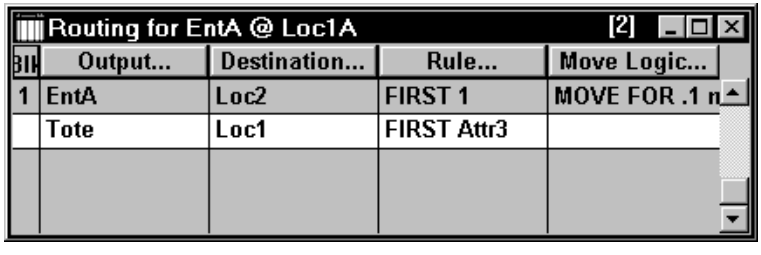
Whenever one entity initiates the creation of other entities through a SPLIT AS, CREATE, or ORDER statement, or as the result of specifying multiple outputs in a routing, the attributes of the original entity are automatically copied to each newly created entity.

The following examples show two methods of splitting an incoming batch of material, called BatchA, into one Tote and six EntA's. The first method uses a CREATE statement in the operation logic to create the new entities, called EntA. The original entity, BatchA, will have its name changed to Tote in the routing table. In the second example, both of the new entity types are created in the routing table, so no CREATE statement is needed in the operation logic. In both cases, the attributes attached to the original entity, BatchA, are duplicated in both Tote's and the EntA's attributes.

Process Logic

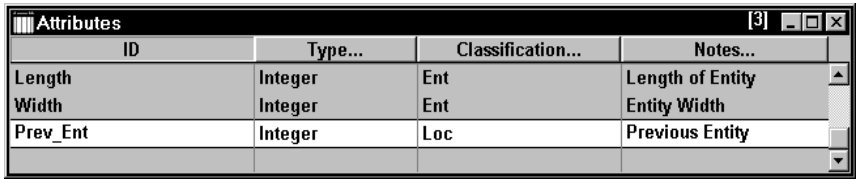


Routing Table



8.1.5 Attribute Edit Table

This edit table is used to define entity and location attributes. A description of each field is given on the next page.



ID The name of the attribute.

Type The type of the attribute, real or integer.

Classification Entity attribute or location attribute.

Notes A general notes field for describing the attribute. Notes fields are for a model's users only and are never analyzed by ProModel. Click on the Notes button or double click in the field to open an edit window for entering detailed notes.

The attribute edit table above shows that two attributes, Length and Width, have been defined for each entity in the model. In addition, an attribute called Prev_Ent has been defined for each location.

8.1.6 Example of Attributes in Logic

An appliance manufacturer's model contains an assembly location to join lids to pots. The pots are either aluminum or steel and both types of pots arrive at the same assembly location. If an aluminum pot arrives at the assembly location, it must be joined with an aluminum lid. The same is true for a steel pot and lid. The entities, steel_lids and alum_lids, are waiting at a queue to be joined to the pots.

Obviously, one way to model the different pot types is to use two different entity types. This example shows how to achieve the same effect using a single entity type (pot) with an attribute designating whether it is steel or aluminum.

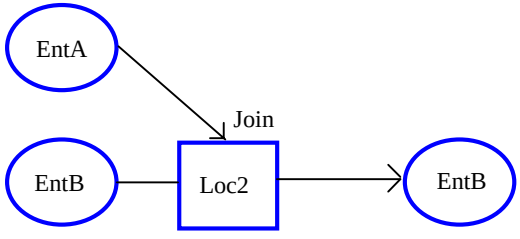
An attribute called "type," defined in the attribute edit table, allows the location to tell what type of pot has arrived at the assembly location. We will use a value of 1 to represent a steel pot and a value of 2 to represent an aluminum pot. When a steel pot enters the system, we assign a value of 1 to the attribute TYPE with the statement TYPE=1. When an aluminum pot enters the system, we set its type to 2.

At the assembly location, we use the following logic:

This logic checks the type of the pot and then joins a lid according to that type.

8.1.7 Attributes and the JOIN Statement

In some cases, one entity joins to another entity using the JOIN statement (see *Join* on page 172 of the *ProModel Reference Guide* for more information). If both entities possess attributes before they join together, the resulting joined entity will possess the attribute values of the entity joined to it. In other words, the entity with the JOIN routing rule is effectively destroyed when it gets joined. Consider the following diagram in which EntA joins to EntB:



EntA is joined to EntB.

The logic for the above diagram is as follows:

Process Table **Routing Table**

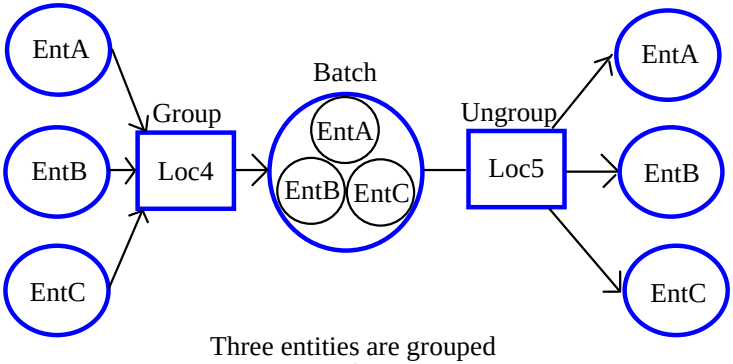
Entity	Location	Operation (min)	Blk	Output	Destination	Rule	Move Logic
EntA	Loc1	Att1 = 1 WAIT 2 min	1	EntA	Loc2	JOIN 1	MOVE FOR 1
EntB	Loc2	Att1 = 2 JOIN 1 EntA	1	EntB	Loc3	FIRST 1	MOVE FOR 1
EntB	Loc3	...	...	...	...	...	...

In the above example, EntB would have an attribute value, Att1, equal to 2 after EntA joined to EntB.

8.1.8 Attributes and the GROUP/UNGROUP Statements

Suppose several entities, EntA, EntB, and EntC, are grouped together and called Batch (see *Group* on page 161 and *Ungroup* on page 239 of the *ProModel Reference Guide* for more information). Each of the original entities have attributes with values assigned to them before they are grouped. The Batch is processed for 30 minutes, sent to Loc5 and then ungrouped into the original entities.

The attribute values of the individual entities are not transferred to the grouped entity, Batch. In other words, Att1=0 for the entity, Batch. However, once the entities are ungrouped, they retain their original attribute values. The following diagram graphically shows the concept of grouping.



Three entities are grouped together to form a batch which is later ungrouped.

The logic for the diagram is as follows:

Process Table

Entity	Location	Operation (min)
EntA	Loc1	Att1 = 1
EntB	Loc2	Att1 = 2
EntC	Loc3	Att1 = 3
ALL	Loc4	GROUP 3 AS Batch
Batch	Loc4	WAIT 30
Batch	Loc5	UNGROUP
ALL	Loc5	...

Routing Table

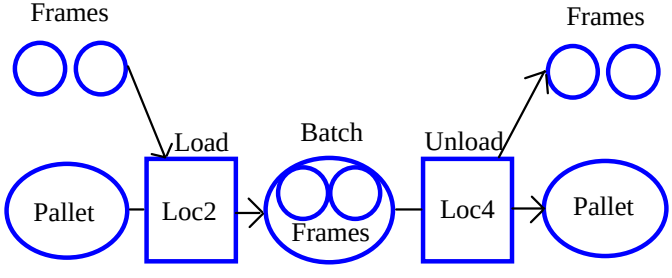
Blk	Output	Destination	Rule	Move Logic
1	EntA	Loc4	FIRST 1	MOVE FOR 1
1	EntB	Loc4	FIRST 1	MOVE FOR 1
1	EntC	Loc4	FIRST 1	MOVE FOR 1
1	Batch	Loc5	FIRST 1	MOVE FOR 5
...	...	...	...	...

Note You can assign an attribute value to a grouped entity. However, once the entities are ungrouped, they retain the attribute values they possessed before they were grouped.

8.1.9 Attributes and the LOAD/UNLOAD Statements

The LOAD statement loads a specified quantity of entities to the current entity. The loaded entities retain their identity for future unloading through an UNLOAD statement (see *Load* on page 178 and *Unload* on page 242 of the *ProModel Reference Guide* for more information). When the entities are loaded onto the current entity, the resulting entity retains the attribute value of the current entity.

For example, entities called Frames are loaded onto another entity, Pallet. The Frames are assigned an attribute value, Att1=1. Pallets are also assigned an attribute value, Att1=2. Once the Frames are loaded onto the Pallet, the loaded pallet is renamed Batch. The Batch then has an attribute, Att1=2 because it inherits the attribute value of the Pallet. However, we then assign an attribute value to Batch, Att1=3. After the Frames are unloaded from the Pallet, the Frames retain their original attribute value, Att1=1. However, the Pallet now has a different attribute value, Att1=3, which was assigned to the renamed entity, Batch. Consider the following diagram and logic in which two Frames are loaded onto a Pallet and renamed Batch for the output entity:



Two Frames are loaded onto a Pallet and renamed Batch in the Output. The Frames are then unloaded from the Batch. Batch is renamed Pallet in the Output.

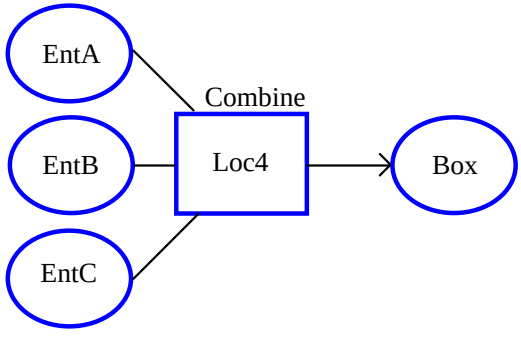
The logic for the diagram is as follows:

Process Table **Routing Table**

Entity	Location	Operation (min)	Blk	Output	Destination	Rule	Move Logic
Frame	Loc1	Att1 = 1 WAIT 2 min	1	Frame	Loc2	LOAD 1	MOVE FOR 1
Pallet	Loc2	Att1 = 2 LOAD 2	1	Batch	Loc3	FIRST 1	MOVE FOR 3
Batch	Loc3	Att1 = 3 WAIT 20	1	Batch	Loc4	FIRST 1	MOVE FOR .5
Batch	Loc4	UNLOAD 2	1	Pallet	Loc5	FIRST 1	MOVE FOR 1
Frame	Loc4	WAIT 10	1	Frame	Loc5	FIRST 1	MOVE FOR 2

8.1.10 Attributes and the COMBINE Statement

Consider the example where several entities are combined permanently into a single entity, Box (see *Combine* on page 128 of the *ProModel Reference Guide* for more information). The combined entity, Box, assumes the attribute values of the last entity that was combined to the single entity. If three entities, EntA, EntB, and EntC, are combined to form a single entity called Box, and EntC was the last entity that was combined, the Box will have the same attribute values as EntC. Therefore, if EntC had an attribute (Att1=5), then Att1=5 for the combined entity, Box. The following diagram demonstrates three entities combining into one entity.



Three entities are combined to form a single entity called Box.

The logic for the diagram is as follows:

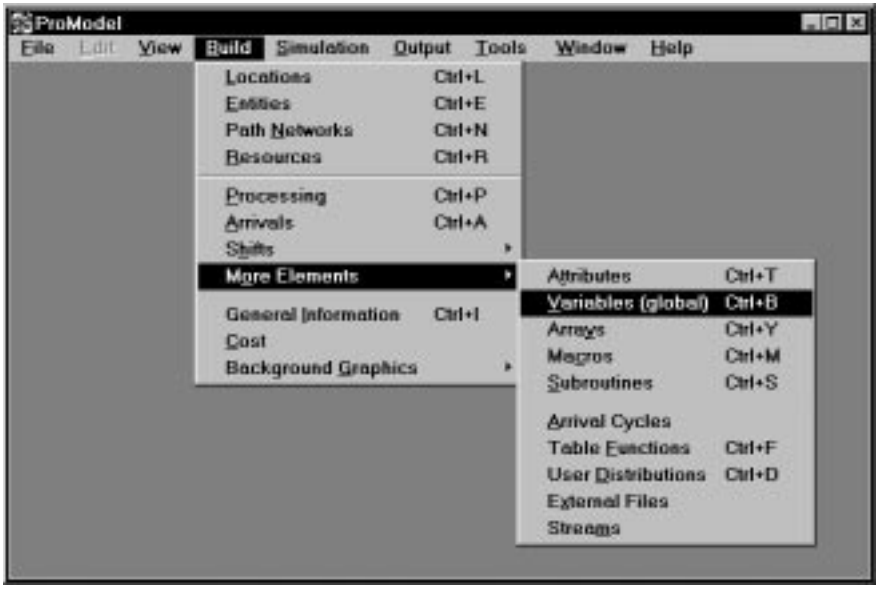
Process Table **Routing Table**

Entity	Location	Operation (min)	Blk	Output	Destination	Rule	Move Logic
EntA	Loc1	Att1 = 1 WAIT 2 min	1	EntA	Loc4	FIRST 1	MOVE FOR 1
EntB	Loc2	Att1 = 3 WAIT 3 min	1	EntB	Loc4	FIRST 1	MOVE FOR 3
EntC	Loc3	Att1 = 5 WAIT 6	1	EntC	Loc4	FIRST 1	MOVE FOR .5
ALL	Loc4	COMBINE 3	1	Box	Loc5	FIRST 1	MOVE FOR 1
ALL	Loc5	...	...	...	...	...	...

In the above example, EntC is the last entity to be combined so the entity Box assumes the attribute value of EntC, Att1=5.

8.2 Variables

Variables are of two types: global and local. Global variables are place holders defined by the user to represent changing numeric values. Local variables are place holders which are only available within the logic that declared them. Variables can contain either Real numbers or Integers, including element index values, and are typically used for decision-making or recording information. A global variable can be referenced anywhere numeric expressions are allowed in a model. If a variable or attribute is only needed in a single block of logic, it is easier to define a local variable right inside the logic block. Global variables are defined in the Variables Editor, accessed from the Build menu. Local variables are defined with the INT and REAL statements. (See *Local Variables* on page 428.)



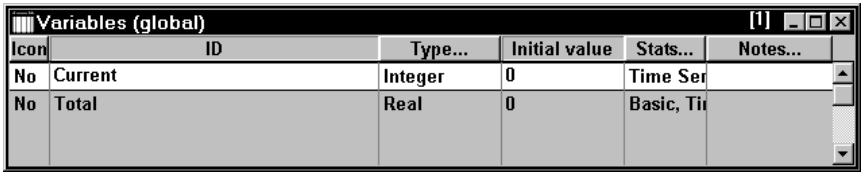
How To

Access the Variable Edit Table:

1. Select **More Elements** from the **Build** menu.
2. Select **Variables** (global) from the submenu.

8.2.1 Variable Edit Table

This edit table is used to define Variables used globally in the model. A description of each field is given below.



Icon This field shows "Yes" if an icon for the variable appears on the layout. A variable's icon looks like a counter and displays the variable's value.

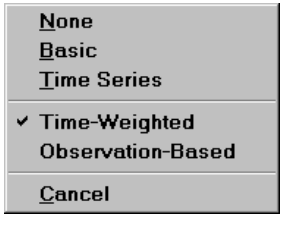
ID The variable's name.

Type The type of variable, real or integer.

Initial Value The initial value of the variable to be assigned at the start of the simulation. By default, initial values are 0, but can be changed in the edit table to whatever value the user wants. Any expression can be entered here (including previously defined variables) except attributes and system functions.

Stats ProModel collects statistics for each variable on three levels of detail, None, Basic, and Time Series.

- **None** No statistics are collected for this variable during simulation.
- **Basic** Collects basic statistics such as total changes, average minutes per change, current value, and average value.
- **Time Series** Collects all the basic statistics plus the value history based on time or observations. When you select Time Series, ProModel collects either time-weighted or observation-based statistics for the variable depending upon the type selected.



Variable Observation Record	Value	Time in Hours
Observation 1	6	1
Observation 2	5	2
Observation 3	6	3
Observation 4	5	4
Total	**22**	**10**

Time-Weighted Collects information on the percentage of time the variable was at a specific value. As shown in the above table, the average value of the variable is:

$$\frac{(6 \times 1) + (6 \times 3) + (5 \times 2) + (5 \times 4)}{10} = 5.4$$

Observation-Based Collects information on the number of times the variable changed to a specific value. As shown in the above table, the value of the variable is 6, then 5, then 6, then 5. The average would simply be:

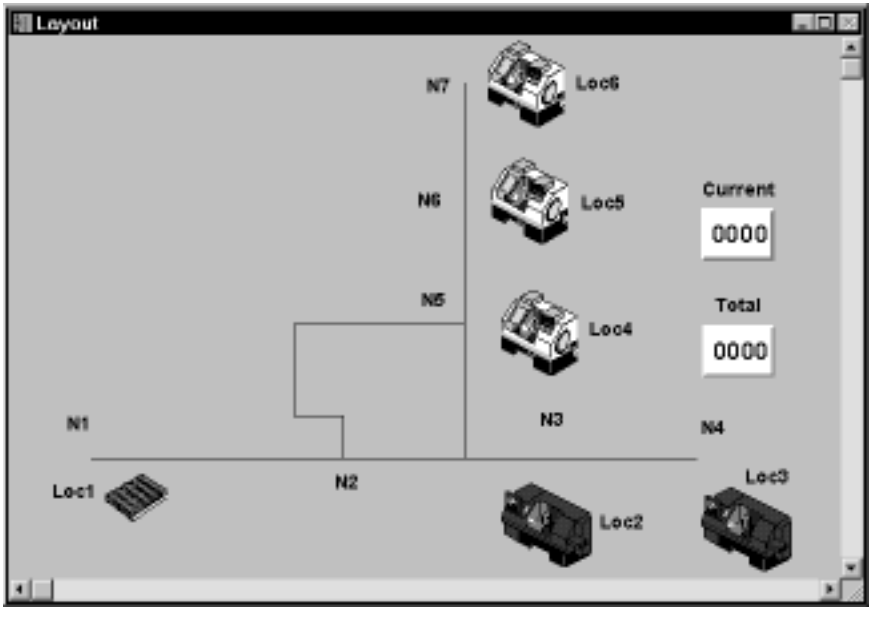
$$\frac{6 + 5 + 6 + 5}{4} = 5.5$$

Notes A general notes field for describing the variable. Click on the Notes button or double-click in the field to open an edit window for entering detailed notes.

Note In order to create plots or histograms for a variable, Time Series stats must be selected in the variables edit table.

8.2.2 Variable Layout

An icon to show a variable's value during a simulation may be placed anywhere on the layout. The window below shows the icons for the variables Current and Total at the right side of the screen. Each icon has been labeled with a background graphic.



How To Place an icon for a variable on the Layout:

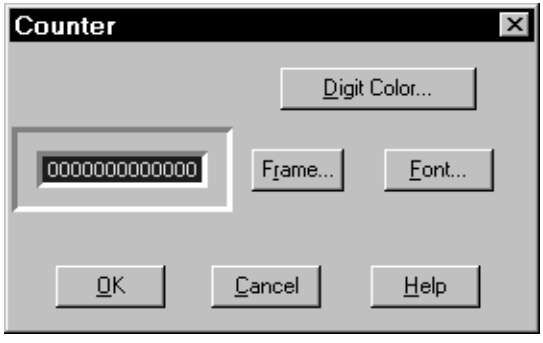
1. Highlight the desired variable in the Variable edit table.
2. Click on the layout where the icon is to appear.
3. Size the icon by dragging an edge or corner of the sizing box.

How To Remove an icon for a variable from the Layout:

1. Double click on the icon.
2. Choose **Delete** from the resulting menu. The icon is removed from the layout, but the variable remains in the model.

8.2.3 Editing a Variable's Icon

A variable's icon can be customized as necessary by simply double-clicking on the icon and choosing Edit. A dialog box appears as shown below for specifying the characteristics of the variable icon or counter.



How To **Edit a variable's icon:**

1. Double click on the icon.
2. Click on the Digit Color, Frame, or Font buttons to adjust the respective setting.
3. Click **OK**.

8.2.4 Local Variables

Local variables function as though they were temporary attributes defined in a specific logic section which disappear when the logic section is finished executing. Local variables are useful for test variables in loops and storing locally used, unique values for each entity at the current location.

Local variables are used within a block of logic (i.e., operation logic, subroutines, etc.) and are declared with an INT or REAL statement. Local variables are only available within the logic in which they are declared and are not defined in the Variables edit table. A new local variable is created for each entity to encounter an INT or REAL statement (See *Int* on page 170 and *Real* on page 207 of the *ProModel Reference Guide*). A local variable is specific to each entity, in much the same way an attribute is specific to an entity, except that the local variable is only available while the entity processes the logic to declare the local variable. Local variables may be passed to subroutines as parameters. Local variables are available to macros.

Example

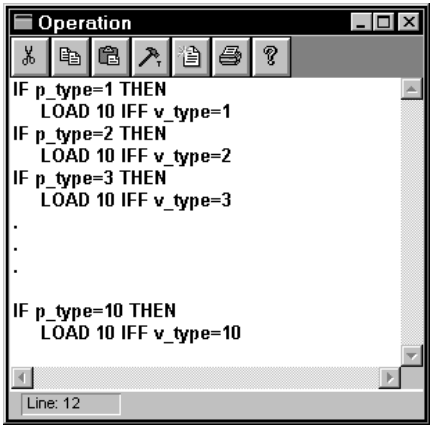
A plant manufactures valves of 10 different sizes, such as 10", 20", and so on. All valves are inspected at a common inspection station and then move to a dock where they are loaded onto pallets. The pallets are designed to hold only a certain size valve. Therefore, a pallet designed to hold 10" valves can hold only 10" valves, not 20' valves.

Suppose a Pallet enters a multi-capacity location, Dock. Each Pallet has a different entity attribute, p_type, describing the type of valve it can hold. Valves are loaded onto the Pallet. The 10" valves must be loaded onto the pallet designed to hold the 10” valves. Therefore, the attribute value of the Valve, v_type, must match the attribute value of the Pallet, p_type. We can use local variables to accomplish this modeling task. The logic is as follows where X is a local variable:

Process Table **Routing Table**

Entity	Location	Operation (min)	Blk	Output	Destination	Rule	Move Logic
Valve	Inspect	WAIT 5	1	Valve	Dock	LOAD 1	MOVE FOR 2
Pallet	Dock	INT X X = p_type LOAD 10 IFF X = v_type WAIT 10	1	Pallet	Delivery	FIRST 1	MOVE FOR 8

If we had not used local variables, we would need to use the following operation logic for Pallet at Dock:



As can be seen from the two examples of logic, the first example is much easier and more straight forward.

It is important to note that using "LOAD 10 IFF p_type = v_type" in the operation logic would not work for the intended purpose. Attributes referenced in IFF conditions always refer to the entity being loaded. Set the value of a local variable, X, to the pallet attribute, p_type, so it can be referenced in the LOAD statement. The pallet attribute cannot be directly referenced in the LOAD statement.

If Dock was a single capacity location, using a global variable would work the same as using a local variable. However, because Dock is a multi-capacity location, it can load valves onto multiple pallets at the same time. If a global variable was used instead of a local variable, the global variable would change each time a pallet entered Dock. If there were two different types of pallets at Dock, there would be only one type of valve loaded on the pallet because the global variable refers to both pallets.

Suppose, for example, a global variable, type, signifies the pallet attribute, p_type. We assign type=p_type at the beginning of the operation logic for location Dock. The first pallet arrives and type=3. Therefore, only valves with v_type=3 are loaded onto the pallet. Another pallet enters Dock and type=5. Now only valves with v_type=5 are loaded onto both pallets.

Note

Local variable notes:

1. You may not use the WAIT UNTIL statement with local variables.
2. The local variable definition only needs to appear somewhere in the logic before being referenced. The entity does not need to execute the local variable definition statement (INT, REAL).

8.3 Arrays

An array is a matrix of cells that contain real or integer values. Each cell in an array works much like a variable, and a reference to a cell in an array can be used anywhere a variable can be used. A one-dimensional array may be thought of as a single column of values. A two-dimensional array is like having multiple columns of values (similar to a spreadsheet).

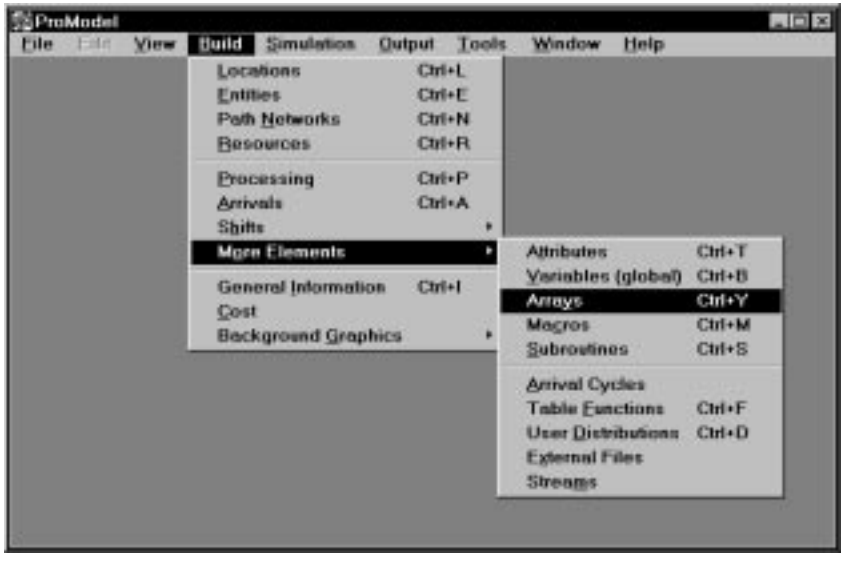
To reference a cell in a one-dimensional array, give the name of the array and enclose the cell number in brackets. For example, if a one-dimensional array containing ten cells is named OpnArray, to reference the fifth cell you would use OpnArray[5]. To reference a cell in a two-dimensional array, simply use the name of the array with the row and column number in brackets. For example, the statement OpnArray[3,4] references a cell on the third row in the fourth column.

Arrays with more than two dimensions are more difficult to picture, but work exactly the same as one and two dimensional arrays. For example, if an array has a third dimension, you can reference any cell simply by adding a comma before the number of the desired cell in the third dimension. For example, Tool[3,5,8]. Four and five-dimensional arrays work the same way.

Array cell values are assigned in the exact same way as you would assign a value to a variable. For example, to assign the value 18 to the cell at the fifth row and second column in an array named Arr1 you would use the statement:

Arr1[5,2]=18

Arrays are defined in the Arrays editor which is accessed through the Build menu.



How To **Use the Arrays Editor:**

1. Select **More Elements** from the **Build** menu.
2. Select **Arrays**.

Example Arrays

The following examples show how elements are referenced in a one-dimensional array with five cells and in a two-dimensional array with fifteen cells.

One-dimensional array

OpnArray[5]

Cell[1]
Cell[2]
Cell[3]
Cell[4]
Cell[5]

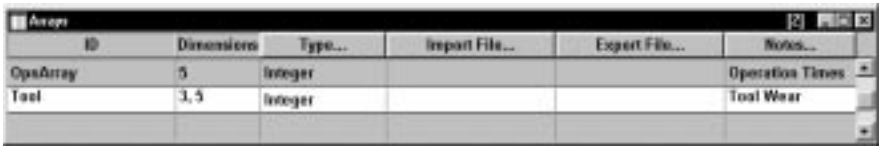
Two-dimensional array

Tool[3,5]

Cell[1,1]	Cell[1,2]	Cell[1,3]	Cell[1,4]	Cell[1,5]
Cell[2,1]	Cell[2,2]	Cell[2,3]	Cell[2,4]	Cell[2,5]
Cell[3,1]	Cell[3,2]	Cell[3,3]	Cell[3,4]	Cell[3,5]

8.3.1 Arrays Edit Table

The Arrays edit table is used to define Arrays that are used in the model. The fields of the Arrays edit table are explained on the following page.



ID The name of the array.

Dimensions The size of each dimension of the array in cells. For example, the dimensions of a one-dimensional array of 100 cells is "100." Likewise, a two-dimensional array with 50 rows and five columns would have dimensions of "50,5." The number of rows is first, followed by a comma and then the number of columns.

Type The numeric type (real or integer) for all cells in the array.

Import File The spreadsheet from which you will populate the array.

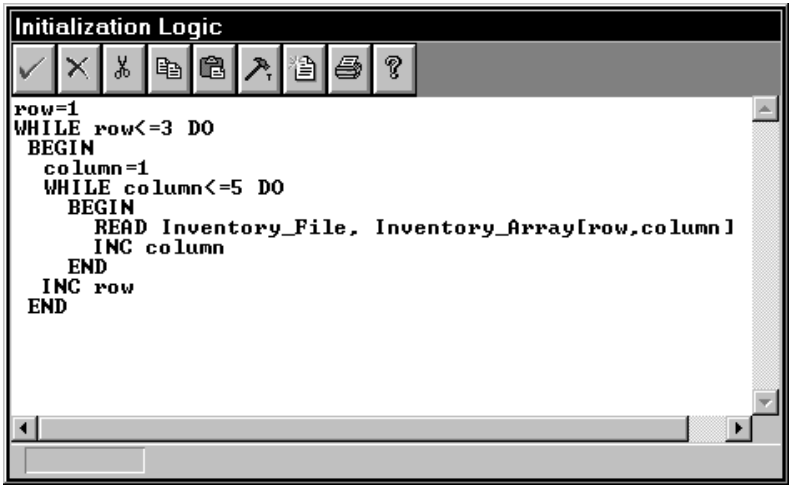
Export File The spreadsheet to which you will save the array data.

Notes A general notes field for entering descriptive information about the array. Click the heading button or double click in this field to open a larger window for entering notes.

The window pictured above shows how to define the example arrays that appear on the previous page.

8.3.2 Initializing Arrays

By default, all cells in an array are initialized to zero. Initializing cells to some other value should be done in the initialization logic. A WHILE...DO loop is useful for initializing array cell values. The logic below fills a 3 x 5 array (3 rows and 5 columns) with values from an external general read file.

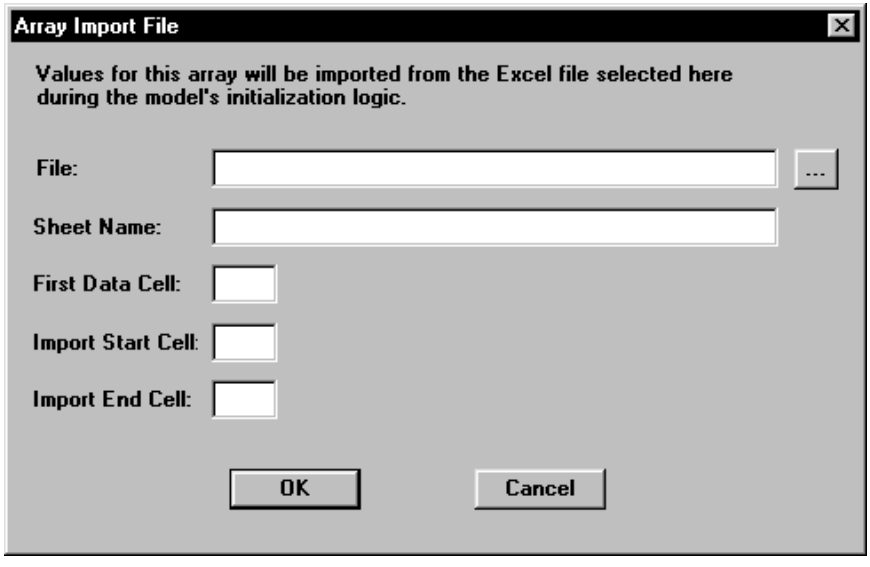


This example uses the variables Column and Row (which may be defined as local variables) along with two WHILE...DO loops to assign every cell in the array Inventory_Array a value from a general inventory file. The logic first sets the value of the variable Column to one. It then assigns all the cells in column one a value by reading a value from the external file and incrementing the variable Row. When all the cells in column one have a value, the logic increments to the second column and does the inside loop again. It repeats this loop until each cell in the array has a value.

Note Assigning values to a cell in an array can be done in any expression or logic field, such as initialization and operation logic. However, arrays cannot be used in logic elements that determine a model's structure, such as location capacity. See *Execution Time of Initialization and Termination Logic* on page 344 for a list of logic elements used to define model parameters.

8.3.3 Import Spreadsheet Data into Arrays

When you import data from an external Excel spreadsheet into an array, ProModel loads the data from left to right, top to bottom. Although there is no limit to the quantity of values you may use, ProModel supports only two-dimensional arrays.



File The name of the spreadsheet you will use.

Sheet Name The name of the sheet from which you will import the array data.

Note If your spreadsheet contains only a single data set, ProModel will automatically load the data into the array—you do not have to define any cell information unless you wish to limit the contents of the array to a portion of the data set.

First Data Cell *(For complex spreadsheets with multiple data sets only.)* The intersection of the first row and column in the spreadsheet (at the uppermost left) to contain either a mathematical expression or numeric data. ProModel will use this cell as a reference point from which to isolate the data in your spreadsheet.

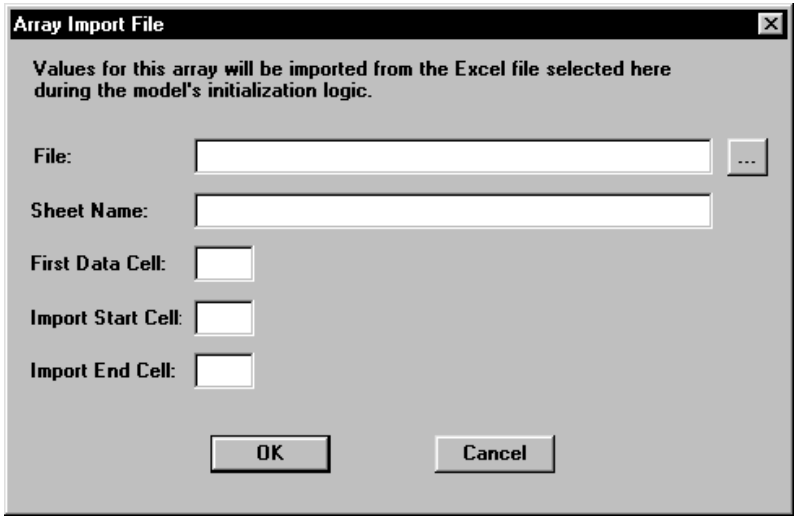
Import Start Cell The *first* piece of data to place into the array.

Import End Cell The *last* piece of data to place into the array.

How To

Import Data into Array:

1. Select **Arrays** from the **More Elements** section of the **Build** menu.
2. Click the **Import File** button on the Arrays dialog.
3. In the **File** field, enter the name of the spreadsheet you wish to use (you may also browse to select a file).

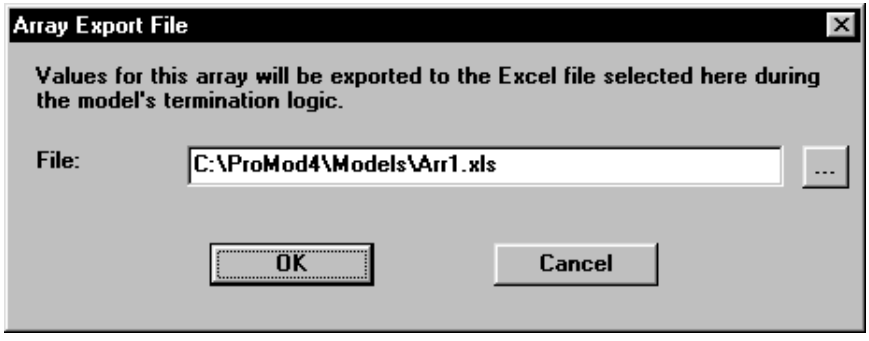


4. In the **Sheet Name** field, enter the name of the sheet that contains the data you wish to use.
5. Click **OK**.

For complex spreadsheets only:

6. In the **First Data Cell** field, enter the intersection of the first row and column in the spreadsheet (at the uppermost left) to contain either a mathematical expression or numeric data.
7. Enter the **Import Start Cell** location. The value in this cell will occupy the *first* position in the array.
8. Enter the **Import End Cell** location. The value in this cell will occupy the *last* position in the array.
9. Click **OK**.

8.3.4 Export Arrays to Spreadsheets

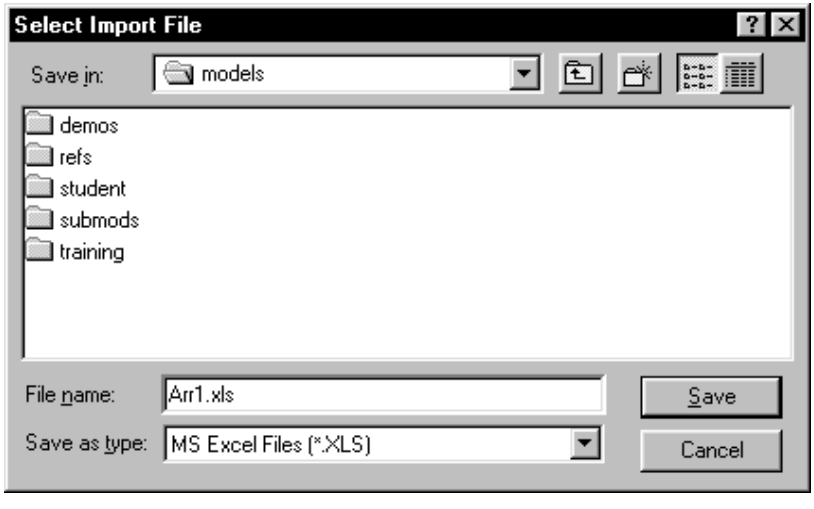


File The name of the spreadsheet you will use. If you export multiple times to the same spreadsheet, ProModel will overwrite the spreadsheet with new data.

How To

Export Array Data:

1. Select **Arrays** from the **More Elements** section of the **Build** menu.
2. Click the **Export File** button on the Arrays dialog.
3. Enter the name of the spreadsheet file you wish to use, or browse to select a file.



4. Click **OK**.

8.3.5 Using Arrays

Using arrays can simplify a model. Suppose you need to model an assembly line that attaches components to a computer motherboard. Furthermore, you want to track the usage of component parts over time. Without an array, hundreds of individual entities of various types would have to represent hundreds of individual components. Keeping track of all the components would be very complex, not to mention all of the join operations and routings for performing the assembly. Instead, various cells in a one-dimensional array could track the number of each type of component used during the simulation.

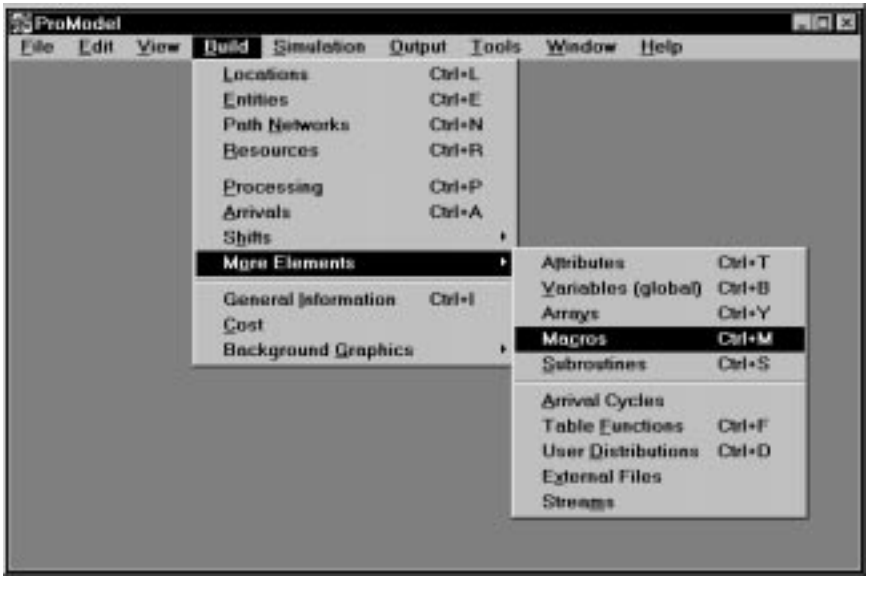
An array can do the job more efficiently. The initial inventory level for each component could be stored in an external file and read into the cells of the array at the start of the simulation. The first cell might contain the inventory level of transistors; the second could contain the inventory level of capacitors and so on. When a motherboard arrives at the location adding the components, each cell's value is decremented according to the number of that type of component joined to the motherboard. If each motherboard requires twelve transistors and five capacitors, then every time a motherboard arrives at the location, the array's first cell is reduced by twelve and the second cell is reduced by five. Thus the model becomes much less complex because it requires fewer entities and less logic.

8.3.6 Notes on Arrays

1. If a warm-up time is specified, array values are not reset.
2. Arrays can only hold numeric values, including name indexes. Character strings are not allowed.
3. Arrays can be nested. For example, if Arr1[2,3] is equal to three, then the statement Arr2[5,Arr1[2,3]] works exactly like the statement Arr2[5,3].
4. You can examine the value of a cell in an array during a simulation by choosing Information and then Arrays. This information can also be printed.
5. Arrays can be used with the WAIT UNTIL statement.
6. Statistics are not generated for arrays. However, if you would like to see the final value of an array's cell, you can place a PAUSE statement in the termination logic and then view the array under the Information menu. You could also print an array's values or write them to an external file as part of the termination logic. If you want more statistical information on a particular cell, assign the cell to a variable and then choose basic or time-series statistics for the variable.

8.4 Macros

A macro is a place holder for an often used expression, set of statements and functions, or any text that might be used in an expression or logic field. A macro can be typed once, and then the macro's name can be substituted for the text it represents anywhere in the model and as many times as necessary. Macros are defined in the Macros Editor, accessed from the Build menu.

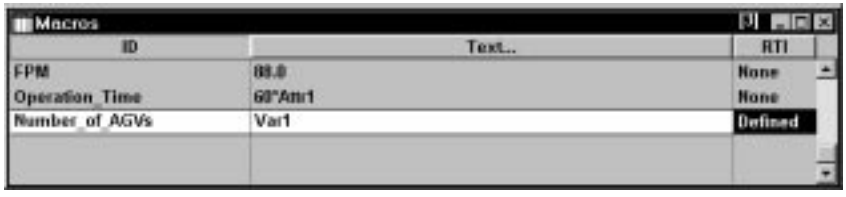


How To **Create and edit macros:**

1. Select **More Elements** from the **Build** menu.
2. Select **Macros**.

8.4.1 Macro Editor

The Macro edit table is used to assign recurring text to a reference name. The fields of the table are defined on the following page.



ID A name to identify the macro.

Text Any text to be substituted where the macro name is called. This text may be a complete expression, an entire logic block, or even part of a logic block.

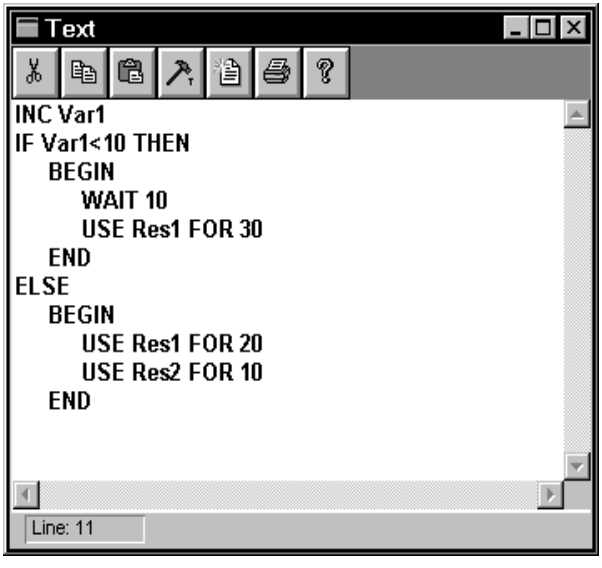
RTI Option to define the macro as a run-time interface parameter.

The example table above defines three macros. The first macro is simply a numeric constant, with fpm representing a conversion factor from miles per hour to feet per minute. If a number is used in multiple places in a model, then a macro makes it possible to change that number throughout the model simply by changing the macro itself. The second macro, Operation_Time, calculates the various operation times at different locations depending on the attributes at the locations. The last macro, Number_of_AGVs, is a run-time interface variable used to define the number of AGVs in the simulation model (see *Run-Time Interface* on page 443).

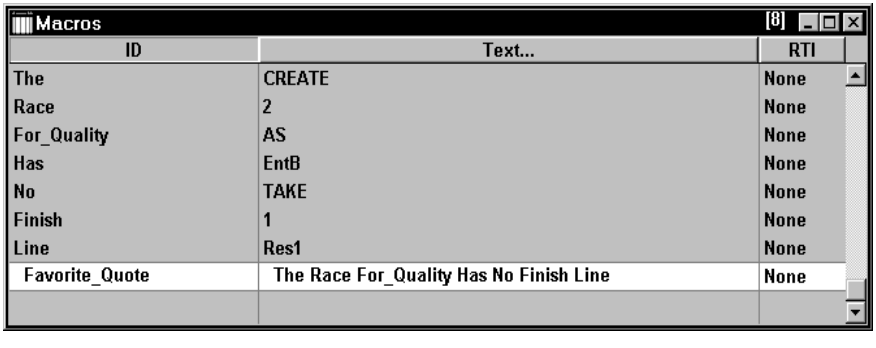
A macro is different from a subroutine because a macro cannot pass or return a value. However, because it is simply a text replacement, a macro can reference any expression that is valid in the expression or logic field that called the macro. For instance, the string “the number of entries is” might be a macro called mac1. This macro by itself is not a valid expression. However, when used with the DISPLAY statement in the operation logic (i.e., DISPLAY mac1), the compiler will recognize the macro as a string.

A macro may be used in any expression field, but may only contain a numeric expression (e.g., Entries (LOC1), U(5,1), Var1+Att2, etc.). In addition, a macro used in an expression field may not contain multiple lines of text. When used in a logic field, the macro may include any logic element valid in that logic field.

Suppose five different locations use the same lines of code. Instead of entering the same logic five times in five different fields, reference the following macro by typing the macro ID, Mac1, in the operation logic of the machine:



Every time the macro is referenced, the logic is executed. Macros can also be nested within other macros. This means that a macro can consist of one or more other macros. Consider the following Macro edit table:



ID	Text...	RTI
The	CREATE	None
Race	2	None
For_Quality	AS	None
Has	EntB	None
No	TAKE	None
Finish	1	None
Line	Res1	None
Favorite_Quote	The Race For_Quality Has No Finish Line	None

The macro, Favorite_Quote references other macros, such as Race and Finish. Note that some of the other macros, such as For_Quality, are only portions of a complete line of code. Although the macro is valid, it will not compile as a part of macro logic because the create statement requires an expression and an entity name. The line Favorite_Quote in a logic field would be interpreted as the following line, Create 2 As EntB Take 1 Res1.

Note

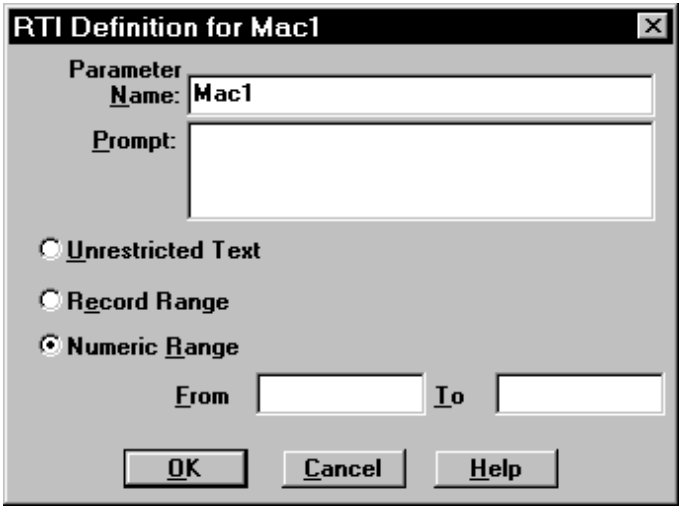
Macro notes:

1. A macro may be used only when the elements contained in the macro are appropriate to the context from which it was called. This restriction means that the macro in the previous example is only valid in operation logic.
2. Because a macro simply substitutes some text for its name, if a macro represents a statement block, then it should contain a BEGIN at the beginning of the block, and an END at the end of the block. This technique is especially important when using a macro immediately after a control statement, such as IF...THEN or WHILE...DO. For more information, see *Statement Blocks* on page 83 of the *ProModel Reference Guide*.

8.4.2 Run-Time Interface

Defining a run-time interface (RTI) for a macro allows the user to easily change simulation/model parameters before the simulation starts. It also provides an experimental framework for defining multiple scenarios to run in a batch (see *Model Parameters & Scenarios* on page 575 for more information). An RTI for a macro allows a macro's text to be changed by the user whenever a simulation run begins. Since macros are allowed in any expression, this gives the user flexibility to edit most model parameters every time a simulation starts without having to directly edit the model data.

The key difference between a macro with an RTI and one without is that when a simulation begins, a macro defined with an RTI provides a menu that allows users to change only the macros you want them to change. An RTI allows you to request a variety of information to substitute for a macro; from simple values (e.g., the initial value of a location's capacity) to complex text (e.g., a line of logic). You may create RTI parameters using the dialog box below, accessed through the macros dialog.



Parameter Name This text will identify the parameter represented by this macro. It should consist of text that clearly describes the parameter to be changed, for example, Operation Time. The macro name and the parameter name can be different. This provides more flexibility and allows the user to view a more descriptive parameter name when defining scenarios.

Prompt This text will appear if the user decides to change the parameter. You should use it to further specify the information to be entered, for example, "Please enter the amount of time that the simulation should run."

Unrestricted Text This option allows the user to enter any text, such as the distribution U(8,2). Note that any text that the user enters will be substituted for the macro

name in the model. Therefore, the text the user enters must be syntactically correct and valid anywhere the macro name appears.

Record Range Allows you to enable an arrival or shift record from a range of records. This allows you to test a variety of shift and arrival combinations to find the combination that works best with your model.

Numeric Range This specifies the lower and upper limits for the parameter if the type is numeric.

How To

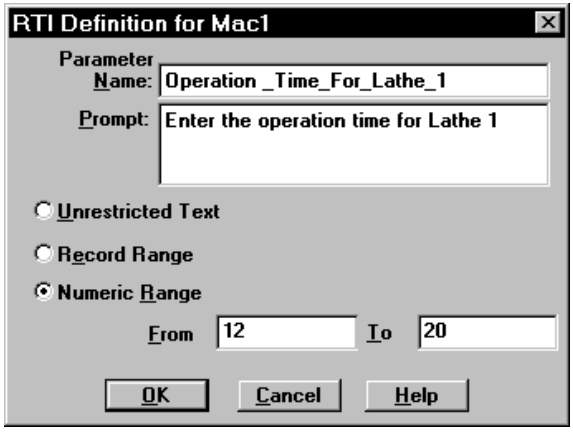
Define an RTI for a macro:

1. Select **Macros** from the **Build** menu.
2. Type the macro name and click on the **Options** button.
3. Choose **Define** from the **RTI** submenu.
4. Define the Parameter Name and enter the prompt (optional).
5. Select the parameter type: **Unrestricted Text**, **Record Range**, or **Numeric Range**.
6. If the parameter is numeric, enter the lower value in the From box and the upper value in the To box.
7. Click **OK**.
8. Enter the default text or numeric value in the Macro Text field.
9. Use the macro ID in the model (e.g., in operation or resource usage time).

Note When using a record range, be sure to group all arrival and shift records. This will allow you to select which series of records to include in the macro. Note also that when you define an arrival or shift RTI, ProModel adds "ARRIVAL_" or "SHIFT_" to the name to help you identify the macro more easily.

Run-Time Interface Example

Suppose you build a factory model and determine the first lathe, Lathe_1, is a bottleneck. The model results indicate the throughput is lower than expected. You decide to perform several what-if scenarios with the model by changing the operation time of Lathe_1. Instead of changing the operation time at Lathe_1 within the Process edit table, it is easier to define a macro with an RTI. This technique allows the model user to easily see the effect of installing a faster lathe without ever editing the model itself. The following example represents the dialog used to define the RTI for the macro where the operation time is a numeric value between 12 and 20:



After defining the RTI for the macro, substitute the macro, Oper_Time, for the operation time in the operation logic in the Process edit table for Lathe_1 as shown below:

You are now able to change the operation time at Lathe_1 using the Model Parameters option in the Simulation menu. For more information see *Model Parameters & Scenarios* on page 575.

Note For more information concerning the differences between macros and subroutines, see *Subroutines* on page 447.

8.4.3 Resource Grouping

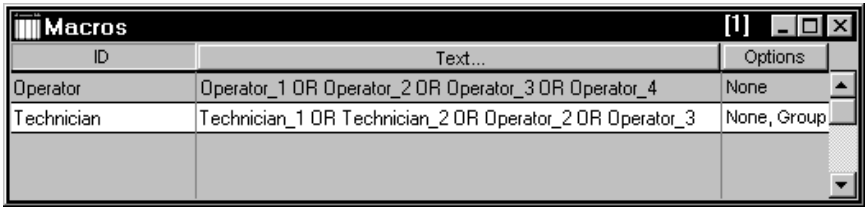
Resource grouping allows you to define specific groups of resources rather than define each unit separately. For example, suppose you need a specific technician to perform an operation. If the technician is not available, you may use either another technician or one of two qualified operators to perform the operation. Rather than define each qualified operator as a separate resource, you may define a macro that includes them.

How To Define a resource group

1. Select **Macros** from the more elements section of **Build** menu.
2. Define a macro ID and enter a list of all resources you wish to include in as part of the resource group.

Note When you create a list of resources, separate each resource using AND or OR (e.g., Tech_1 AND Tech_2 OR Tech_3 AND Tech_4).

3. Click on the **Options** button and select **Resource Group** from the submenu.



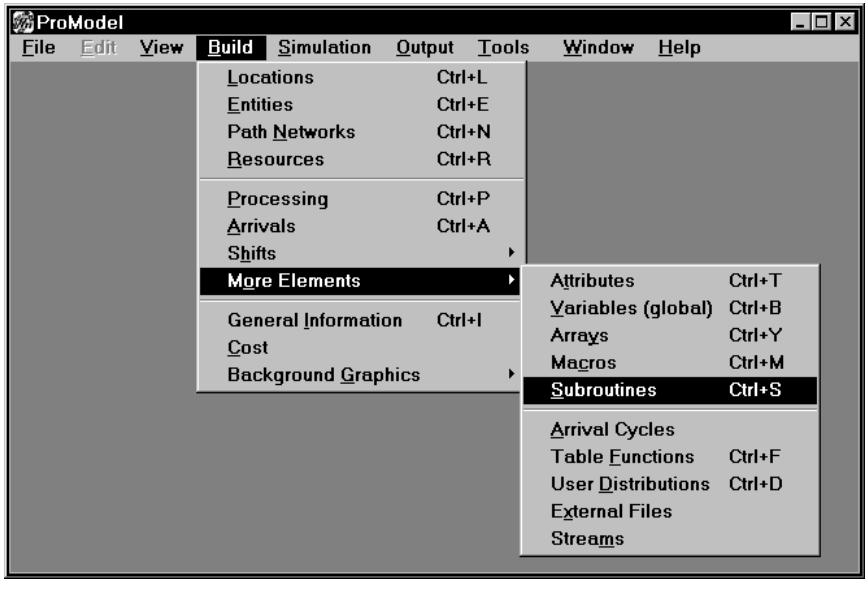
At the end of the simulation, ProModel creates a statistical report containing information collected for each resource included in the resource group, as well as the entire group. This will allow you to track individual, as well as group performance. For information on statistics and how to graph the results, see *Creating Reports* on page 631.

8.5 Subroutines

A subroutine is a user-defined command that can be called to perform a block of logic and optionally return a value. Subroutines may have parameters (local variables) which act as variables local to the subroutine and that take on the values of arguments (i.e., numeric expressions) passed to the subroutine.

ProModel handles subroutines in three ways. First, a subroutine may be processed by the calling logic as though the subroutine is part of the calling logic. This way is the most commonly used, and is done by simply referencing the subroutine by name in some logic or expression. Second, a subroutine may be processed independently of the calling logic so that the calling logic continues without waiting for the subroutine to finish. This method requires an ACTIVATE statement followed by the name of the subroutine (see *Activate* on page 115 of the *ProModel Reference Guide*), or you may use the Run-time Interact Menu (see *Run-Time Interact Menu* on page 612). Third, ProModel allows subroutines written in an external programming language to be called through the XSUB() function.

Subroutines are defined in the Subroutines Editor which is accessed from the Build Menu.

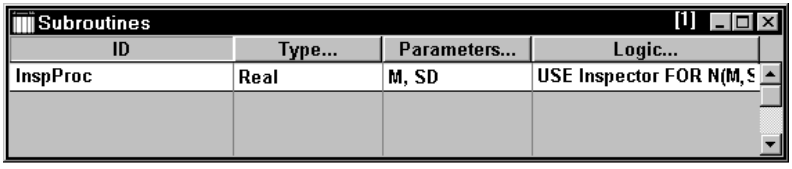


How To **Create and edit subroutines:**

1. Select **More Elements** from the **Build** menu.
2. Select **Subroutines**.

8.5.1 Subroutine Editor

The Subroutines edit table consists of several fields which identify the components of a subroutine. Each of these fields is described below.



ID A name that identifies the subroutine.

Type The type of numeric value that is returned by the subroutine, which can be Real, Integer, None, or Interactive. Use Real and Integer if the subroutine returns a number. Use None when no return value is expected, as is often the case in initialization or termination logic. Subroutines of type Interactive are identical to subroutines of type None, except that interactive subroutines are also accessible for activation by the user through the run-time menu. Interactive subroutines are displayed in the Interact menu during run time. For more information on Interactive subroutines, see the discussion later in this section.

Parameters Arguments passed to the subroutine get assigned to local variables called parameters. Items passed to a subroutine as arguments can have different names than the parameters that receive them. Parameters can be real or integer. The first parameter receives the first argument, the second parameter receives the second argument, and so on.

Logic One or more statements to be executed whenever the subroutine is called. Statements in subroutines must be valid in the logic that calls the subroutine. Subroutine logic may contain a RETURN statement with a value to be returned from the subroutine. (See *Return* on page 218 of the *ProModel Reference Guide*.)

Note

Subroutine editor notes:

1. If the subroutine is of type Integer and the return value is a real number, the return value will be truncated unless the ROUND() function is used (e.g., RETURN ROUND(<numeric expression>)).
2. If you do not want a stand-alone subroutine referenced in operation logic to be treated as an implicit wait statement, define the subroutine as type None.
3. When using the ACTIVATE statement to call a subroutine, the calling logic continues without waiting for the called subroutine to finish. Therefore, independent subroutines can run in parallel with the logic that called them.
4. Independent subroutines called with the ACTIVATE statement cannot contain entity-specific or location-specific system functions.

8.5.2 Subroutine Format

A subroutine may be named any unique, valid name. The general format for calling a subroutine is as follows:

```
SubroutineName(arg1, arg2, ....,argn)
```

Examples:

```
GetOpTime(3,7)
DoInitialization()
```

Note

Subroutine format notes:

1. If no arguments are specified, open and closed parentheses are still required.
2. Statements in subroutines must be valid in the logic that called the subroutine. For example, if a subroutine is called from the operation logic, the subroutine may contain only those statements which are valid in the operation logic. Subroutines called from an ACTIVATE statement or from the Interact Menu at runtime can have any general logic statements, including WAIT.
3. A subroutine may be used in any logic field. In addition, a subroutine may be used in any expression field, provided that the RETURN statement is used to return a value to the expression field. Expression fields include the Qty Each column of the Arrivals edit table and the routing rule for processing.
4. If a subroutine does not return an expression with the RETURN statement, a value of zero will be returned for subroutines of type Real and Integer. No value will be returned for a subroutine of type None or Interactive.

8.5.3 Subroutine Example

Gears are routed from a fabrication center to one of four manual inspection stations. The operation logic for the gears is identical at each station except for the processing times which are a function of the individual inspectors. Each gear is assigned two attributes during the fabrication process. The first attribute, OuterDi, is the dimension of the outer diameter of the gear. The second attribute, Thickness, is the dimension of the thickness of the gear. These dimensions will be tested at the inspection stations and the values entered into a database for quality tracking. After inspection, gears are routed to a shipping location if they pass inspection, or to a scrap location if they fail inspection. In addition, a fixture is removed from each gear and returned to the fixture queue.

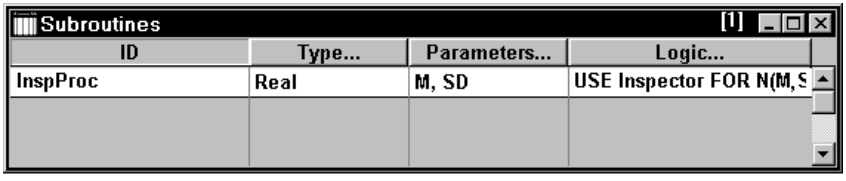
Because the operation logic at each inspection station is identical except for the processing time, a subroutine can perform the operations, saving typing and ensuring that the processing logic is identical at all locations. The processing time at each inspection station is a normal distribution with the mean and standard deviation passed as parameters to the subroutine. The operation logic, which calls the subroutine, and routing are shown below. The subroutine itself is shown later in this section.

Process Table **Routing Table**

Entity	Location	Operation (min)	Blk	Output	Destination	Rule	Move Logic
Gear	Fab	WAIT U(3.2,.1)	1	Gear	Inspect1	FIRST 1	
				Gear	Inspect2	FIRST	
				Gear	Inspect3	FIRST	
				Gear	Inspect4	FIRST	
Gear	Inspect1	InspProc(4,.3)	1	Gear	Shipping	FIRST 1	
				Fixt	FixQue	DEP	
			2	Gear	Scrap	FIRST 1	
				Fixt	FixQue	DEP	
Gear	Inspect2	InspProc(4,.2)	1	Gear	Shipping	FIRST 1	
				Fixt	FixQue	DEP	
			2	Gear	Scrap	FIRST 1	
				Fixt	FixQue	DEP	
Gear	Inspect3	InspProc(5,.1)	1	Gear	Shipping	FIRST 1	
				Fixt	FixQue	DEP	
			2	Gear	Scrap	FIRST 1	
				Fixt	FixQue	DEP	
Gear	Inspect4	InspProc(5,.2)	1	Gear	Shipping	FIRST 1	
				Fixt	FixQue	DEP	
			2	Gear	Scrap	FIRST 1	
				Fixt	FixQue	DEP	

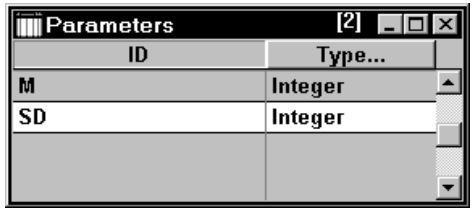
Subroutines Edit Table

The Subroutine edit table lists the name of the subroutine, the return type, the parameters to be passed to the subroutine, and the logic. The following subroutine uses an inspector for a period of time. In this case, we define the subroutine type as None since no value will be returned from the subroutine.



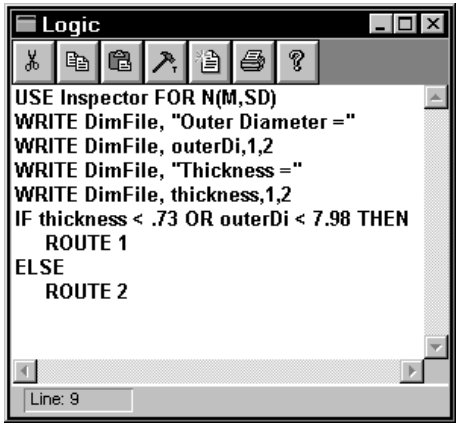
Subroutine Parameters

The subroutine parameters, M (for mean) and SD (for standard deviation), are defined by clicking on the Parameters heading button. These values are unique to each inspection location, and are passed to the subroutine as parameters of the normally distributed inspection time.



Subroutine Logic

The final step in defining the subroutine is to specify the logic that will be processed when the subroutine is called. In this example, the logic should include a processing time, a procedure to write the values of the attributes to a file, and a routing decision based on the values of the two attributes. The logic window is accessed by clicking on the Logic heading button.



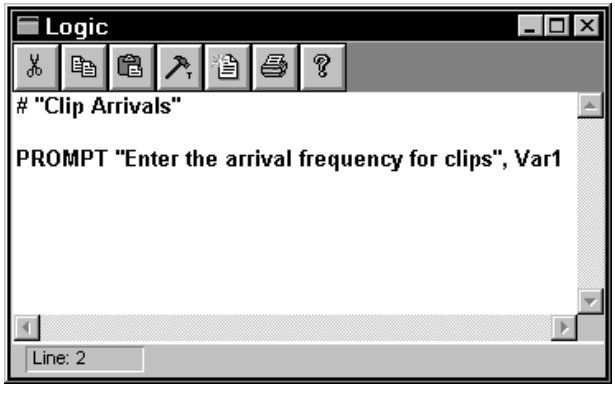
The first line of the logic uses a resource called Inspector for a normally distributed amount of time with mean and standard deviation specified by the subroutine parameters. The next four lines write the values of the dimension attributes to a General Write File called DimFile. Finally, an IF...THEN statement is used to test the values of the dimension attributes. If the Thickness attribute is less than .73 or the OuterDi attribute is less than 7.98, the gear is scrapped. Otherwise, the gear moves on to the shipping location. In either case, a fixture is removed from the gear and returned to the fixture queue (refer to the process/routing table at the start of this example).

8.5.4 Interactive Subroutines

Interactive subroutines are subroutines activated by the user anytime during run time by choosing the subroutine from the Interact menu. The name appearing in the Interact menu is either the subroutine name or if a string is entered as a comment statement at the beginning of the subroutine logic, the string is used as the name (e.g., # “Arrival Frequency”). Interactive subroutines allow the user to interact with the simulation during runtime. Subroutines are defined as type Interactive in the Subroutine edit table. Normally, subroutines are activated by entities. However, interactive subroutines can be user-activated in addition to being entity-activated. Interactive subroutines can be used for:

- changing model parameters during a simulation run
- changing routings
- calculating and reporting user-defined statistics

Suppose you want to interactively change the arrival frequency for a certain entity, clips, during runtime. Define a variable, Var1, and assign it an initial value to be used for the initial arrival frequency. Enter Var1 in the arrival frequency field for the entity clips. Create a subroutine of type Interactive and enter the following logic:



During run-time, you can then change the arrival frequency for the clips by choosing the Clip Arrivals from the Interact menu (see *Run-Time Interact Menu* on page 612 for more information).

Note Interactive subroutines may also be called from any logic or expression where no return value is required. See *Statements & Functions* on page 111 of the *ProModel Reference Guide* for more information.

8.5.5 External Subroutines

There may be some cases where you need to perform actions ProModel is not capable of doing. You may need extended capabilities with more sophisticated commands. ProModel allows you to interface with external subroutines located in thirty-two bit Windows DLL files you have created. This feature could be useful for doing sophisticated file I/O, performing statistical analysis, making your simulation interactive, or helping with other simulation needs.

Because of the intricacies of the Windows development environment, you must have a sound knowledge of your external programming language (C, C++, Pascal, etc.) to use external subroutines. In addition, you must also have a good Windows platform knowledge, specifically with respect to creating DLLs in your language. Because it is a 32-bit program, ProModel can load and call only 32-bit DLLs, and requires that you use a 32-bit Windows compiler.

For more information about this feature, you can load, study and run XSUB.MOD in the reference model directory (also see *Xsub()* on page 255 of the *ProModel Reference Guide*). This model uses XSUB.DLL, found in the MODELS directory. The source code and make files for XSUB.DLL (XSUB.CPP, XSUB.MAK, XSUB.IDE) are also included in the MODELS directory. Some general explanation is contained in the comments of this source code.

Due to the complexities of Windows programming and the variety of uses for this advanced feature, PROMODEL Corporation can provide only minimal support for this feature. Many questions regarding Windows programming and other programming languages cannot be handled through our customer support department. Please consult your language programming manuals, language customer service centers, Microsoft, and other resources to resolve these types of problems.

8.5.6 Subroutines vs. Macros

Although subroutines and macros work similarly, they have subtle differences. Any logic may use both macros and subroutines. The main difference is in the way they are used. Only subroutines can be used when you need to pass arguments, get a return value, or activate the independent execution of logic. Only macros can be used when defining run-time interface parameters.

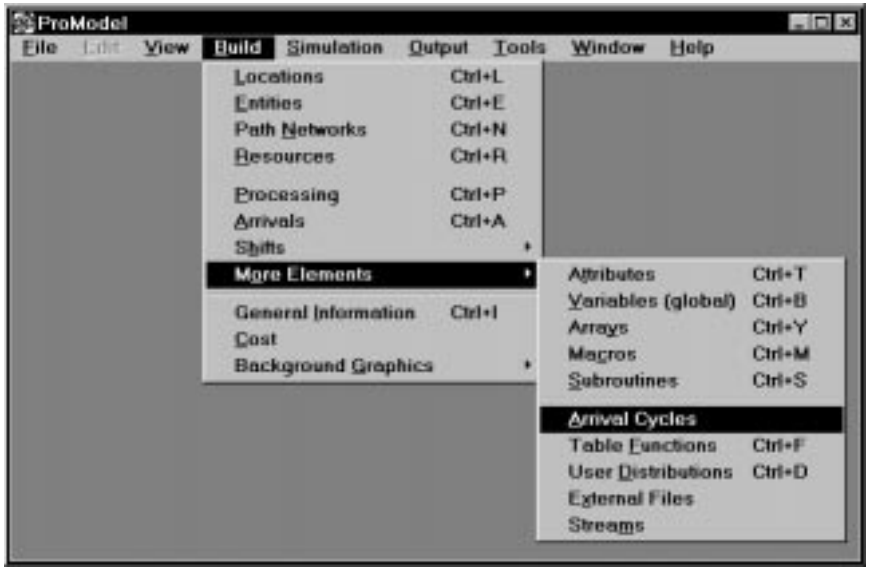
Macros may be used in any expression field, but the macro may only contain an expression (i.e., Entries(LOC1), U(5,1), etc.). When a macro is used in a logic field, the macro may include any logic element that is valid in that logic field.

Subroutines may also be used in an expression field provided that the RETURN statement is used to return a value to the expression field. When a subroutine is used to represent one or more logic statements, the subroutine may only include statements valid for the particular context.

8.6 Arrival Cycles

An arrival cycle is a pattern of individual arrivals which occurs over a certain time period. Some examples of arrival cycles that exhibit a pattern are the arrival of customers to a store and the arrival of delivery trucks to a truck dock. At the beginning of the day, arrivals may be sparse; but as the day progresses, they build up to one or more peak periods and then taper off. While the total quantity that arrives during a given cycle may vary, the pattern or distribution of arrivals for each cycle is assumed to be the same.

Arrival Cycles are defined in the Arrival Cycles edit table, accessed through the Build Menu.

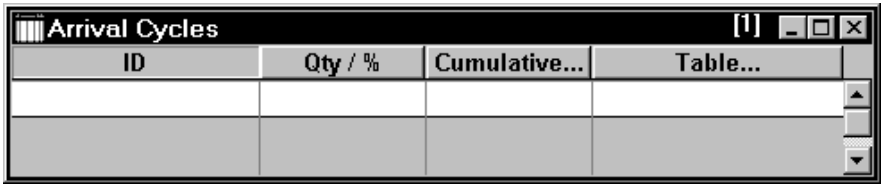


How To **Edit Arrival Cycles:**

1. Choose **More Elements** from the **Build** Menu.
2. Choose **Arrival Cycles**.

8.6.1 Arrival Cycles Edit Table

Arrival cycles are defined by entering the proper data into the Cycle edit table. The fields of the Cycle edit table are explained below.



ID The cycle name.

Qty / % Select either Percent or Quantity as the basis for the total number of arrivals per cycle occurrence.

Cumulative... Select Yes to specify the % or Qty values in a cumulative format. Select No to specify these fields in a non-cumulative format.

Table... Click on this button (or double click in this field) to open an edit table for specifying the cycle parameters.

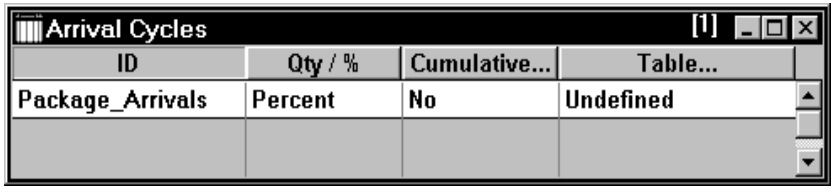
8.6.2 Arrival Cycles Example

Suppose we are modeling the operations of a material handling system (or any manufacturing system) and we need to specify a pattern for material arrivals. From past data, we know that packages arrive throughout the day (9:00 AM to 5:00 PM) according to the following approximate percentages.

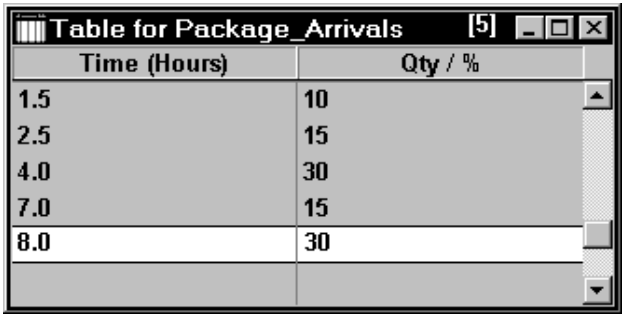
From	Before	Percent
9:00 AM	10:30 AM	10
10:30 AM	11:30 AM	15
11:30 AM	1:00 PM	30
1:00 PM	4:00 PM	15
4:00 PM	5:00 PM	30

Defining the Arrival Cycle

The arrival cycle will be named Package_Arrivals in the Arrival Cycles edit table. Because the data is expressed in terms of percentages, we select Percent as the basis for the cycle. Also, the percentage values are not cumulative so we specify No in the Cumulative field.



Next, we click on the Table heading button to open another edit table for entering the cycle data.

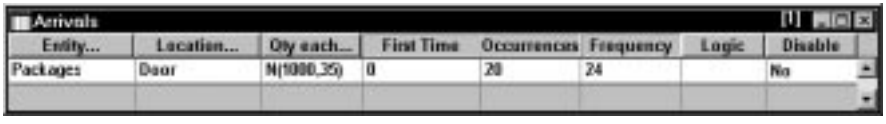


In this example, even though the percentages of packages that have arrived are not cumulative, the time is always cumulative. Therefore, the table reads as follows: ten percent of the daily customers arrive in the first 1.5 hours, fifteen percent of the packages arrive between hour 1.5 and 2.5, and so on. The arrivals are randomly distributed during the time interval in which they arrive.

The arrival cycle is now defined and can be assigned to an arrivals record in the Arrivals edit table.

Assigning Arrivals to the Arrival Cycle

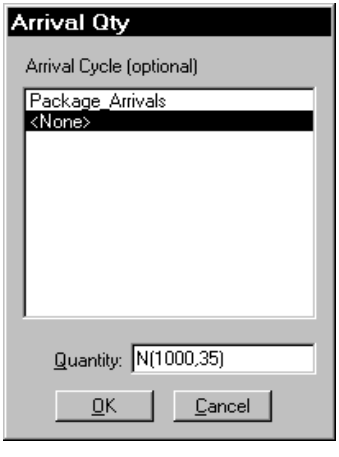
After an arrival cycle has been defined, the next thing to do is to assign an arrival record to the arrival cycle in the Arrivals edit table for the arriving packages. The arrival record for the example appears below. (See *Arrivals* on page 315 for more information.)



The total number of packages per day is normally distributed with a mean of 1000 and a standard deviation of 35. The number of occurrences of the cycle is 20, which represents 20 working days (1 month) of time. The frequency in this case refers to the period of the cycle or the time between the start of one cycle and the start of the next cycle, which is every 24 hours. Make sure the arrival frequency is defined with the same time unit as the arrival cycle.

Without an arrival cycle, all the packages for a single day would arrive at the start of the simulation. An arrival cycle will divide the quantity specified in the Qty each field into various sized groups which will arrive throughout the day.

To assign the arrival cycle to the arrival record, simply click on the Qty each heading button to open a dialog with the names of all defined cycles.



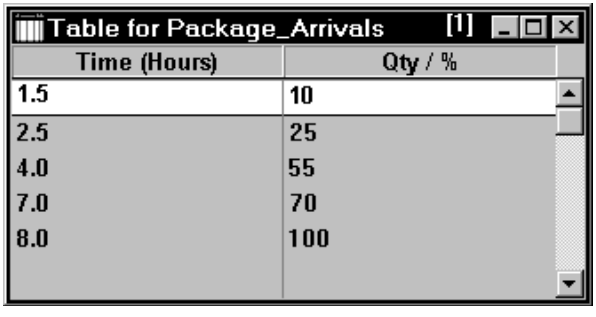
Click on the entry Bank_Arrivals and select OK. The Qty each field now includes the cycle.

8.6.3 Cumulative Cycle Tables

In the previous example, percentages were expressed non cumulatively. This same data could have been expressed cumulatively as follows:

Before	Percent
9:00 AM	0
10:30 AM	10
11:30 AM	25
1:00 PM	55
4:00 PM	70
5:00 PM	100

The data is now expressed cumulatively and could be entered in the cycle table as follows.



To specify cycles in cumulative form, simply choose Yes in the Cumulative field of the Arrival Cycles edit table.

Note Time values remain cumulative regardless of the form of the percentages.

8.6.4 Arrival Cycles by Quantity

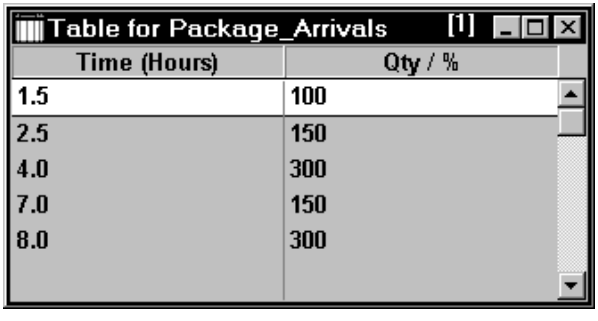
The previous example was based on the assumption that a certain percentage of arrivals came within a specified time interval. An alternate method of specifying an arrival cycle is to specify the number of arrivals to arrive within each time interval.

Example 1

Suppose that in the material handling example we knew that for each cycle period, the number of packages to arrive during each time interval within the cycle period is as follows:

From	Before	Number
9:00 AM	10:30 AM	100
10:30 AM	11:30 AM	150
11:30 AM	1:00 PM	300
1:00 PM	4:00 PM	150
4:00 PM	5:00 PM	300

With the data in this format, we specify the Arrival cycle by choosing Qty in the "Qty/Percent" field and complete the cycle table as follows:

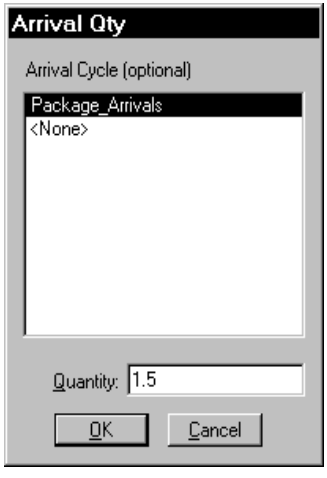


This table could also be specified cumulatively by choosing Yes in the Cumulative field and entering the quantity values in a cumulative format.

Note When specifying Arrival cycles by quantity, the value entered in the "Quantity each" field of the Arrivals edit table changes meaning. Instead of the total arrivals per cycle, it represents a factor by which all entries in the cycle table will be multiplied. This field may be any valid expression which evaluates to a number. This allows the same arrival cycle to be entered for more than one arrival record that has a different factor applied to it. For example, you might want to define a different factor to a package arrival pattern depending on the day of the week. In this case, define an arrival record for each day with a frequency of one week.

Example 2

Suppose we wish to see the effect on the material handling example if the number of arrivals is increased by 50%. The relative quantities per time interval remain the same but now 50% more packages arrive each day. Using the data from the previous example, we enter the same values in the Arrival cycle quantity fields, but specify a value of 1.5 in the quantity field of the Arrival edit table.

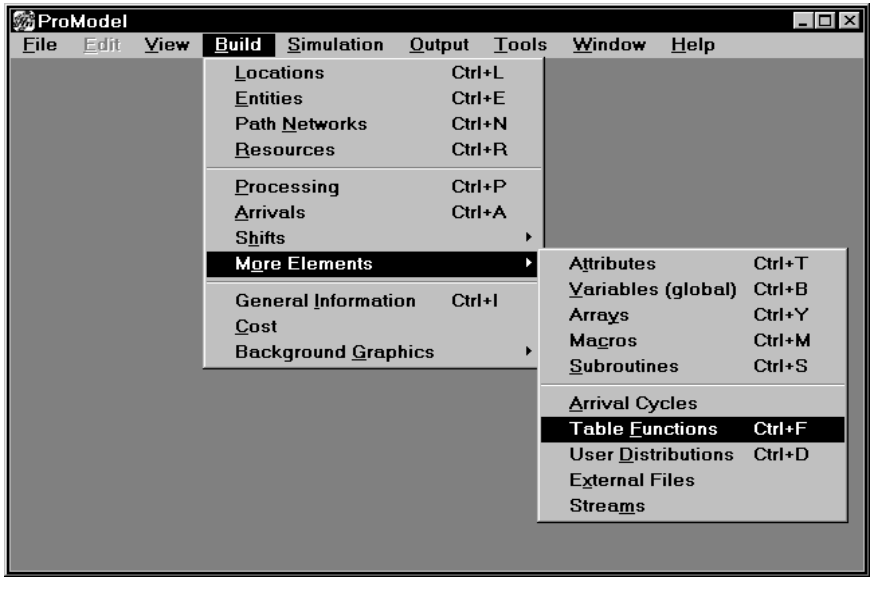


The result is an arrival schedule with the following parameters.

From	Before	Quantity
9:00 AM	10:30 AM	150
10:30 AM	11:30 AM	225
11:30 AM	1:00 PM	450
1:00 PM	4:00 PM	225
4:00 PM	5:00 PM	450

8.7 Table Functions

Table functions provide an easy and convenient way to retrieve a value based on an argument (i.e., some other value) that is passed to the table. Table functions specify a relationship between an independent value and a dependent value. All table functions are defined in the Table Functions editor which is accessed from the Build menu.



How To **Access the Function Table Editor:**

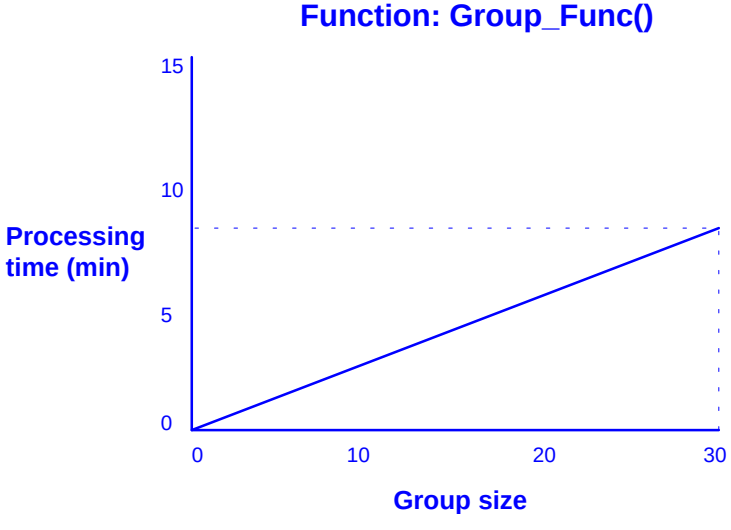
1. Select **More Elements** from the **Build** menu.
2. Select **Table Functions**.

8.7.1 Table Functions Editor

Table functions are defined by the user and return a dependent (or look-up) value based on the independent (or reference) value passed as the function argument. Independent values must be entered in ascending order. If the independent value passed to a table function falls between two independent values, a dependent value for the unspecified reference value is calculated by linear interpolation. The following two examples show how to specify a linear function (where only two reference values are needed to define the entire function) and a nonlinear function (where more than two reference values need to be specified).

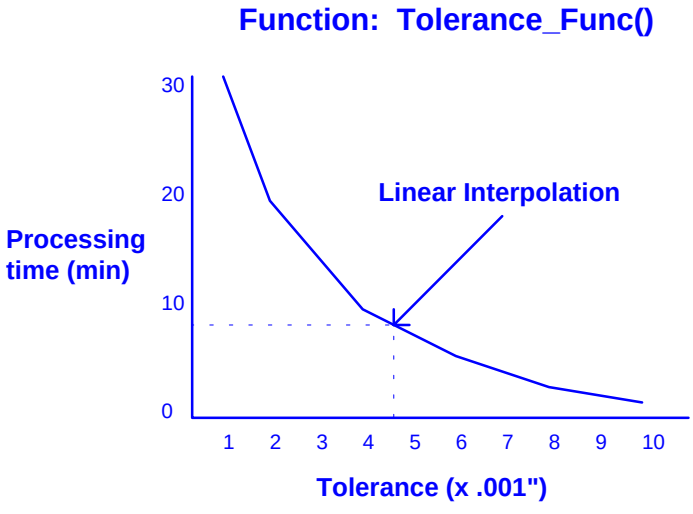
Example 1

The example below shows a linear relationship between the number of entities in a group and the time required to process the group. As the number of entities in the group increases, the time required to process the group also increases. The relationship in this case is a linear relationship, meaning the processing time is directly proportional to the number of the entities in the group. Because of the linear relationship, only the two endpoints need to be entered in the Table Function editor. (The function table for this example is given in the discussion on the Table Function editor.)



Example 2

In this example the relationship between the independent value and the dependent value is nonlinear and inversely proportional. In addition, interpolation is required to determine the dependent value if the independent value passed to the function lies between the independent values given explicitly in the table function.



The independent value in this example represents the tolerance for a machining process and the dependent value represents the processing time. Typically, processing time increases as the tolerance becomes more precise. The time required to machine with a tolerance of .001" is approximately 30 minutes compared with a processing time of approximately two minutes when the tolerance is .010". (The function table for this example is listed in the Function Editor discussion below.)

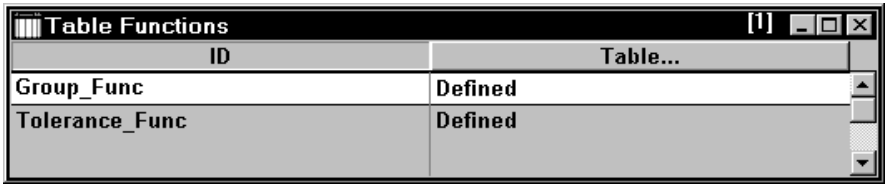
The example below shows the tolerance function used to determine the operation time at location Loc1. The tolerance required for each entity type (in thousandths of an inch) is stored in an attribute called Attr1.

Process Table **Routing Table**

Entity	Location	Operation (min)	Blk	Output	Destination	Rule	Move Logic
EntA	Loc1	**USE Res1 For Tol(Attr1)**	1	EntA	Loc2	FIRST	MOVE FOR 5

8.7.2 Table Function Edit Table

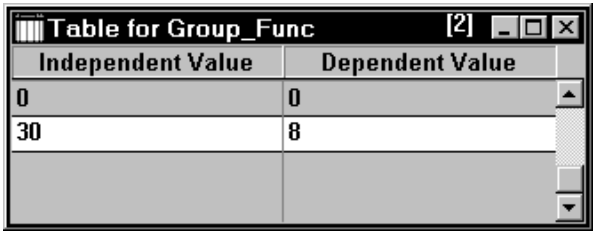
The Function Table Editor is where all function tables are created and edited. The fields of the Function Table editor are defined below.

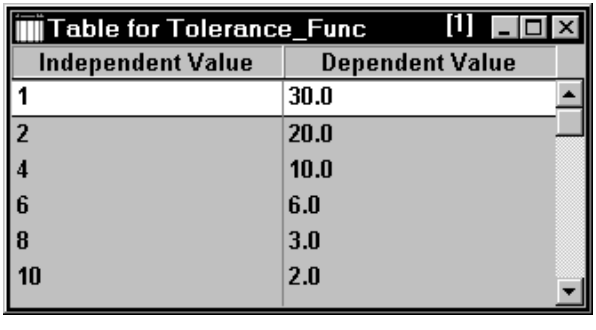


ID The name of the function table.

Table... Click on this heading button to open a table for defining the independent and dependent values of the function.

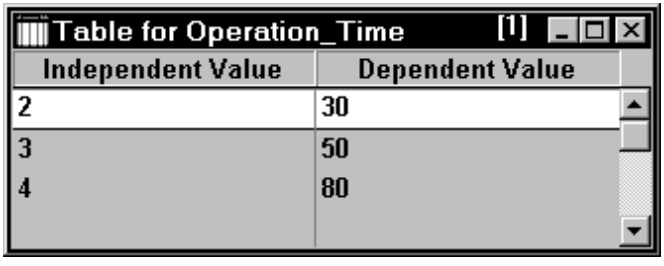
The tables for the two example functions are given below.





The independent and dependent values allow any general expression such as numbers, variables, math functions, etc. The fields are evaluated only at translation, and cannot vary during the simulation.

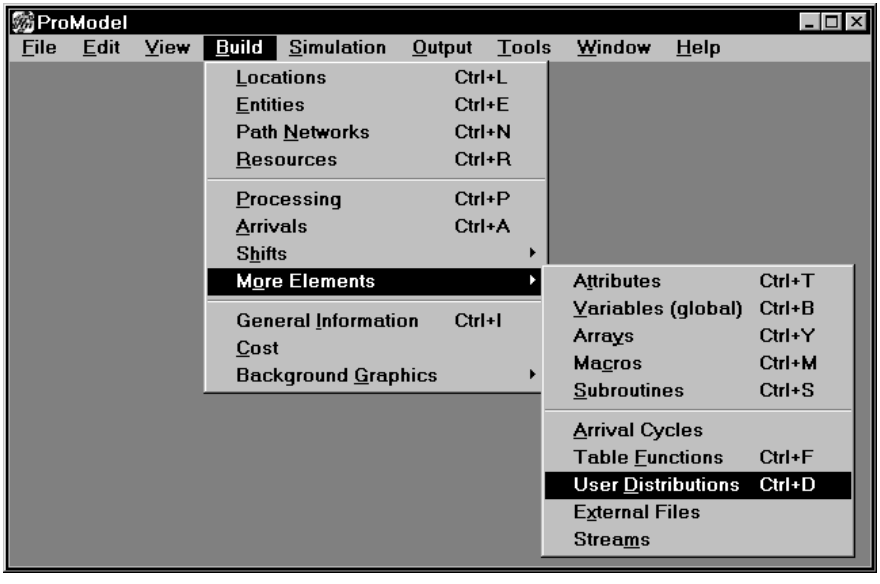
When calling a user-defined table function, if the independent value is out of range, then the table function will return a zero for the dependent value. Consider the following function table, Operation_Time:



If the function were called with the command "Operation_Time(5)," the independent value passed to the table function Operation_Time would be five. But five is beyond the limits of the table, so the dependent value returned will be zero. Likewise, if the independent value is 1, the dependent value returned will be zero. However, if 2.7 is entered as the independent value, ProModel will interpolate and return a value between 30 and 50.

8.8 User Defined Distributions

Occasionally, none of ProModel's built-in distributions can adequately represent a data set. In these cases, the user may define a User Distribution to represent the data set. User Distributions specify the parameters of user-defined (empirical), discrete, or continuous probability distributions.

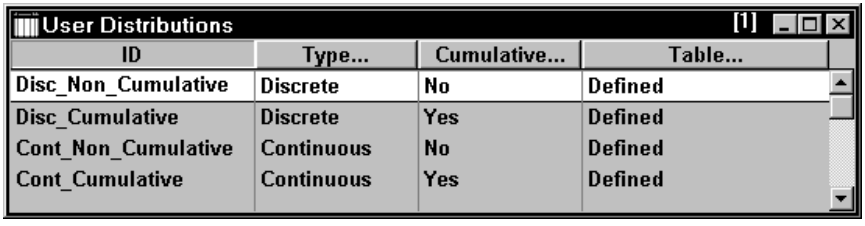


How To **Create and edit User Distributions:**

1. Choose **More Elements** from the **Build** menu.
2. Choose **User Distributions**.

8.8.1 User Distribution Edit Table

A user-defined distribution is a table of empirically gathered data. User distributions may be either continuous or discrete, and may be cumulative or non-cumulative (more information concerning these options is found later in this section). The data is entered into the User Distribution edit table. The User Distribution edit table's fields are described below.



ID The name of the distribution. When referencing distribution tables (in the operation logic, for example) the open and closed parentheses after the distribution name must be used, such as Dist1(), OpTime().

Type... Discrete or Continuous depending on the number of possible outcomes.

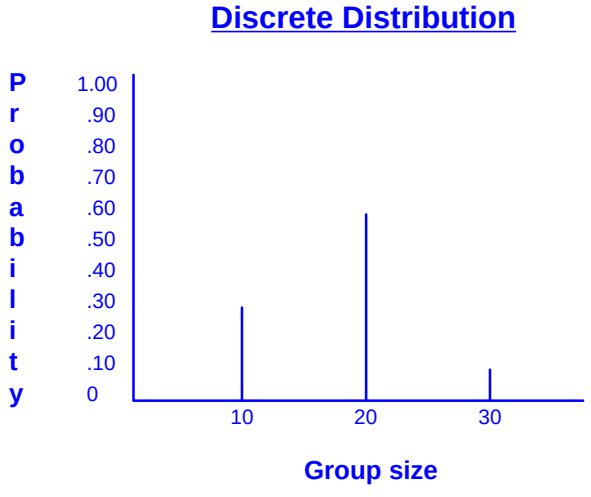
Cumulative... Yes or No depending on whether the distribution is to be specified in cumulative or non-cumulative format.

Table... Click on this button (or double click in the field) to open an edit table for defining the parameters of the distribution. Once a distribution has been defined, the field changes from "Undefined" to "Defined."

The combination of Discrete and Continuous distributions, along with the ability to express either in cumulative or non-cumulative terms, creates four possible formats for specifying distributions. The remainder of this section gives examples and procedures for specifying each of these distribution types.

8.8.2 Discrete Distributions

Discrete distributions are characterized by a finite set of outcomes, together with the probability of obtaining each outcome. In the following example, there are three possible outcomes for the group size: 30% of the time the group size will be 10, 60% of the time the group size will be 20, and 10% of the time the group size will be 30.



One way to represent a discrete distribution is by its probability mass function, listing the possible outcomes together with the probability of observing each outcome. A probability mass function for the example above could be expressed as follows (with G representing the group size).

G	10	20	30
P(G)	.30	.60	.10

An alternate way to represent a distribution is through a cumulative distribution function, listing each possible outcome together with the probability that the observed outcome will be less than or equal to the specified outcome. A cumulative distribution function for the example above could be expressed as follows.

G	10	20	30
P(G)	.30	.90	1.0

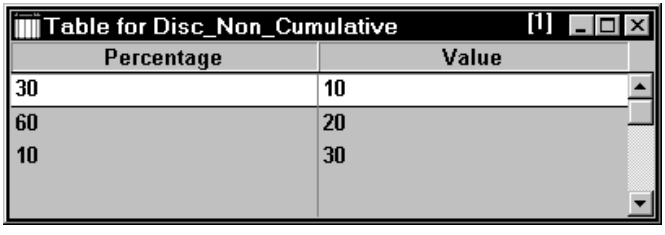
In the next example, the number of parts are grouped into a batch according to a user distribution.

Process Table **Routing Table**

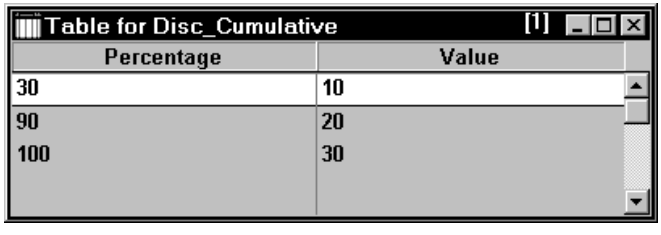
Entity	Location	Operation (min)	Blk	Output	Destination	Rule	Move Logic
EntA	Loc1	**GROUP Dist() AS Batch**					
Batch	Loc1	WAIT 10 min	1	Batch	Loc2	FIRST 1	MOVE FOR 5

ProModel provides the flexibility to specify discrete distributions according to a probability mass function or a cumulative distribution function. Select Yes or No in the Cumulative field of the Distribution edit table and fill in the table according to the probability mass function or the cumulative distribution function. The following tables show the discrete distribution example defined in both formats.

Discrete (probability mass function)

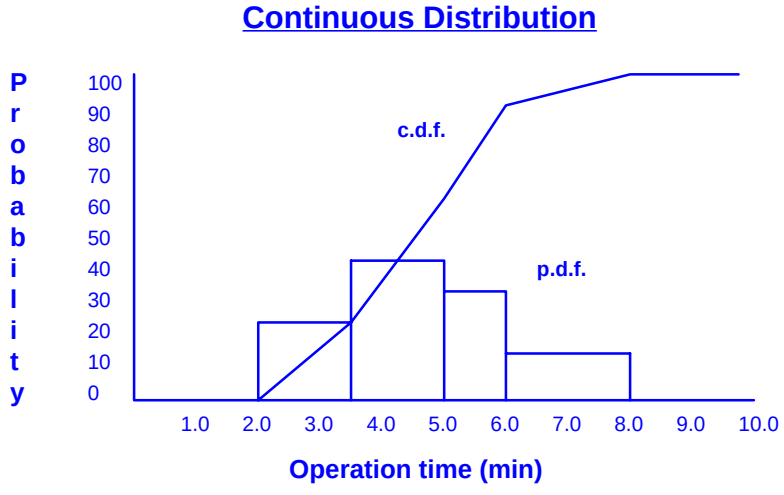


Discrete (cumulative distribution function)



8.8.3 Continuous Distributions

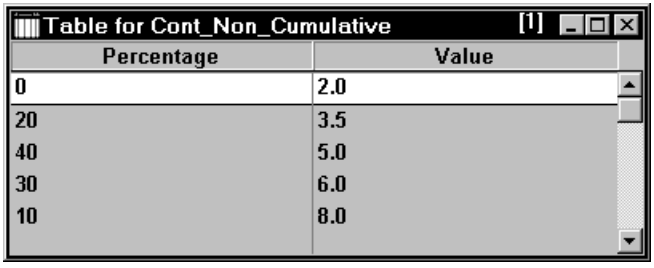
Continuous distributions are characterized by an infinite number of possible outcomes, together with the probability of observing a range of these outcomes. In the following example, there are an infinite number of possible operation times between the values 2.0 minutes and 8.0 minutes. Twenty percent of the time the operation will take from 2.0 to 3.5 minutes, 40% of the time the operation will take from 3.5 to 5.0 minutes, 30% of the time the operation will take from 5.0 to 6.0 minutes, and 10% of the time the operation will take from 6.0 minutes to 8.0 minutes.



As with a discrete distribution, a continuous distribution can be defined in two ways. A probability density function lists each range of values along with the probability that an observed value will fall within that range. Each of the values within the range has an equal chance of being observed, hence the piece-wise linearity of the c.d.f. within each range of values. A probability density function for the example above is expressed as follows.

P(0.0 <= X < 2.0) = 0.00
P(2.0 <= X < 3.5) = 0.20
P(3.5 <= X < 5.0) = 0.40
P(5.0 <= X < 6.0) = 0.30
P(6.0 <= X <= 8.0) = 0.10
P(8.0 < X) = 0.00

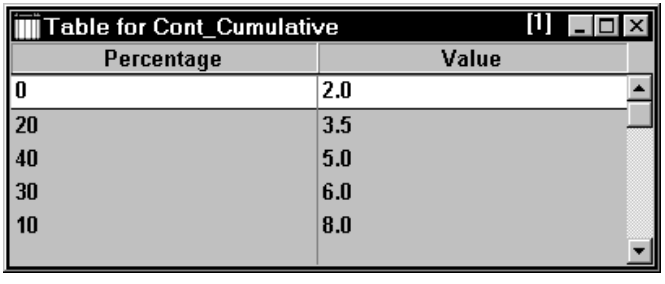
The following table represents the p.d.f. for this example.



As with a discrete distribution, a cumulative distribution function for a continuous distribution specifies the probability that an observed value will be less than or equal to a specified value. A c.d.f. for the example distribution is as follows (where x represents the return value).

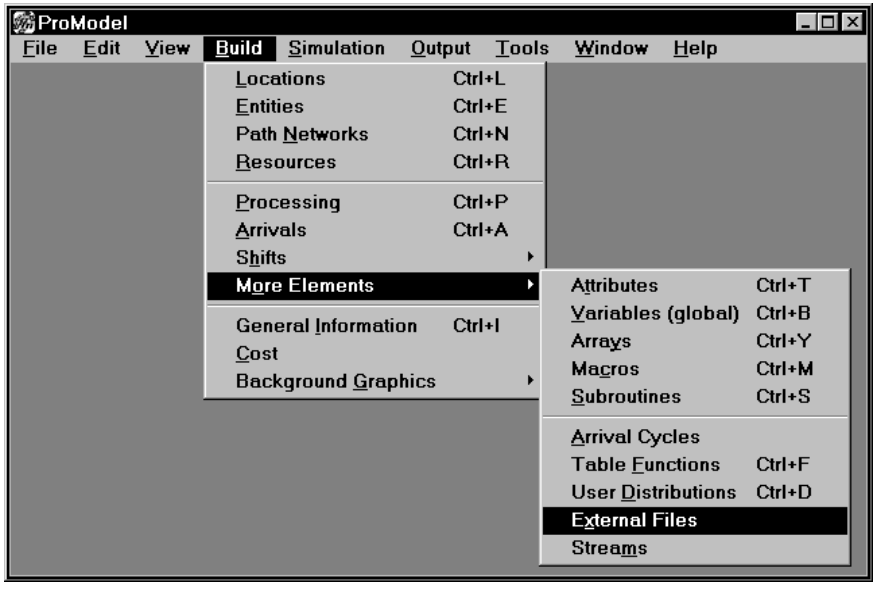
x	2.0	3.5	5.0	6.0	8.0
P(X <= x)	0	.20	.60	.90	1.0

The following table represents this c.d.f.



8.9 External Files

External files may be used during the simulation to read data into the simulation or write data as output from the simulation. Files can also be used to specify such things as operation times, arrival schedules, shift schedules, and external subroutines. All external files used with a model must be listed in the External Files Editor which is accessed from the Build menu.

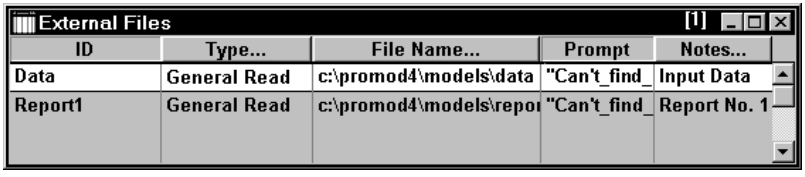


How To **Define external files:**

1. Select **More Elements** from the **Build** menu.
2. Select **External Files**.

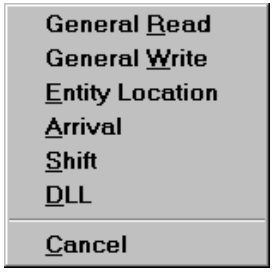
8.9.1 External Files Editor

The External Files Editor consists of an edit table with fields specifying the external files to be used during the simulation. Each of these fields is defined below.



ID An alias to be used in the model for referencing the file. Note that this ID does not have to be the same as the DOS file name.

Type Click on this heading button to display the following menu. The six file types are discussed in the remainder of this section.



File Name... The actual DOS file name, including the path. Press the heading button or double click in this field to open a dialog box for choosing the filename.

Prompt A prompt to be displayed at run-time in the event the specified DOS file cannot be opened.

Notes... A general notes field for entering descriptive information about the file. Click the heading button or double click in this field to open a larger window for entering notes.

8.9.2 File Types

External files may be defined as one of six types depending upon the purpose of the file.

General Read File

A General Read file contains numeric values read into a model using a READ statement. Values must be separated by a space, comma, or end of line. Any non-numeric data will be automatically skipped when obtaining the next numeric value (See *Read* on page 205 of the *ProModel Reference Guide*). For example, if you specify a normal distribution such as N(5,1) in the General Read file, ProModel will not return a numeric value following the distribution. Instead, it will read in the first value, 5, and the next value, 1.

A General Read file must be an ASCII file. Data created in a spreadsheet must be saved as a text file.

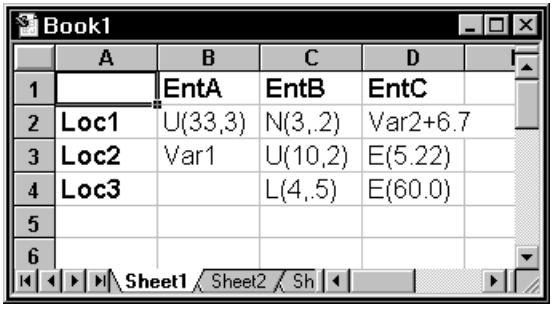
General Write File

A General Write file is used for writing text strings and numeric values using the WRITE, and WRITELINE statements. Text strings are enclosed in quotes when written to the file, with commas automatically appended to strings. This enables the files to be read into spreadsheet programs like Excel or Lotus 1-2-3 for custom viewing, editing, and graphing. Write files may also be written to using the XWRITE statement which gives the modeler full control over output and formatting. (See *Write* on page 252, *WriteLine* on page 254, and *Xwrite* on page 258 of the *ProModel Reference Guide*.)

If you write to an external file during multiple replications or a single, independent run, the data will be appended to the data from the previous replication. However, if the RESET statement is used, the data is overwritten for each replication.

Entity-Location File

An Entity-Location file (or expression file) is a spreadsheet (WK1 format only) file containing numeric expressions listed by entity and location name. Entity names should appear across the top row, beginning in column 2, while location names should be entered down the first column beginning on row 2. A numeric expression for each Entity-Location combination is entered in the cell where the names intersect. An example of a spreadsheet file is shown next.



To use the value stored in an Entity-Location file as an operation time, call out the file identifier in the operation logic as shown in the following example. (In this example, "SvcTms" is the File ID of the desired Entity-Location file.)

Process Table

Entity	Location	Operation (min)
EntA	Loc1	WAIT SvcTms()

Routing Table

Blk	Output	Destination	Rule	Move Logic
1	EntA	Loc2	FIRST 1	MOVE FOR 5

By specifying SvcTms() with no arguments in the parenthesis, a value is returned from the Entity-Location File "SvcTms" for the current entity at the current location, i.e., EntA at Loc1. You may also return the value stored in any other cell of an Entity-Location file by explicitly specifying the entity and location names in the parentheses, e.g., SvcTms(EntB, Loc1) or SvcTms(EntC, Loc2).

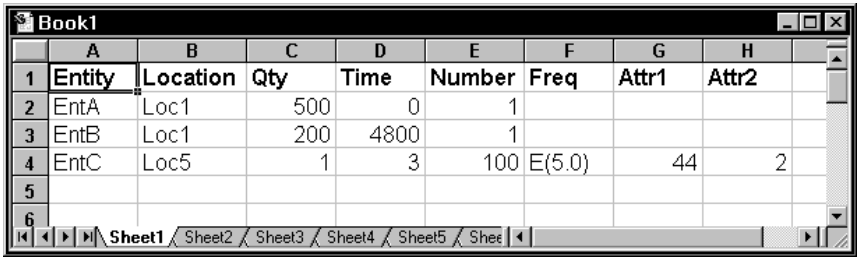
Arrivals File

An Arrivals file is a spreadsheet (WK1 format only) file containing arrival information normally specified in the Arrival Editor. One or more arrival files may be defined and referenced in the External Files Editor. Arrival files are automatically read in following the reading of the Arrival Editor data. The column entries must be as follows:

Column	Data
A	Entity name
B	Location name
C	Quantity per arrival
D	Time of first arrival
E	Number of arrivals
F	Frequency of arrivals
G through...	Attribute assignments

Columns A through F may have any heading desired as long as the data is of the proper type. If attributes are to be assigned, columns G and higher should have headings that match the names of the attributes being assigned. The following example illustrates these points.

Example



The values in the spreadsheet cells must be a numeric expression as opposed to a formula commonly used in spreadsheets. For example, if cell E4 in the spreadsheet above was actually a formula generating the value 100, the value ProModel generates is zero. ProModel only recognizes expressions for the Qty, Time, Number, and Frequency columns in a spreadsheet.

When defining an External Arrivals File, you do not need to define arrivals in the Arrivals edit table. If several entities are scheduled to arrive at the same time, entities arrive in the system according to the order in which they appear in the arrival list. However, when there is more than one occurrence for the arrival record, the next entity will not arrive until the frequency has elapsed. Meanwhile, other entities listed below the record may be allowed to arrive.

Shift File

A shift file record is automatically created in the External Files Editor when you assign a shift to a location or resource. If shifts have been assigned, the name(s) of the shift file(s) will be automatically created in the External Files Editor. If no path is listed for the shift file, ProModel will search in the default models directory.

Note Creating a shift file record in the External Files Editor should not be done. It is done automatically through the shift assignment.

DLL File

A DLL file is needed when using external subroutines through the XSUB() function. See *Subroutines* on page 447.

Other External Files

In addition to allowing the user to define external files, ProModel creates other external files. ProModel automatically creates and/or opens files depending on the specifications in the model. Below is a description of the different files ProModel creates (* indicates files that remain open while the model is running):

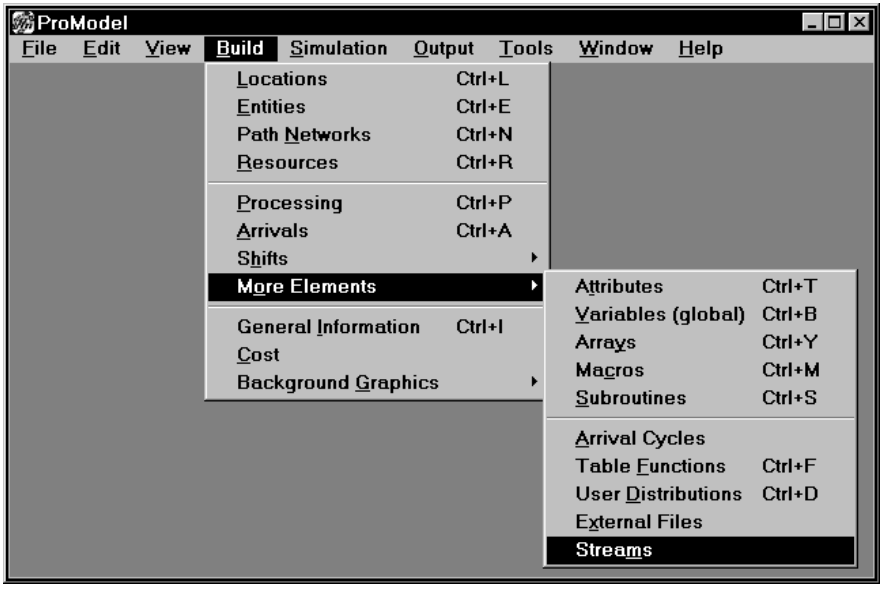
Extension	Type	Description
CSV	ASCII	Export Data (comma delimited)
MOD*	Binary	Model File
RDB	Binary	Output Database (basic statistics)
RDT	Binary	Output Time Series data
RDW	Binary	Output Reports & Graphs saved for a model
SED*	Binary	Seed File used to store seed values for each replication
SFT	Binary	Shift created when defining a shift in the shift editor
TRC*	ASCII	Trace File
GLB	Binary	Graphic Library Files
GBM	Binary	Graphic Bit Map file created when loading a model file.

Open Files

Depending on the model specifications, there may be some occasions where several files need to be open simultaneously to execute the model. There is a feature which allows the user to access up to 255 open files at any time. The [General] section of the **promod4.ini** file contains the following statement: OpenFiles=*n* where *n* is the number of files between 20 and 255. The default is 40. To change the number of available open files, simply edit the **promod4.ini** file such that OpenFiles equals the desired number and then restart ProModel. There is also the option to close files using the CLOSE statement (see *Close* on page 127 of the *ProModel Reference Guide*).

8.10 Streams

A stream is a sequence of independently cycling, random numbers. Streams are used in conjunction with distributions. Up to 100 streams may be used in a model. A stream generates random numbers between 0 and 1, which in turn are used to sample from selected distributions. By default, all streams use seed value #1 and are not reset between replications if multiple replications are run. To assign a different starting seed value to a stream or to cause the seed value to be reset to the initial seed value between replications, use the Streams Editor. The Streams Editor is accessed from the Build menu as shown below.

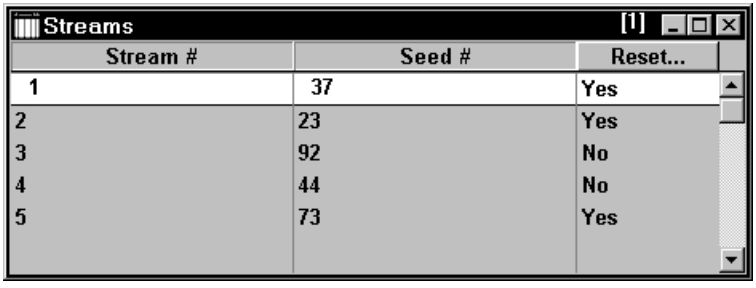


How To **Access the Streams Editor:**

1. Select **More Elements** from the **Build** menu.
2. Select **Streams** from the submenu.

8.10.1 Streams Edit Table

The Streams Editor consists of an edit table with three fields. The fields of the Streams edit table are explained below.



Stream # The stream number (1 to 100). This number identifies each stream.

Seed # The seed value (1 to 100). Streams having the same seed value generate the same sequence of random numbers. For more information on seed values, see *Using Random Number Streams* on page 483.

Reset... Set this field to YES if you want the stream to be reset to the initial seed value for each model replication. Set this field to NO if you want the stream to continue where it left off for subsequent replications.

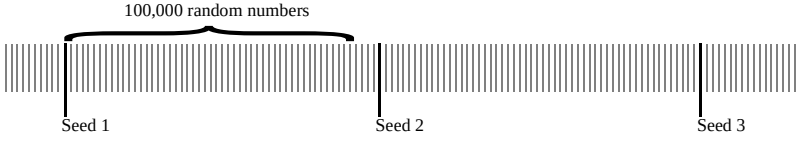
The Streams Editor pictured above shows five defined streams. Each stream has a unique seed value. Streams 1, 2, and 5 will be reset for each replication, while streams 3 and 4 will continue where they left off for subsequent replications.

8.10.2 Using Random Number Streams

One of the most valuable characteristics of simulation is the ability to replicate and isolate probabilistic functions and activities within a system for specific study. In the real world, events tend to occur randomly, according to a certain statistical pattern or distribution. To help you model this randomness, ProModel uses distributions.

When you include a distribution (e.g., Normal, Beta, and Gamma) in your model, ProModel uses a random number generator to produce a set sequence or *stream* of numbers between 0 and 1 (0 <= x < 1) to use in the distribution. Before it can select any numbers, however, ProModel requires an *initial seed value* to identify the point in the stream at which to begin. Once you specify a seed value, ProModel "shifts" the random number selection (in increments of 100,000 numbers) by that number of positions and starts sampling values. Since there is only one random number stream, this will ensure that the selected values do not overlap. ProModel includes 100 seed values, and each seed produces a unique set of random numbers. If you do not specify an initial seed value, ProModel will use the stream number as the seed value (i.e., stream 3 uses seed 3).

Random Number Stream



When you use a specific seed value (e.g., 17), ProModel produces a unique sequence of numbers to use each time you apply that seed value. This allows you to maintain the consistency of some model elements and permit other elements to vary. (To do this, specify one random number stream for the set of activities you wish to maintain constant, and another random number stream for all other sets of activities.) In fact, because each seed value produces the same sequence of values every time, completely independent model functions must use their own streams. For example, Arrival distributions specified in the Arrival Module should have a random number stream used nowhere else in the model. This will prevent activities that sample random stream values from inadvertently altering the arrival pattern (i.e., the activities will not affect the sample values generated from the arrival distribution).

Note The random number generator is a prime modulus multiplicative linear congruential generator. The C code implementation for most of the random variates was written by Stephen Vincent and based on the algorithms described by Law and Kelton (see *Appendix A General References* on page 661).

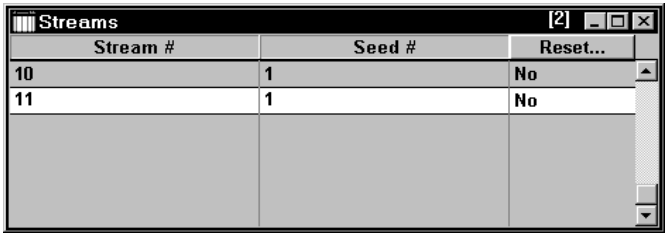
8.10.3 Stream Example

The following example shows one reason why multiple streams are useful.

Two machines, Mach1 and Mach2, are to go down approximately every 4 hours for servicing. To model this, the frequency or time between failures is defined by a normal distribution with a mean value of 240 minutes and a standard deviation of 15 minutes, N(240,15). (For more information on distributions, see *User Defined Distributions* on page 469). The machines will go down for 10 minutes. Because no stream is specified in the normal distribution, ProModel uses stream number one to generate sample values for both machines. So if the next two numbers in stream number one result in sample values of 218.37 and 264.69, Mach1 will receive 218.37 and Mach2 will receive 264.69. Therefore, the two machines will go down at different times, Mach1 after almost four hours and Mach2 after somewhat more than four hours.

Suppose, however, that the resource to service the machines must service them both at the same time. The machines would have to go down at the same time. Stream number one will not bring them down at the same time because stream number one sends the machine's distributions two different numbers. Using two streams (in the example below numbered ten and eleven) with the same initial seed will ensure that the machines receive the same random number every time. The two streams have the same starting seed value so they will produce exactly the same sequence of random numbers. The first number of stream ten will be exactly the same as the first number of stream eleven; the second numbers will be the same; indeed, every number will be the same.

The Streams window below shows streams ten and eleven with the same seed values.



By assigning stream ten to Mach1 and eleven to Mach2, both machines will go down at exactly the same time.

Clock downtimes for Mach1 [1]

Frequency	First Time	Priority	Scheduled...	Logic...	Disable
N(240,15,10)	0	99	No	WAIT 10	No

Clock downtimes for Mach2 [1]

Frequency	First Time	Priority	Scheduled...	Logic...	Disable
N(240,15,10)	0	99	No	WAIT 10	No

This first clock downtime for each machine will occur at the beginning of the simulation. After that, the first downtime occurs at 218.37 minutes for both machines. Defining different seed values for the streams would produce different sequences and therefore different downtimes. Using the same stream number for the clock downtimes would also produce different values. But two different streams with the same seed value will bring both machines down at the same times.

Note that if a third machine were to use one of the streams, for example if Mach3 were to use stream eleven, the two machines would no longer go down together. Mach1 would use the first value from stream ten; Mach2 would use the first value from stream eleven; and Mach3 would use the second value from stream eleven. The first time Mach1 and Mach2 went down, they would go down at the same time because the first number in streams ten and eleven is the same. But thereafter they would not. The second time they would go down at different times because Mach1 would receive the second value from stream ten, Mach2 would receive the third value from stream eleven, and Mach3 would receive the fourth value from stream eleven.

Note

Stream notes:

1. If a stream value is not specified for a distribution, the stream is assumed to be stream one. Stream #1 does not automatically reset after each replication.
2. The stream parameter is always the last parameter specified in a distribution unless a location parameter is also specified. See *Distribution Functions* on page 87 of the *ProModel Reference Guide* for details.