Extend™

Professional simulation tools

Version 5 Programmer's Reference for:

Extend
Extend+Manufacturing
Extend+BPR
Extend+Industry
Extend Suite
Industry Suite

For Windows or Macintosh

Imagine That, Inc. • 6830 Via Del Oro, Suite 230 • San Jose, CA 95119 USA
Telephone 408-365-0305 • FAX 408-629-1251
Email: extend@imaginethatinc.com • Web Site: http://www.imaginethatinc.com

Extend was created by Bob Diamond

Extend 5 was designed and written by Bob Diamond, Steve Lamperti, and Dave Krahl

The Extend Discrete Event library architecture was created by Steve Lamperti and Dave Krahl

Extend libraries designed and written by: Dave Krahl, Steve Lamperti, Anthony Nastasi, and Bob Diamond

Contents

Introduction

This Programmer's Reference is a companion to the Extend User's Guide. The purpose of this manual is to show the inner workings of Extend blocks and guide the Extend block developer in creating custom blocks and libraries of blocks. The Programmer's Reference presumes that you have read and are familiar with the Extend User's Guide.

How to start

Chapter 1 is an introduction to the internals of a block. This chapter is also helpful for non-programmers, to understand how the blocks in their models work. Chapters 2 and 3 explain how to create your own blocks and how to modify the blocks that come with Extend. Chapter 4 gives ModL coding tips, and Chapter 5 shows how to use the ModL source-code debugger. As you will see, programming blocks in Extend is quite easy, especially if you have ever programmed in any language.

On-line help

Extend's on-line help is available any time you are using Extend:

 Macintosh: select the Extend Help command in the Help menu.

 Windows: select the Help command in the Help menu or press F1 on your keyboard.

The block help from any libraries which are open is automatically included with Extend's on-line help topics. This help is also accessed through a button labeled "Help" in each block's dialog.

Technical support

You must be a registered user to receive technical support, so be sure to mail your registration card immediately after purchasing Extend.

We always appreciate hearing from you, and it would help us if you provide us with the following information when you contact us:

- Your Extend serial number, which is located in the front of your manual.

- Your name and phone number (email and FAX number are also helpful), in case we need to return your call.
- The version of Extend you are using:
  - *Macintosh*: this number is located in the About Extend command under the Apple menu.
  - *Windows*: this number is located in the About Extend command under the Help menu.
- The version of the libraries you are using:
  - *Macintosh*: the version number is contained in the Finder's Get Info box. To see it, select the library file in the Finder, then select Get Info from the File menu. Information on the library version, size, and last modification date is also listed at the top of each Library Window.
  - *Windows*: the version number is located in the Library Window at the top.
- The type of computer you are using, such as PowerMacintosh 8100 or Pentium 230.
- The version number of the operating system you are using, such as "Windows 95" or "Macintosh System 8.0".
- Anything else about your machine configuration: hard disk, RAM Cache, video cards, etc.

Telephone:408-365-0305 (8:30 AM to 5:30 PM Pacific Standard Time, Monday through Friday)
Email:support@imaginethatinc.com
FAX:408-629-1251
Mail:6830 Via Del Oro, Suite 230, San Jose, CA 95119 USA
Web Site:http://www.imaginethatinc.com

Chapter 1: Parts of a Block

An introduction to the internals of a block so that you can create blocks of your own

"We shape our buildings;
thereafter they shape us."
— Winston Churchill

Up to this point, you have only seen how to use Extend with the blocks that are provided in the package. These blocks work as described, but you may wonder exactly what makes them work. Once you understand what goes into a block, it is a very easy task to modify existing blocks and create blocks of your own.

You should read the chapters at the beginning of the Extend User's Guide before reading this chapter. Many of the concepts in building blocks are based on your understanding of how blocks are used in models.

This chapter describes the internal parts (structure) of standard blocks.

Chapter 2: ModL Code Overview and Block Creation, describes the "action" part of a block, the ModL code, in much more detail and shows you how to build a new block.

Chapter 3: The ModL Language, is a reference for ModL, the programming language used in blocks,

Chapter 4: ModL Programming Techniques gives coding suggestions including special instructions if you program discrete event blocks, and

Chapter 5: Using the ModL Source Code Debugger will show you how to debug your blocks.

Table of block internals

Blocks have six parts:

Part	Description
Dialog	The dialog that you see when you double-click on the block's icon. You specify all the buttons, text, and entry boxes that go into the dialog, and these determine the dialog's size. You can separate the dialog items into functional groupings using *tabs* which appear at the top of the dialog. The empty text box to the left of the bottom scroll bar is the dialog's *label* field. Although block labels are user-enterable, they can also be accessed by the ModL code.
ModL code	The ModL program that makes a block work. The program reads information from the connectors, dialog, and the model environment and produces output that can be used by other blocks. A block can also query and control other blocks in the model through its code.
Icon	The icon you see in the model. You can draw the icon with Extend's drawing environment or your own painting program, or copy clip art and paste it in.
Animation	What you see if Show Animation is selected when you run the simulation. Animation can be displayed on a block's icon and may also show outside of the icon.

Part	Description
Connectors	The input and output connectors on the block. These appear in the icon and transmit information to and from the ModL code. Blocks can also transmit information without using connectors, through block messaging and global variables.
Help text	The text that appears when you click the Help button in the block's dialog. The help text for each block is also available in the Help command from the Help menu.

These parts are interconnected. For example, the ModL code reads information from the connectors, the help text is displayed through the dialog, and so on.

Opening an existing block

There are three methods you can use to open an existing block's internal structure:

- Select the block by clicking once on the block's icon in a model, or right-click the block. Then choose Open Structure from the Define menu, ***or***
- Double-click on the block's icon in a model while holding down the Option (*Macintosh*) or Alt (*Windows*) key, ***or***
- Double-click on the block's icon in the Library Window while holding down the Option (*Macintosh*) or Alt (*Windows*) key. You can open the Library Window by choosing Open Library Window at the top of each library's menu.

The Accumulate block is a good example for investigating the internals of a block. Note, however, that any changes you make to any block in Extend's libraries will affect all existing blocks in all your existing models. Thus, it is very important that you don't make any changes to these blocks unless you are very sure you want those changes. For this chapter, use the Accumulate block from the **Practice** library (*Macintosh:* Practice Lib; *Windows*: practice.lix). If you make changes to this block, it won't affect the main Extend libraries.

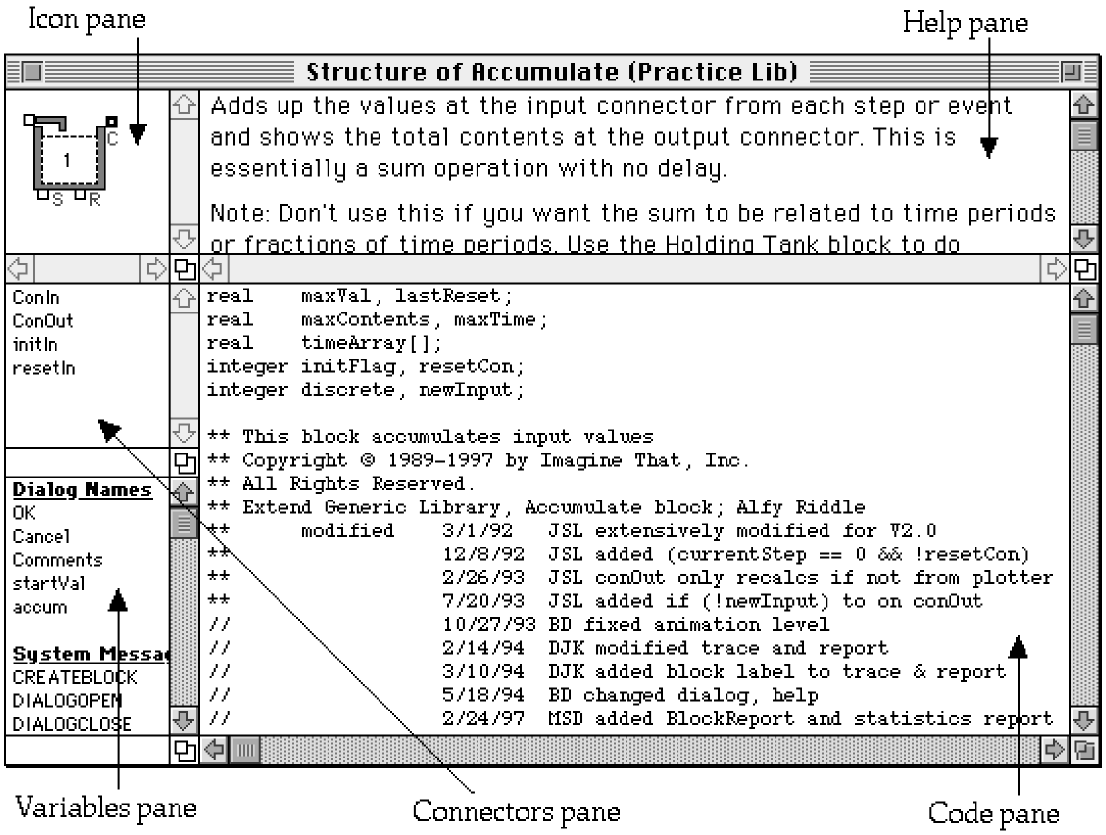
Open the internal windows for that block:

- Choose Open Library from the Library menu. In the dialog, select and open the Practice library (*not* the Generic library).
- Drag down in the Library menu to the Practice library and choose Open Library Window from the top of the hierarchical menu. This opens the library window.
- Double-click on the Accumulate block's icon in the library window while holding down the Option (*Macintosh*) or Alt (*Windows*) key.

When you look at a block's internals, you see two windows overlapping on your screen. The first window, the *structure window*, holds all the parts of the structure other than the dialog. The second window, the *dialog window*, shows just the block's dialog. The title bars of the structure and dialog windows display the block name and the name of the library.

Structure window

The structure window looks like:



Adds up the values at the input connector from each step or event and shows the total contents at the output connector. This is essentially a sum operation with no delay.

Note: Don't use this if you want the sum to be related to time periods or fractions of time periods. Use the Holding Tank block to do

ConIn
ConOut
initIn
resetIn

Dialog Names
OK
Cancel
Comments
startVal
accum

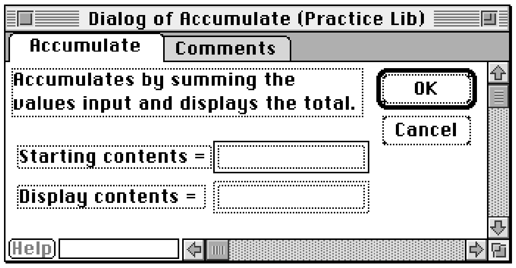
System Messa
CREATEBLOCK
DIALOGOPEN
DIALOGCLOSE

```
real    maxVal, lastReset;
real    maxContents, maxTime;
real    timeArray[];
integer initFlag, resetCon;
integer discrete, newInput;

** This block accumulates input values
** Copyright © 1989-1997 by Imagine That, Inc.
** All Rights Reserved.
** Extend Generic Library, Accumulate block; Alfy Riddle
**      modified    3/1/92   JSL extensively modified for V2.0
**                  12/8/92  JSL added (currentStep == 0 && !resetCon)
**                  2/26/93  JSL conOut only recalcs if not from plotter
**                  7/20/93  JSL added if (!newInput) to on conOut
//                  10/27/93 BD fixed animation level
//                  2/14/94  DJK modified trace and report
//                  3/10/94  DJK added block label to trace & report
//                  5/18/94  BD changed dialog, help
//                  2/24/97  MSD added BlockReport and statistics report
```

Structure window for Accumulate block (Macintosh)

Dialog window

The dialog window, behind the structure window, looks like:



Dialog of Accumulate (Practice Lib)

Accumulate | Comments

Accumulates by summing the values input and displays the total.

Starting contents =

Display contents =

Dialog window for Accumulate block (Macintosh)

The Help button (which causes the Help text from the structure window to show) and the block label field are located to the left of the bottom scroll bar.

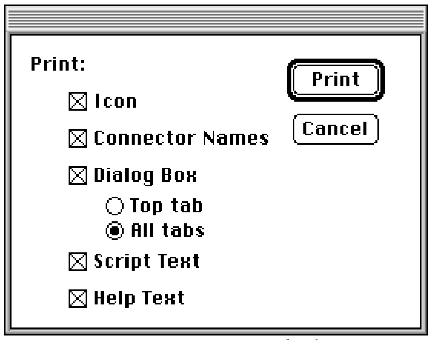
Working with the structure and dialog windows

By default, the dialog window is in front. If you want the structure window to open in front of the dialog window, select the “Structure window opens in front” choice in the Programming tab of the Preferences dialog.

You can change the size of any of the panes by dragging any of the ⧉ icons. You can also change the font used to display the ModL code by making changes in the Programming tab of the Preferences command.

To close these windows, choose Close from the File menu or click on the close box in the upper left corner of either window. If you have made any changes to either the dialog or structure windows, Extend will prompt you before saving to “Save, Compile if Needed”, “Save Without Compiling”, or “Discard Changes”. Compiling causes changes to be reflected in the code of the block.

You can print the structure window with the Print command from the File menu. You can choose which panes you want printed from the first Print dialog you see:


Print:
☒ Icon
☒ Connector Names
☒ Dialog Box
○ Top tab
● All tabs
☒ Script Text
☒ Help Text

Print Structure dialog

Dialogs

Every block has a dialog. A dialog may be as simple as some text with OK and Cancel buttons, or it may be quite complex. Setting up a dialog is quite easy because Extend has a built-in dialog editor.

A dialog contains *dialog items*, a *block label*, a *Help button* and may contain multiple *tabs*. Each dialog item has its own definition. The items in a dialog depend on how the block was defined.

For example, the Accumulate dialog shown on page P7 has two *tabs* (*Accumulate* and *Comments*) and a total of eight dialog items:

- Three text labels:

  "Accumulates by summing the values input and displays the total."

  "Starting contents ="

  "Display contents "

- Three entry boxes. Two of these boxes are *parameter* items for entering or displaying numbers and one (located on the *Comments* tab) is for entering *text*. The first box on the Accumulate tab (the one that is selected when you open the dialog) is for entering the starting value. The second one is for viewing the total accumulation. Note that the second parameter is a specialized "display only" box so that the user can see but cannot change the total accumulation.
- An OK button
- A Cancel button

The Help button and block label fields are always located to the left of the bottom scroll bar.

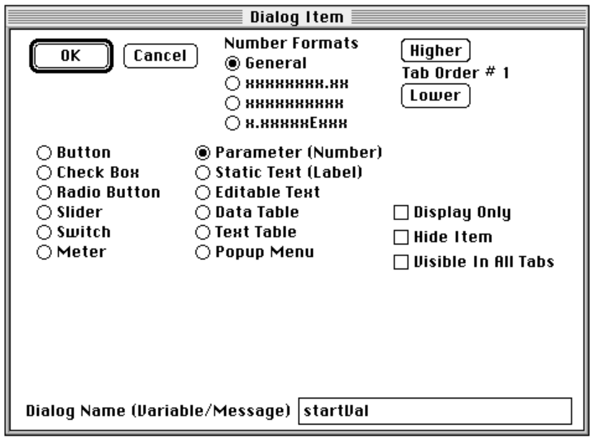
Dialog tabs

Tabs are similar to pages in a dialog and are used to group dialog items by function. You can see the names of the tabs in the upper portion of the dialog. In the Accumulate block, dialog items are separated into one tab dealing with the accumulate functions and a second tab reserved for user comments.

By clicking on a tab name, you can change which tab is foremost. To add new dialog items to a tab, bring the tab to the front of the dialog window before adding the new items. You can rename or delete a tab using the Edit Tab command from the Define menu or by double-clicking on the tab name in the dialog window. (Note that, in order to delete a tab, the tab must not contain any dialog items.)

Types of dialog items

To see the definition of a dialog item, double-click on that item in the dialog window. When you double-click on the entry box next to "Starting contents", you see:


Dialog Item

Number Formats
[✓ General]
xxxxxxxx.xx
xxxxxxxxxx
x.xxxxxExxx

Tab Order # 1

Button
Check Box
Radio Button
Slider
Switch
Meter
[✓ Parameter (Number)]
Static Text (Label)
Editable Text
Data Table
Text Table
Popup Menu

☐ Display Only
☐ Hide Item
☐ Visible In All Tabs

Dialog Name (Variable/Message) startVal

Definition of the top entry box

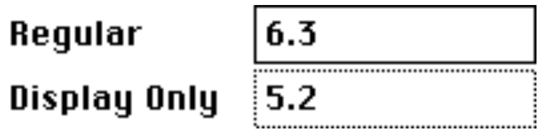
There are 12 types of dialog items:

Type	Description
Button	A button that can be clicked like an OK button or Cancel button. Buttons have *titles* that appear as text inside the button.
Check box	Square buttons that have an “X” in them when they are selected and are empty when they are not. Check boxes have *titles* that appear as text to the right of the button.
Radio button	Round buttons that appear in groups. Only one button in the group can be selected. When you select one button in a group, any other button in the group that is selected becomes deselected. Radio buttons have a group number and have *titles* that appear as text to the right of the button.
Slider	A control that resembles a slider on a stereo. You can drag the knob to change a value, or your block can move the knob to show a value.
Switch	A switch that resembles a light switch. It has two values, 0 and 1.
Meter	An output-only item that shows a needle in a meter.
Parameter (Number)	Entry box that takes or displays a number. You can specify the numeric format of the number as it appears in the box: General, integer, currency (2 decimal places), or scientific.

Type	Description
Static Text (Label)	Text that appears in the dialog.
Editable text	Entry box that takes or displays text.
Data table	A two-dimensional table (similar to a spreadsheet) for holding numbers. A table can have up to 255 columns and 3260 rows, limited to a total of 3260 cells. (You can work around this limitation by attaching a dynamic array using the DynamicDataTable function discussed on page P136 and setting the number of rows dynamically. Dynamic data tables have essentially no limit on the number of rows they can contain.) The table comes with scroll bars for moving around in the table. Column widths can be adjusted by placing the cursor directly over the column divide, and dragging the divide to the desired location while holding down the mouse button. You define the number of rows, number of columns, number format, and headings for the columns.
Text table	A two-dimensional table (similar to a spreadsheet) for holding text up to 15 characters per cell. A table can have up to 255 columns and 2030 rows, limited to a total of 2030 cells. (You can work around this limitation by attaching a dynamic array using the DynamicDataTable function discussed on page P136 and setting the number of rows dynamically. Dynamic data tables have essentially no limit on the number of rows they can contain.) The table comes with scroll bars for moving around in the table. Column widths can be adjusted by placing the cursor directly over the column divide, and dragging the divide to the desired location while holding down the mouse button. You define the number of rows, number of columns, and headings for the columns.
Popup menu	A shadowed rectangle containing a menu of items. Popup menus have *titles* for each of the items in the menu.

Display Only, Hide Item, and Visible In All Tabs options

Two of the types of dialog items, parameter and editable text, have an additional option: *Display Only*. If you choose one of these types of dialog items and check the Display Only option, the item cannot be changed through the dialog box. Instead, it can only be set through the ModL code. In the dialog window, display-only items have a gray outer border instead of the standard black border. They also appear differently in the block's dialog:



Display only item

This is a very useful feature since it prevents users from accidentally changing dialog values which the block calculates. For example, the queue blocks in the Discrete Event library display information about queue length, average wait time, and so forth. Those parameters are set to only display the results. Of course, display-only parameters can still be copied and cloned.

The *Hide Item* option serves a different purpose. After a block is built, you may want to remove a dialog item or change its nature. For example, if your dialog has a checkbox option, you may want to change it to a series of radio buttons. If you just delete the dialog item, and the block is being used in a model, the order of the data in the dialog box will be disrupted. This will cause users to have to reenter data in the affected blocks in all models. The Hide Item option provides a safe alternative to deleting or changing dialog items.

When you choose the Hide Item option, the item will not appear in the block's dialog. Items that are hidden show in the dialog window as dimmed rectangles without text. Hidden dialog items can be moved but they cannot be resized. You can also revert a hidden dialog item so that it appears in the dialog. To restore a dialog item to not being hidden, just double-click the item in the dialog window and unselect Hide Item.

Note You can also temporarily hide and show dialog items using the HideDialogItem function described in "Dialog items" on page P132.

If the dialog your are working with has tabs, the *Visible In All Tabs* option will cause the dialog item to appear on every tab of the dialog.

Tool Tips on block dialog and dialog editor windows

You may notice that the name of a dialog item is not visible when looking at the block dialog or at the dialog editor window while the block structure is open. One way to determine the dialog name is to double-click on the dialog item in the dialog editor thus opening the item's definition window. Another, simpler, way is to utilize Tool Tips. With Tool Tips turned on, you can rest your cursor above a dialog item and a small window will appear containing the name of the dialog item. Tool Tips on block dialog and dialog editor windows can be turned on or off using the libraries tab of the Preferences command in the Edit menu.

Working with dialog items

As you may remember from other programs, you can move between entry boxes by pressing the Tab key. Each parameter and editable text box in a dialog window has a tab order number. To change the order, click the Higher and Lower buttons in the definition box for that item. When you change the tab order for a dialog item, the tab order for the other dialog items is automatically adjusted.

All dialog items have *dialog names* (for labels, the dialog name is optional). These dialog names are variable names and message names used by the block's code to interact with the dialog, as shown in "Dialogs" on page P25. Some dialog items have *titles* or text that appear in the dialog.

You can move an item in a dialog window or dialog tab simply by selecting it and dragging it to a new location within the same window or tab. You can resize many items in a dialog window by selecting the item and dragging its handle (the black square in the lower right corner of the item). Extend places dialog items in a grid; to position them off the grid, hold down the Option (*Macintosh)* or Alt (*Windows*) key as you drag. You can select multiple dialog items at once by holding down the Shift key or dragging a frame around them.

To add a new item to a dialog window or to a tab within that window:

- If your dialog has tabs, bring the tab you want to work on to the front.
- Choose New Dialog Item from the Define menu.
- Choose the type of item you want from the dialog.
- Drag the item to the desired location.

Modifying and creating dialogs, as well as ModL code interaction with dialogs, are described in more detail in "Dialog" on page P45.

ModL code

The block's ModL code is in the lower right pane of the structure window. You can edit and view in this pane like you can in other programs. Scroll through the code to get a feeling for how it looks.

If you are familiar with the C programming language, you are probably very comfortable with the ModL code. ModL is Extend's internal programming language. When you build blocks in Extend using ModL, the block's program is compiled to native machine code. ModL is essentially C with a few extensions and changes. If you are not familiar with C but know some other programming language such as BASIC or FORTRAN, you can probably follow the general logic of the ModL code and get a feeling for how it works. If you don't know any programming, the ModL code probably looks fairly foreign to you. You can skip the next few paragraphs and go straight to something more understandable, namely icons.

If you know a programming language, you can probably see the structure of the ModL code. The first lines are the declaration of the types of variables used in the code. The next few lines, the ones that begin with "//" or "**", are comments. After the comments are lines of programming code that are grouped into sections. Each section is either a *message handler* or a procedure. Message handlers begin with a line "on *xxx*". Procedures begin either with "*procedure xxx*" or "*type xxx*." For example, a message handler you see in every block begins "on Simulate" and the message handler code starts and ends with curly braces ({ and }).

The "on *xxx*" message handlers tell Extend what to do in various circumstances. For instance, the lines in the "on Simulate" message handler are executed for every step in the simulation. The lines in the "on InitSim" message handler are only executed once, at the beginning of the simulation. When you create your own block, you can add message handlers that are executed at defined times, such as when the dialog for the block is opened (so you can initialize the dialog's contents), when the simulation is stopped, when a dialog button was clicked, or when the block gets a message from another block.

ModL code and it's structure are discussed in much more detail in Chapter 2: ModL Code Overview and Block Creation.

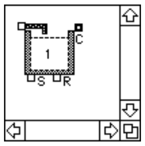
Icons

A block's icon is the most apparent aspect of a block since it appears in the model window. You can see the icon in the upper left pane of the structure window. The icon consists of a drawing or group of drawn objects and the block's connectors.

There are two ways to draw an icon: with Extend's drawing tools or by pasting drawings from the Clipboard. Extend's toolbar provides a drawing environment for those who do not have a paint or drawing program. If you have one of those programs, you might want to use it to draw or edit your icon, then paste it into the icon pane.

Use the same tools for selecting objects in the icon pane as you do in your model. Extend automatically gives you a grid for placing objects. If you want to use fine placement for your objects, hold down the Option (*Macintosh)* or Alt (*Windows*) key before selecting the object to override the grid as you move objects in the pane.

The icon pane looks like:



Icon pane

Animation

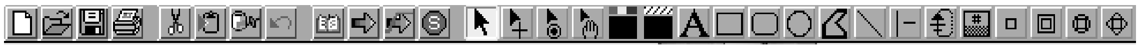
There are several ways to animate a block. Typically, the block's icon is animated, however it is also possible to animate outside of the icon. Blocks can show, hide, and change the patterns and colors of text and shapes; move a shape and increase or decrease its size; show a changing level; show a picture or a movie; or move a picture along connection lines between two blocks.

Many blocks have built-in animation that shows on or near the block's icon. In the "Icon pane" picture above, the white square at the center of the icon (labeled with the number 1) is an *animation object*. Animation objects are added in the icon pane of the structure window. The ModL code interacts with this object to show animation as the simulation runs. For the Accumulate block, the animation shows the level of the block.

In "Adding animation" on page P53, you will see how to add animation objects to a block you build. Also see "Animation" in the Extend User's Guide for an overview of animation, including block-to-block animation that flows along the connections in discrete event models.

Connectors

Most blocks have connectors. Connectors are added to the icon in the icon pane using tools from the toolbar. When the structure window is the front-most window, the toolbar changes from the standard toolbar and shows connector tools as well as an animation object tool:


Structure window toolbar

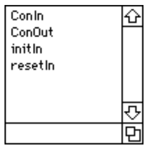
Adding connectors to the icon

At the far right of the toolbar, there are four tools for adding connectors: value (▫), item (▣), universal (⊚), and diamond(◈). The main Extend User's Guide describes the use of the value, item, and universal connectors. The diamond connector does not have any "special" properties, and is provided for your convenience. For example, if you are designing a new set of blocks and want to be sure that users only connect those blocks to each other, you can use the diamond connector. If the user tries to connect a diamond connector to a value or item connector, Extend will not let them. To add a connector, click on one of these tools, then click in the icon pane at the desired position.

The animation object tool, ▦, adds animation objects, as discussed above.

Naming connectors

Every connector has a unique name. In addition, the name of the connector defines whether it is an input or output connector: input connectors must end in "In" or "in" and output connectors must end in "Out" or "out". Connector names are shown in the connector pane. For example, the names of the connectors in the Accumulate block are:


ConIn
ConOut
initIn
resetIn

Connector pane

You can change the name to whatever you want, but the name must end in "In" or "Out". To change the name of a connector:

- Select the connector name in the connector pane. Extend highlights that connector in the icon pane so you can identify it.
- Type a new name or edit the name.
- Press the Enter or Return key or click anywhere else in the connector pane to save the edited name.

When you add connectors to the icon, they are all initially input connectors. To make one of these connectors an output connector, change its name to something that ends with “Out”.

Changing connector types

If you selected the wrong type of connector (such as a value connector when you wanted an item connector), you can easily change its type:

- Select the connector on the icon pane by clicking on it.
- Click on the correct connector type in the toolbar.

Note When you build blocks it is important that you use this method and change connector types, rather than deleting a connector. If you delete a connector, the order of the connections will be disrupted.

Help text

You can change the help text in the upper right pane to anything you want. Simply edit the text in the window. You can also add formatting to the text in the help window by selecting it and giving commands from the Text menu. The help text appears when you click the Help button in the lower left corner of the dialog and is also available in the Help command from Help menu.

Chapter 2: ModL Code Overview and Block Creation

A description of the steps for creating ModL code and putting together your own blocks

*"I must create a system,
or be enslaved by another man's."
— William Blake*

This chapter and the two that follow it are only for people who want to create their own blocks or who want to change existing blocks. Note that Chapters 1 through 4 only describe standard blocks, not hierarchical blocks, which are described in the main Extend User's Guide.

As you saw in Chapter 1: Parts of a Block, there are many parts of a block, the most complex of which is the ModL code. The first part of this chapter shows you how a blocks code is laid out and the basic ways that the code lets blocks interact with other blocks, with their own dialogs, and with the general parameters of a simulation. The second part shows you how to build a new block with a custom dialog and animation.

Language overview

A block's code is written in ModL, a language that is much like the popular C programming language. In order to create or modify ModL code, you should have a basic understanding of C. Imagine That! recommends the following:

- **If you have some familiarity with C**: You can certainly program in ModL. ModL uses many concepts from C. You do not need to know much C in order to be completely comfortable in ModL. The table below lists major differences between ModL and C.
- **If you program in some programming language other than C**: ModL will not present too much of a challenge if you feel comfortable with programming. To get the most out of ModL, you should learn C. Learning C after learning another language such as BASIC is fairly easy. There are dozens of good books on the C language, and even small bookstores and libraries often have a good selection of beginning C books. Two introductory books with strong followings are *Learn C Now* by Augie Hansen (Microsoft Press) and *C Primer Plus* by Mitch Waite and Stephen Prata (Sams). The table below compares common constructs for ModL and for other common languages.
- **If you do not program**: You may want to skip these chapters. You can make some modifications to existing blocks and build simple blocks without knowing programming. However, to get the full benefit from the ModL language, you should know programming. It is, unfortunately, very difficult to learn programming from a manual such as this. If you want to learn to program, you should probably learn C first since it has become the language of choice for most programmers today. Once you learn the basics of C, you can then use ModL.

As you read this chapter, you should probably be looking at the code of a few of the blocks that come with Extend.

ModL and other languages

ModL is very much like C, and C has many features that are similar to other programming languages. The following tables compare ModL and C and show how constructs in ModL compare to constructs in Pascal, BASIC, and FORTRAN.

Note XCMDs, XFCNs, and DLLs provide a method for incorporating other technologies, such as Visual Basic or Visual C++, into Extend. For more information, see "XCMDs and XFCNs" on page P237 or "DLLs" on page P238.

ModL compared to C

There are only a few differences between ModL and C. They are:

ModL	**C**
case insensitive	case sensitive
real or double (*Macintosh 68K:* 20 significant digits; *PowerPC and* *Windows*: 16 significant digits)	double
integer or long (32 bit)	long

string or Str255 (255 characters maximum)	typedef struct string
	{
	unsigned char length;
	unsigned char str[255];
	} string;

str15 (15 characters maximum)	typedef struct string
	{
	unsigned char length;
	unsigned char str[15];
	} string;

str31 (31 characters maximum)	typedef struct string
	{
	unsigned char length;
	unsigned char str[31];
	} string;

// comments to end of line only ** comments also supported	/* comments can be multi-line */
Array bounds subscript checking produces error messages when array bounds are exceeded	No array bounds checking
Functions declared using ANSI declaration only (prototype declarations).	Functions can be declared either K&R or ANSI

ModL	**C**
"i++;" is a statement. It cannot be used as an expression. See below.	"i++" is a statement which can be used as an expression
Statements cannot be used as expressions. For example "a[++i] = 5;" or "a[i=i+1] = 5;" are not allowed. They must be "i++;" or "i=i+1;" and "a[i] = 5;".	Expressions can be statements
The ModL "for" statement is: for (statement; boolean; statement)	The C "for" statement is: for (expression; expression; expression)
^ is used as the exponentiation operator (like it is in BASIC and spreadsheets)	^ is used as the exclusive-OR operator in logical expressions
To concatenate a string use: stringVar = stringVar+"abc";	The C equivalent is: strcat(stringVar, "abc")
To convert a number to a string: stringVar = x;	The C equivalent is: ftoa(x, str);
if (stringVar < "abc") ...	if (strcmp(stringVar, "abc") < 0) ...
Resizable dynamic arrays	Pointers
Structures not supported. Use dynamic arrays that have other dynamic arrays as elements (arrays of arrays)	Structures
!= or <> are not-equal operators	!= is the not-equal operator
% or MOD are the modulo operators	% is the modulo operator
Bit handling done by functions (such as BitAnd(n, m))	Bit handling done by operators (such as n&m)

ModL compared to Pascal, BASIC, and FORTRAN

XCMDs, XFCNs, and DLLs provide a method for linking between Extend and languages other than ModL. For more information, see "XCMDs and XFCNs" on page P237 or "DLLs" on page P238.

The following table compares some of the common constructs in ModL to other languages. (Due to space limitations, some lines in the BASIC and FORTRAN code fragments are wrapped incorrectly.)

ModL	Pascal	BASIC	FORTRAN

real a[10], x;
integer i;
 	
for (i=0; i<10; i++)
{
if (i == 5)
x = 3;
else
x = 5+i;
a[i] = x;
}
	a: array[0..9] of real;
	x: real;
	i: integer;
	
	FOR i:=0 TO 9 DO
	BEGIN
	IF i = 5
	THEN
	x := 3;
	ELSE
	x := 5+i;
	a[i] := x;
	END
		DIM A(9)
		
		FOR I=0 to 9
		IF I= 5
		THEN
		x = 3
		ELSE
		x = 5+I
		A(I) = x
		NEXT I
			REAL A(0:9), X
			INTEGER I
			
			DO I=0,9
			IF (I .EQ. 5)
			THEN
			X = 3
			ELSE
			X= 5+i
			END IF
			A(I) = X
			END DO

integer b[10][5];	b: array[0..9,0..4]of integer;	DIM B(9,4)	INTEGER B(0:9,0:4)

switch (i)
{
case 0:
x = 3;
break;
case 1:
case 2:
x = 5+i;
break;
default:
x = 0;
break;
}
	CASE i OF
	0:
	BEGIN
	x = 3;
	END
	1,2:
	BEGIN
	x = 5+i;
	END
	ELSE
	BEGIN
	x = 0;
	END
		ON I+1 GOTO
		10,20,20
		x = 0
		GOTO 100
		10 x = 3
		GOTO 100
		20 x = 5+I
		100 .....
			GOTO (10,20,20)
			I+1
			x = 0
			GOTO 100
			10 x = 3
			GOTO 100
			20 x = 5+I
			100 .....

while (i < 10)
{
x = x+i;
i = i+1;
}
	WHILE i < 10 DO
	BEGIN
	x := x+i;
	i := i+1;
	END
		WHILE I< 10
		x = x+I;
		I= I+1;
		WEND
			DO WHILE (i .LT.10)
			x = x+I;
			I = I+1;
			END DO

ModL	Pascal	BASIC	FORTRAN

do
  {
  x = x+i;
  i = i+1;
  }
while (i < 10);
	REPEAT
	  x := x+i;
	  i := i+1;
	UNTIL i >= 10
		10 x = x+I;
		  I = I+1;
		IF I < 10 GOTO 10
			10 x = x+I;
			  I = I+1;
			IF (I .LT .10)GOTO10

//comments ** comments (NOTE: only to the end of the current line)	{comments enclosed in braces}	REM comments or ' comments (NOTE: only to the end of the current line)	* comments or C comments (NOTE: only to the end of the current line)
strng = strng+"abc";	CONCAT(strng, "abc");	strng = strng+"abc"	strng = strng // 'abc'
strng = x;	ftoa(x, strng);	strng = STR$(x)	implementation dependent

if (strng < "abc")
  ...
	IF strng < 'abc'
	  THEN
	  ...
		IF strng < 'abc'
		  THEN
		  ...
			IF strng .LT. 'abc'
			  THEN
			  ...

Language terminology

The language used by computer scientists has changed over the last 40 years. The following describes the terms used in ModL coding.

Term	Description
array	An indexed list of numbers.
constant	Value that does not change.
data type	The type of storage used for the data: real, integer, or string
E notation or scientific notation	Exponential number specified as a number raised to a power of 10. For example, “6.3E3” means 6,300 and “5E-1” means 0.5.
function	Procedure which returns a value.
global variables	Variables that can be viewed or modified by any block. These variables’ values are preserved between simulation runs but are not stored with the model when saved to disk. They are all predefined by Extend.
identifier	Name that is typed in.
literal	Number or string that is typed in as a constant.

Term	Description
local variable	Variable that is valid only within the message handler or user-defined function in which it is defined. Note that these are temporary variables and their values are not preserved after exiting a message handler or function. Use with caution because local variables with the same names as static variables override the static variables.
message handler	Grouping of code that tells Extend what to do in a particular circumstance which is defined by the message.
procedure	Predefined named group of code instructions with specified arguments. A procedure can be called in a block's code. Does not return a value.
statement	Section of code ending with ";".
static variable	Variable that is valid throughout the block's code in which it is defined. The values for these variables are preserved and are stored with the model when saved to disk.
system variable	Variable that is valid in any block in a model. These variables are declared by Extend and can be viewed or modified by any block.
type declaration	Defining a variable as a certain data type.

Layout of a block's ModL code

Like C programs, ModL code starts with type declarations and constant definitions. Because you declare these at the beginning of the code, before any message handlers or user-defined functions, they are considered *static* or permanent. They are therefore valid throughout the block's code unless overridden by a local variable declaration.

After the type declarations, there are procedure and function definitions, and many *message handlers*. The procedures, functions, and message handlers are just definitions: they need to be called in order to be executed. Message handlers interpret messages that come from the simulation or from you. Extend runs the message handler whenever one of the messages is passed to the block. You declare local variables in message handlers, functions, and procedures.

Like C, ModL ignores blank lines and indentation. Of course, it is a good idea to indent your code with Tab characters to make them easier to read.

One small difference between ModL and C is that comments are text that is preceded by "//" or "**". Anything from the two slashes or two asterisks to the end of the line is ignored by the ModL compiler. Another small difference is that ModL is not case-sensitive. See "ModL compared to C" on page P19 for a listing of the differences.

Type declarations and constant definitions

There are five data types in ModL: 1) real or double, 2) integer or long, 3) str15, 4) str31, and 5) string or Str255. For the data types with multiple identifiers, the identifiers are interchangeable; this manual uses *real*, *integer*, *str15*, *str31* and *string*. The type declarations and constant definitions are set up like they are in C. For example, some typical type declarations might be:

```
real nextTime;
real dataArray[], averages[10], theMatrix[10][10];
str15 str15Array[];
str31 str31Array[];
string strArray[], errorStrings[6], theError;
integer checkUsed;
```

(As you can see from the example above, ModL also has fixed and resizable arrays.)

ModL already has some pre-defined variables that can be treated just like other static variables. These system variables are described in "System variables" on page P80.

A constant definition might be:

```
constant maxPower is 52.5;
```

Structures, data types, declarations, and definitions are described in detail in Chapter 3: The ModL Language.

Functions, procedures, and message handlers

ModL code has functions, procedures, and message handlers that group the code into sections. Message handlers are denoted by:

```
on messagename
{
     one or more statements;
}
```

The statements in the body of the message handler are in the same format as C procedures. For example, a simple message handler is:

```
on CreateBlock   // The user added this block to a model
                 // from the Library menu
{
     checkUsed = 1;  // static variable declared at top of the code
     myNumber = 1;   //Initial setting for dialog item
}
```

The statements in this message handler are executed when the block receives the "CreateBlock" message. The variable "checkUsed" is a static variable for the block, defined at the top of the code, as seen above. "myNumber" is the dialog name of a dialog parameter item for the block. This statement causes the parameter field to display 1 when the block's dialog is first opened.

Message handlers are discussed in detail in "Using message handlers" on page P31. ModL functions are listed and described in Chapter 3: The ModL Language.

Note You can declare local variables at the beginning of a message handler. However, you should use caution if you do this because a local variable is temporary and loses its value when the message handler is exited. Also, within each message handler, local variables can override static variables. (If a local variable is defined with the same name as a static variable, any references to that name within that routine or message handler will change or reference the local variable, and the static one will not be modified.)

Accessing connectors and dialog items from the block's code

Connectors

In Chapter 1: Parts of a Block, you saw that you give each connector a name that ends in "In" or "Out" when you create a block. Refer to "Naming connectors" on page P15 for more information about this. Connector names are used as variables in your ModL code. To read from an input connector, simply use its name in the right side of a statement. For instance, you can read from an input connector called "firstConIn" with:

```
myNumber = firstConIn;
```

To set the value of an output connector, simply assign it a value. To set the value of an output connector called "totalOut," you might use:

```
if (myNumber > 0)
     totalOut = 1.0;
```

Note that all connectors are real type variables. If you set a connector to an integer, Extend converts the integer to a real; this conversion (like all conversions) slows your simulations.

Arrays of multiple values can be passed through connectors. Passing arrays is an easy way to pass more than one piece of information at a time through blocks. Passing arrays is covered in more detail in "Passing arrays" on page P213.

Dialogs

In Chapter 1: Parts of a Block, you saw that a dialog item can be a parameter, editable text, check box, radio button, button, popup menu, data or text table, label, switch, slider, or meter. Refer to "Types of dialog items" on page P9 and "Working with dialog items" on page P12 for complete details.

The default dialog names (OK, Cancel, and Comments) as well as the dialog names for any dialog items added to a block, are listed in the Variables pane at the lower left of the structure window. You can copy names from that pane to use in your ModL code using the Copy and Paste commands.

To have a parameter or editable text item not be accessible from the dialog, use the Display Only option. To hide dialog items, for instance, if you no longer want them in the dialog, choose Hide Item. These options are described in "Display Only, Hide Item, and Visible In All Tabs options" on page P11.

You can read and set dialog items in the same way as you do connectors, using their *dialog names*. The dialog names for some items (check box, radio button, popup menu, switch, slider, and meter) can be used in the ModL code as both a variable name and a message name.

Parameters and editable text
Assume that your dialog has a parameter entry with the dialog name "numberOfRecords". You could have a statement such as:

```
myNumber = numberOfRecords/100;
```

You can also set dialog items from inside the ModL code. For instance, if you want to set the value shown in the "numberOfRecords" to "1000," you would use the statement:

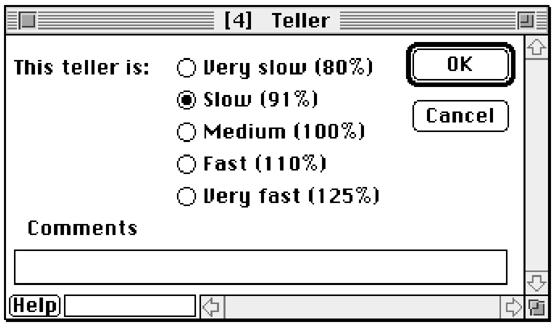
```
numberOfRecords = 1000;
```

The same methods work for editable text:

```
if (temp > 1500)
     displayHeat = "Hot";
     else displayHeat = "Cool";
```

When you define the parameter or editable text, you can choose the Display Only choice in the New Dialog Item dialog. If you do, the item cannot be changed by the user. It can only be changed in the block's code, using the same techniques you have just seen.

Check boxes and radio buttons
Check box and radio button dialog items return true/false values: true if the check box or radio button is selected, false if not. They also send the message name to the block when they are clicked. For example, imagine that you created your own block for a teller in a bank. Instead of entering a number, you want to set the speed using a radio button. Your dialog might look like:



Teller dialog (Macintosh)

The five speed radio buttons would have the dialog names VSlow, Slow, Med, Fast, and VFast. They must all be defined to be in the same radio group when they are created to make sure that only one of them can be selected. In this example, leave the group value at the default (group 0).

To set the variable "theDelay" based on which button was chosen, the code uses the statements:

```
if (VSlow)  theDelay = initDelay * 1.25;   // v slow is 1.25 times normal
if (Slow)   theDelay = initDelay * 1.1;    // slow is 1.1 times normal
if (Med)    theDelay = initDelay * 1;      // medium is 1 time normal
if (Fast)   theDelay = initDelay * .91;    // fast is 0.91 times normal
if (VFast)  theDelay = initDelay * .8;     // v fast is 0.8 times normal
```

The "if" statements are executed only if that button value is TRUE (non-zero); of course, only one of them can be true because they are all in the same radio button group. You could also structure the checking with five message handlers, such as:

```
on VSlow     // VSlow radio button was clicked
{
     theDelay = initDelay * 1.25;
}
```

Note that in the first instance, the "if" statements would be executed during the simulation, usually in the InitSim message handler. In the second instance, the VSlow button message handler would be executed when you clicked the button labeled "Very slow".

Similarly, to specify that, when a block is placed in the model, the "Medium" button should be selected, the CreateBlock message handler contains:

```
Med = TRUE;
```

This also sets the other radio buttons in the group to FALSE.

Note When setting the state of a radio button group in your ModL code, you should always explicitly state which button is set to TRUE. Setting a radio button to TRUE will automatically set the remaining radio buttons in the group to FALSE. However, setting a radio button to FALSE will not affect the state of the other radio buttons in the group. It is therefore possible to have a condition in which all radio buttons are set to FALSE. In most cases, this would be an error condition.

Buttons

When buttons are clicked, they send a message to the ModL code. The most common buttons in your block are OK and Cancel. The OK and Cancel buttons are handled automatically. If you add other buttons, such as shown later in this chapter, you add your own message handler for those buttons.

You can change the title of a button by setting the button's message name to some text in your code. Note that the read value of the button's title is always the value typed in when creating the dialog item, even if it is changed by setting it to a different text value in the ModL code. If you change the title of the button, your code must reset the title when the user opens the block. You do this using the On DialogOpen message handler. See also "Changing dialog text in response to a user's action" on page P180.

Data tables and text tables

A data table or text table in a dialog represents a two-dimensional array of real numbers or text. Tables have an interface that allows you to type in any input and also, like all dialog items, allows you to display values generated in the code.

The following code fragment shows how to read and write to a data table called "dataTable" that has 4 rows and 3 columns. Data tables are treated as arrays, which are discussed in detail in "Arrays" on page P66. As in the C language, array subscripts in arrays start at 0, not 1.

```
. . .
// Read the first row, second column cell into myValue.
myValue = dataTable[0][1];
. . .
// Set the fourth row, third column cell to myValue
dataTable[3][2] = myValue;
```

The text table allows you to type in text, numbers, or both. Since all entries are strings, if you want to use the numbers in your ModL code, you must convert them to real values using the StrToReal function.

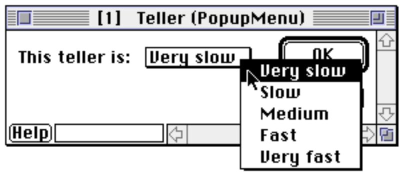
Static text (label)

Static Text is text that appears in a dialog. This is regular text, not text that appears in an editable text box. You can give static text a dialog name that can then be used in the ModL code to change the text of the label. Note that the read value of static text is always the value typed in when creating the dialog item, even if it is changed by setting it to a different text value in the code. If you change the text of the label, your code must reset the text when the user opens the block. You do this using the On DialogOpen message handler. See also "Changing dialog text in response to a user's action" on page P180.

Popup menu items

When an item in a popup menu is selected, the menu's dialog name returns a value which is the integer corresponding to the position of the item in the menu, where 1 is the value for the first item in the list. For example, in a 5-item menu, the values of its dialog name will be set to 1, 2, 3, 4, or 5 based on which menu item the user chooses.

Popup menus replace series of radio buttons. For instance, instead of using a several radio buttons to represent teller speed, as shown in "Check boxes and radio buttons" on page 239, you could use a popup menu. In this case, your dialog would look like:



Teller dialog with popup menu (Macintosh)

This popup menu has the dialog name "MyMenu". The five menu items have the titles as shown above. Since "Very slow" is selected, MyMenu is set to 1. If "Medium" were selected, MyMenu would be set to 3.

To set the variable "theDelay" based on which menu item is selected, the code in the InitSim message handler has these statements:

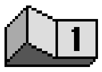
```
if(MyMenu == 1) theDelay = initDelay * 1.25; // v slow is 125% times normal
if(MyMenu == 2) theDelay = initDelay * 1.1; // slow is 110% times normal
if(MyMenu == 3) theDelay = initDelay * 1; // medium is normal speed
if(MyMenu == 4) theDelay = initDelay * .91;// fast is 91% of normal
if(MyMenu == 5) theDelay = initDelay * .8;// very fast is 80% of normal
```

To specify that, when a block is placed in the model, the "Medium" menu item should be selected, the CreateBlock message handler contains:

```
MyMenu = 3; // defaults to the "Medium" menu item, which is the third in the list
```

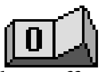
You can use the popup menu's dialog name (for example, *MyMenu*) as the message handler name (for example, *On MyMenu*) to perform specific actions when the user selects a menu item. For instance, you could use this to report errors to the user, show or hide other dialog items, or cause a sound to play.

Switch
A switch looks like a standard light switch:



Switch

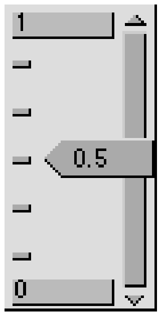
When you click on the side that is not down, it changes to the other value, makes a small clicking sound, and sends a message to the ModL code.



Switch in off position

The switch dialog name always returns either 0 or 1, depending on the value of the switch. You can also control the switch by setting its variable name to 0 or 1 in the code.

Slider
A slider allows you to enter a value by dragging its knob along the length of the slider, similar to the volume knobs on some stereos and televisions. It looks like:



Typical slider

You set the minimum and maximum values from the ModL code or by clicking and editing the minimum and maximum values. The current value can be set from the code and can also be set as the model runs by dragging the slider up or down. In this way, you can use the slider both for visual output and for input.

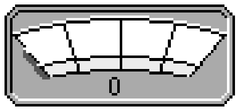
The slider is represented in a three-element array of reals that represent the minimum, maximum, and current values. Thus, if your slider is called "theSlider", you might initialize it with the lines:

```
theSlider[0]=0.0;      // Minimum of 0
theSlider[1]=10.0;     // Maximum of 10
theSlider[2]=3.33;     // Starting value of 3.33
```

As the model runs, you can check the value that the slider is at by reading the third array element, such as:

```
theSetting = theSlider[2];
```

Meter
A meter allows you to view a value on a meter. It looks like:



Typical meter

You set the minimum, maximum, and current values from the ModL code.

The meter is represented in a three-element array of reals that represent the minimum, maximum, and current values. Thus, if your meter is called "theMeter", you might initialize it with the lines:

```
theMeter[0] = 0.0;      // Minimum of 0
theMeter[1] = 30.0;     // Maximum of 30
theMeter[2] = 10.0;     // Starting value of 10
```

As the model runs, you set the value shown on the meter in the third array element, such as:

```
theMeter[2] = theSetting;
```

Embedded Object (*Windows* only)
An embedded object, when first placed in a dialog, looks like:

Empty embedded object

An embedded object is a placeholder for an Active-X object which can be manually inserted by the user or inserted by the ModL code of the block. Please see “Embedding OLE Objects and ActiveX Controls” in the main Extend User’s Guide.

Using message handlers

You are probably wondering what the messages that trigger the message handlers are. Extend has many messages, but only a few of them are used in most block’s code. There are two types of messages: *system messages* (which includes *connector messages* and *user defined messages*), and *dialog messages*. System messages are sent to blocks by the Extend engine; dialog messages are the names of dialog items and are sent to the block when the user interacts with the block’s dialog.

System messages

System messages are sent to blocks by Extend when you build models, run a simulation, or use ModL function calls that explicitly send system messages to other blocks. For instance, when you add a block to a model, Extend sends the “CreateBlock” message to the new block. If there is a message handler for the “CreateBlock” message (that is, a bracketed set of lines that are preceded by “on CreateBlock”), it is executed; if not, nothing happens. See the complete list of message handlers in“Message handlers” on page P75

System messages include connector messages and user-defined messages:

- Connector messages are sent by ModL function calls that send messages to connected blocks. See ConnectorName in the table below for more information.
- User defined messages can be sent to any blocks in the model (connected or unconnected) by calling ModL functions in your code. See BlockReceive in the table below for more information.

When you run a simulation, some system messages are sent to all blocks; others are sent only to a specific block. For instance, the "InitSim" message, which tells the blocks that a simulation is starting, is sent to all blocks. The "CreateBlock" message is sent only to the block that was added to the model. All ModL system messages are listed in the Variables pane at the lower left of the structure window. You can copy message names from that pane to use in your ModL code with the Copy and Paste commands or drag and drop editing.

System messages sent to all blocks during a simulation run
The system messages that are sent to all blocks are sent in the following order (See the complete list of message handlers in"Message handlers" on page P75):

Message	When sent
CheckData	After the Run Simulation command and before the simulation begins. This is the best place to check whether all data that is to be used in the block is valid. If the data is bad, you should execute an "abort" statement, so that Extend will select the block and alert you that the data is missing or bad. During this message, connectors that are connected to something else in the model always have a *true* value and unconnected ones have a *false* value. This makes it easy to check whether a block is connected properly before running a simulation.
StepSize	After all CheckData messages. If your ModL code has to have a particular StepSize, you should set it here. In continuous models, set the DeltaTime system variable to your maximum tolerable step size. Extend queries all blocks and uses the smallest DeltaTime value returned. This message is also used to setup the attribute global arrays in a discrete event model.
InitSim	Just before the simulation starts. If your block's code uses DeltaTime (continuous model only), you should check it here. Also, allocate any arrays that are dependent on DeltaTime, NumSteps, or any other system variable. This is also a good time to set any static variables, dialog items, or connectors that change when the simulation starts.
Simulate	Every step of the simulation. Note that this message gets sent over and over, not just once. This is where most of the "action" in a block takes place. For instance, you check and change the value of connectors in this message handler. *In continuous simulations*, the first Simulate message gets sent with CurrentTime=StartTime and CurrentStep=0. For subsequent Simulate messages, Extend adds 1 to CurrentStep and adds DeltaTime to CurrentTime. The last Simulate message occurs when CurrentStep=NumSteps-1 and CurrentTime=EndTime, so each block gets NumSteps Simulate messages. *In discrete event simulations*, the first Simulate message gets sent with CurrentTime = StartTime and CurrentStep = 0. For subsequent Simulate messages, the Executive block controls how CurrentTime is advanced.
FinalCalc	Sent after the Simulate messages are over and before the BlockReport messages. Used for any final calculations that the block might need before the BlockReport and EndSim messages.
BlockReport	Sent after the FinalCalc messages. If a block receives this message, it has been selected for a report. To organize the reports by block type (see "Block types" on page P56), Extend cycles through each block type and sends this message to any block selected for a report.
EndSim	At the end of the simulation. Use this to clean up any memory that you used or to reset values. This message gets sent even if you or the block's code aborts the simulation.

System messages sent to individual blocks

The system messages that are sent to an individual block on particular events are (See the complete list of message handlers in“Message handlers” on page P75):

Message	When sent
AbortDialog-Message	If the user stops the execution of one of your own dialog message handlers or that message handler executes an ABORT statement, this message gets sent to the block. Use this as an “exception handler” to clean up after an error occurs.
AbortSim	After any premature stopping of a run, such as when you stop a simulation using the Stop button in the Status bar or when an error occurs before the simulation is finished. A good use for the AbortSim message is to reset values that may be incorrect due to incomplete calculation. Note that EndSim always gets executed after AbortSim.
ActivateModel	Sent to all blocks in a model when that model is brought in front of a different model. This allows blocks to modify their data or appearance if the model is activated.
AnimationStatus	Sent to all blocks in a model when the Animation menu item changes state. This is useful to change a block's appearance according to the value of the AnimationOn variable.
AdviseReceive	Sent to the block when it receives updated data from an advise conversation (see “Interprocess Communication (IPC)” on page P102).
BlockClick	Sent when the user clicks on a block. Use GetMouseX(), GetMouseY(), and GetBlockPosition() to find out what portion of the block was clicked. See the Mandelbrot model (*Macintosh:* Mandelbrot; *Windows*: mandlbrt.mox).
BlockLabel	Sent to the block that just had its Block Label changed, when the label becomes deactivated. The block can then use the new value of the label.
BlockMove	Sent to all blocks that have been moved upon the completion of that move. (See “Scripting” on page P157)
BlockRead	This message must be used only with extreme caution. It can cause a crash if used improperly. It is used to convert version 3.x block data tables to version 4.0 dynamic data tables. See the Information block. Call only the GetFileReadVersion() and ResizeDTDuringRead() functions in this message handler.
BlockReceive0 through 4	Custom defined message. These are used by the Discrete Event library as system messages. We suggest you use “UserMsg0 through 9” when you define a message for your own purpose.
CellAccept	Sent to a block when the user finishes editing any cell in any of the block’s data tables. You can use this to check the data entered in cells in a data table.

Message	**When sent**
ClearStatistics	When a block's statistical variables need to be reset. This message is typically sent by a block in the Statistics Library. See the Activity, Delay block in the Discrete Event library for an example of receiving this message.
Connection-Make	Sent to all newly connected blocks when the user makes a connection on the model. For hierarchical blocks, this is sent to all internal blocks. To prevent the connection from being completed, you can call the *Abort* statement.
Connector-Name	Connector message. When the named connector on a connected block receives a message from another block using the connector message functions.
CopyBlock	Sent to all blocks selected before a Copy operation. Useful to dispose of a dynamic array, create a dynamic array, or add a Global Array to the clipboard using the GAClipboard() function, before the copy.
CreateBlock	When the block is added to a model. This is the place to set initial values for the dialog. Note that this message is only sent to the block when you add it to a model. If you change the CreateBlock code for a block that already is in a model, the changes won't affect the existing blocks, only new blocks added to the model.
DeleteBlock	Sent when the user deletes a block from the model. For hierarchical blocks, this is sent to all internal blocks.
DialogClick	Sent when the user clicks on a dialog item, before the actual dialog item message is sent to the block. Call WhichDialogItemClicked() to find the name of the item that was clicked. This message is used, for example, to modify the items in a popup menu at the time it is clicked on but before it opens to the user.
DialogClose	When the OK or Cancel buttons or the close box are clicked.
DialogOpen	When the block's dialog is opened. If you want to display any static text labels that can change based on what is happening in the simulation, set them here. If you don't want the dialog to open, execute an ABORT statement (see Plotter I/O block code).
DragCloneToB-lock	This message gets sent when a user drags a clone onto a block and releases the mouse button when the block is highlighted. If you want to get information about the clones, call the GetDraggedCloneList() function. Used in the Evolutionary Optimizer block.
HBlockClose	When a hierarchical block's submodel or structure is closed. For hierarchical blocks, this is sent to all internal blocks.
HBlockHelp-Button	This message is sent to all the blocks inside an Hblock when the Hblock help button is clicked. This can be used to intercept the help button message for the hBlock, and do something of your own implementation instead. Aborting this message will prevent the message from getting to the rest of the blocks in the hBlock, and prevent the help text from opening up.

Message	When sent
HBlockOpen	When a hierarchical block is opened, this message is sent to all blocks in its submodel.You can use this to correct animation in a hierarchical block that is opening. If you don't want the hierarchical layout or structure windows to open, execute an ABORT statement in the on HBlockOpen message handler in one of the blocks in the submodel.
HelpButton	Sent when the user clicks the Help button on the lower left of the block's dialog. Executing an ABORT statement during this message handler will prevent the Extend Help from opening.
MailSlotReceive	This message is sent repeatedly when there are mailslot messages waiting to be picked up. See the mailslot functions later in this documentation for more information.
ModelSave	This message is sent to each block at the beginning of a save. You might use this message handler to dispose of unneeded data before it gets saved.
OpenModel	When a model is opened or to a new hierarchical block that has just been placed on the model worksheet. You can use this message handler to set some block variables to values that you only want to reset when a model is opened, not at the beginning of a simulation.
PasteBlock	When the user pastes a block onto the model. For hierarchical blocks, this is sent to all internal blocks.
PauseSimulation	Sent to ALL blocks when the user pauses the simulation. Contrast this to ResumeSim which is sent only to the blocks that have been changed by the user.
Plotter0Close, Plotter1Close, Plotter2Close, Plotter3Close	Sent when the user closes a plotter window.
ProofAnimation	This message is used by the blocks that are interacting with Proof to produce proof animation functionality. Please see "Proof Animation (Windows only)" on page E278 for more details on adding Proof animation to your models.
ResumeSim	When the user either: • clicks dialog buttons during the simulation, or • edits a parameter and chooses Resume from the Run menu or clicks the Resume button in the simulation status bar. This allows the ModL code to respond to changes before continuing the simulation. Note: This message is sent only to the blocks that have had dialog values edited.

Message	When sent
ShiftSchedule	Sent by a block to all the blocks in the model when a shift schedule has been changed.
TabSwitch	This message is sent to a block when the block's dialog is switched from one tab to another. This will also be sent when the dialog is first opened, as that is basically treated as a click on the last tab that was previously opened.
TimerTick	Sent by the Timer chore started by the StartTimer function. (See scripting functions below.)
UpdateStatistics	When a block's statistical variables need to be recalculated and updated. This message is typically sent by a block in the Statistics Library. See the Activity, Delay block in the Discrete Event library for an example of receiving this message.
UserMsg0 through 9	User defined message. When a block (connected or unconnected) receives a message from another block using the SendMsgToBlock function.

Dialog messages

Dialog messages are the names of dialog items and are sent to the block when you click controls (i.e. buttons) or unselect parameters. Dialog messages work in a similar manner to system messages. For example, assume a block has a "Count" button. When you click on that button in the block's dialog, Extend sends the "Count" message to the block. If the block has an "on Count" message handler, it will be executed; if not, nothing happens.

When you click on a button in a dialog or unselect a parameter, Extend sends a message with the same name as the item to your ModL code. Dialog names for all dialog items in a block are listed in the Variables pane at the lower left of the structure window. The dialog messages are (See the complete list of message handlers in"Message handlers" on page P75):

Message	When sent
OK	When you click the OK button. You do not need to do any special handling in this section unless you want to check input data.
Cancel	When you click the Cancel button.
YourButton	If you have created a button, radio button, or check box named *YourButton*, this message handler is activated when the button is clicked.
YourItem	Whenever the selection in a dialog is in a parameter or editable text dialog item and you click another item or press the Tab key (taking the selection out of the item), the message handler with that dialog item's name is invoked. This is useful if you want to check the value of the item after it might have been changed.

Creating a block

Chapter 1: Parts of a Block, showed you the parts of a block except for the ModL code. This section shows you how to create a block and have it perform useful tasks. Before you begin, you may want to review the code of some other simple blocks such as the Add and Subtract blocks in the Generic library (*Macintosh*: Generic Lib; *Windows*: generic.lix). For this example, you are going to create a block which converts miles to feet. The final block, called “Miles”, is located in the Custom Blocks library (*Macintosh*: Custom Blocks Lib; *Windows*: custom.lix).

Creating a new block

As you will see when you build the Miles block below, the steps to create a new block are:

- Choose Build New Block from the Define menu.
- In the dialog that appears, name the block and install it in a library. First, give the block a name. Then choose a library for installation. There are three ways to select a library. Either:
    - Select a library from the list of open libraries on the left **or**
    - Open an existing library that you have created (*not* an Extend library) **or**
    - Create a new library
- Once you have selected a library for installation, click Install In Selected Library

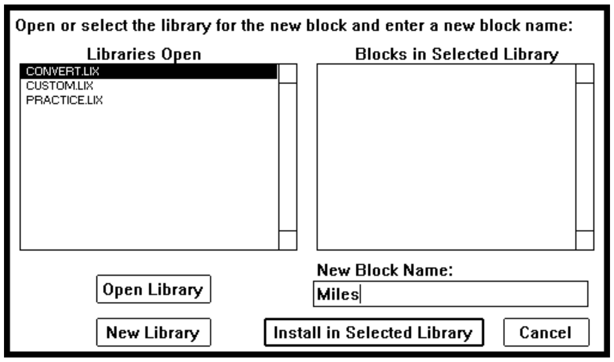
Building a block that converts miles to feet

In this first step, the Miles block will have two value connectors, an input and an output. The block will look at the value at the input connector (a number of miles), multiply it by 5280 (the number of feet in a mile), and put it out to the output connector.

To start, you should perform the simple tasks: create the dialog, an icon with connectors, and the help text. It is a good idea to create these first so that you know the names of the dialog items and connectors from the icon to use in your ModL code.

To create the Miles block:

▸ Choose Build New Block from the Define menu. The command prompts you for the library you want the block in and the name of the new block:


Open or select the library for the new block and enter a new block name:

Libraries Open	Blocks in Selected Library
CONVERT.LIX	
CUSTOM.LIX	
PRACTICE.LIX	

New Block Name: Miles

Dialog for naming a new block and installing it in a library (Windows)

▸ Click New Library.

▸ Name the library (*Macintosh*: “Convert Lib”; *Windows*: “convert.lix”).

▸ Click Save.

▸ Enter the name “Miles” for the new block, as shown above.

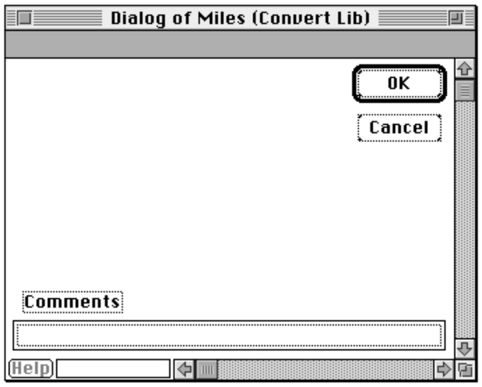
▸ Click Install in Selected Library.

Extend opens the default dialog and structure windows, as seen below.

Dialog of the Miles block

By default the dialog window opens behind the structure window. You can cause the dialog window to open in front by unselecting the *Structure windows opens in front* checkbox in the Programming tab of the Preferences command. For now, bring the dialog window to the front by clicking on it or by selecting it in the Window menu.

The dialog window is empty except for two buttons, a label, and an entry box.



Default dialog window (Macintosh)

Add some explanatory text to the left of the OK and Cancel buttons.

▸ With the dialog window in front, choose New Dialog Item from the Define menu.

▸ Click on "Static Text (Label)".

▸ Type `Converts miles to feet.` in the text box labeled "The text" (dialog names are optional for labels).

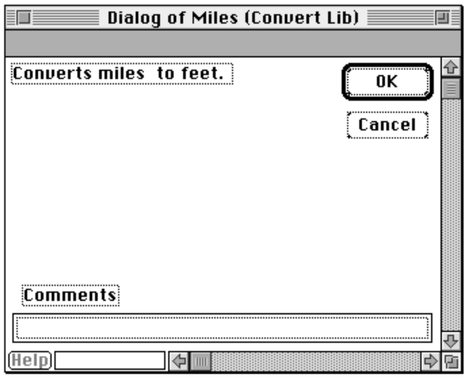
▸ Click OK.

▸ Resize this text box to show all the text.

Note that new dialog items are placed in the upper left corner of the dialog. You want to move this down a bit. Extend automatically puts dialog items on a grid. To drag an item without using the grid, hold down the Option (*Macintosh)* or Alt (*Windows*) key while you drag the item.

▸ Drag the text box to the new location, one grid step down, and resize if necessary.

Your dialog should now look like:

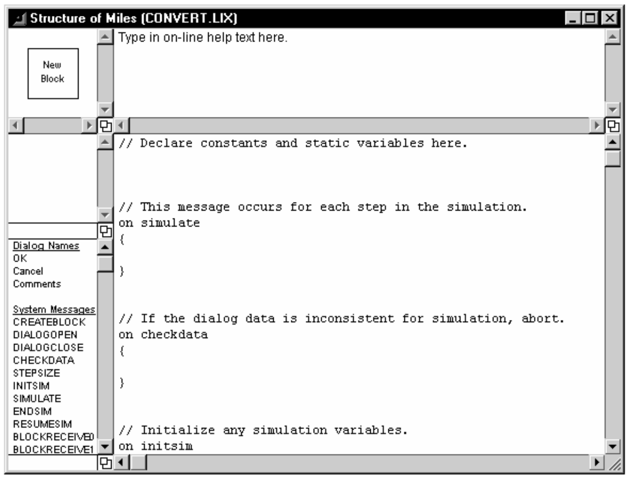


New dialog (Macintosh)

Note that dialogs can have much more information than this; you will see a much friendlier and more informative dialog later in this chapter.

Structure of the Miles block

The structure window has the default structure for the icon, the help, and three message handlers defined for you. Most blocks' code use these message handlers (Simulate, CheckData, and InitSim), so Extend puts them into the blank code for you.



Type in on-line help text here.

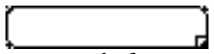
```
// Declare constants and static variables here.



// This message occurs for each step in the simulation.
on simulate
{

}


// If the dialog data is inconsistent for simulation, abort.
on checkdata
{

}


// Initialize any simulation variables.
on initsim
```

Default structure window (Windows)

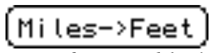
Icon

For this block, you only need a simple icon that has an input connector on the left and an output connector on the right.

- Delete the default rectangle and text in the icon pane by clicking on each and pressing the Delete or Backspace key.
- In the pattern swatch window, click the white color. Then use the rounded rectangle tool to make a small rectangle:

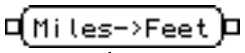

Rectangle for icon

- Select the Text tool from the tool bar, click over the rectangle, type in **`Miles->Feet`**, and press Enter (in the numeric keypad). Hold down the Option (*Macintosh)* or Alt (*Windows*) key to disable the automatic grid and drag the text so that it is centered.


Icon for new block

Connectors
You now want to add a connector to each end of the icon.

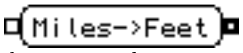
- Click on the value connector tool, ▫. in the toolbar.
- Click near the left side of the icon picture.
- Click on the connector tool again.
- Click near the right side of the icon picture.
- Select each connector and drag it so that you can make them touch the icon. You can use the Option (*Macintosh)* or Alt (*Windows*) key to remove the grid to get the connectors exactly where you want them.


Icon with connectors

When you add connectors to an icon, Extend adds input connectors with the names “Con1In,” “Con2In,” and so on. To make an input connector an output connector, you must change its name so that it ends in “Out”.

- In the Connector pane, click on “Con1In.”
- Type **`MilesIn`**, then click on “Con2In.” This changes the name of the input connector.
- With “Con2In” selected, type **`FeetOut`**.

▸ Press the Enter or Return key or click anywhere else in the window. This changes the second connector to an output connector, as you can see from the icon:



Icon with input and output connectors

Help text

Replace the default help text in the help pane with some simple help text. Even though there is not much to say, it is always a good idea to have help text since you may let other people use the block some time in the future.

ModL code

You are now ready to create the ModL code. As shown earlier in this chapter, the "action" in a block's code happens in the Simulate message handler. You can leave the other default message handlers blank since there is nothing in this block to check or initialize at the beginning of the simulation run. Type the ModL code into the code pane at the lower right of the structure window.

The code for this block is simple:

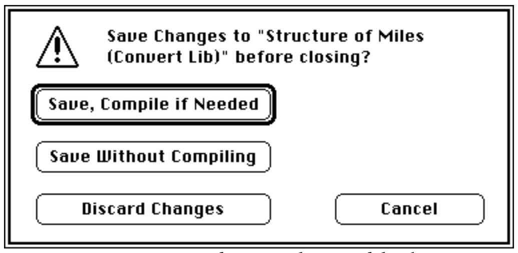
```
on Simulate
{
     FeetOut = MilesIn * 5280.0;
}
```

The spaces on each side of the operators are optional; the semicolon (;) at the end of the statement is not. Because ModL is not case sensitive, the connector name "FeetOut" could just as well have been written "feetout".

The code means that for each step of the simulation, Extend reads the value at the input connector, multiplies it by the real value 5280.0, and sets the output connector to that product.

Compiling and saving the block

After you finished writing your code, close the block by clicking on the close box in the upper left corner. Extend prompts:


Save Changes to "Structure of Miles (Convert Lib)" before closing?

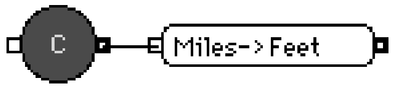
Saving and compiling a block

Click “Save, Compile If Needed” to compile the ModL code and save the block in the library. If there are any compilation errors, Extend will not close the block and will give you a chance to fix them.

Testing the block

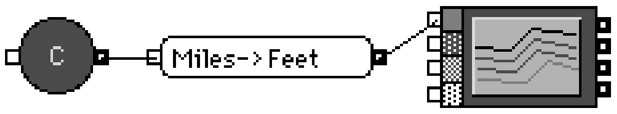
To test the Miles block:

- Open a new model window if one is not already open.
- Add a Miles block by selecting the Convert library (*Macintosh*: Convert Lib; *Windows*: convert.lix) from the Library menu and dragging across to the Miles block.
- Choose Open Library from the Library menu and open the Generic library (*Macintosh*: Generic Lib; *Windows*: generic.lix).
- Add a Constant block from the Inputs/Outputs submenu to the model and move it to the left of the Miles block.
- Connect the output of the Constant block to the input of the Miles block:



Constant block connected

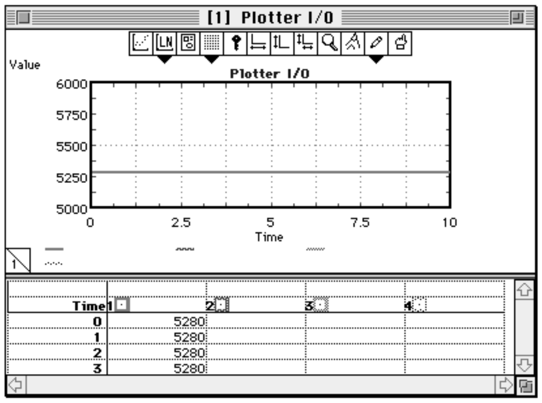
- Open the Constant block’s dialog and enter `10` for the constant value.
- Open the Plotter library (*Macintosh*: Plotter Lib; *Windows*: plotter.lix).
- Add a Plotter, I/O block to the model and move it to the right of the Miles block.
- Connect the output of the Miles block to the top input of the Plotter, I/O block:



Test model completed

- Run the simulation.

▸ By default, the plotter will autoscale at the end of the simulation run. The scaled results are:



Time	1	2	3	4
0	5280			
1	5280			
2	5280			
3	5280			

Plotter output (Macintosh)

As expected, the value 52800 was put out for the entire length of the simulation. You can verify that the block works with other numbers by entering them in the Constant block, running the simulation, and rescaling the plotter output.

Adding user interaction and display features

If moving numbers between input and output connectors was all that Extend did, it would not be a very useful program. As you have seen in previous chapters, robust blocks let you do two things:

- change their parameters
- give information about what is happening during the simulation

To see how easy this is to do in a block, you now want to extend the Miles block to do two more things:

- allow you to convert miles to kilometers, yards, feet, or inches as specified in the dialog
- show the numbers passing through the block as the simulation is run

In addition, you will add tabs to the dialog to separate dialog items by function.

Dialog

The new dialog must have buttons for choosing the unit to convert to and entry boxes to display the numbers that are passing through the block.

▸ Open the structure and dialog windows for the Miles block by holding down the Option (*Macintosh)* or Alt (*Windows*) key and double-clicking on the icon in the model. Bring the dialog window to the front.

⇨ Make the dialog taller.

⇨ Double-click on the text (`Converts miles to feet.`) to open the dialog item.

⇨ Change the text to read:

```
Converts miles to other units.
```

⇨ Click OK.

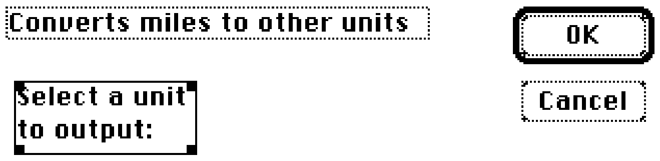
⇨ Choose New Dialog Item and add another text label:

```
Select a unit
to output:
```

You can make sure that the text will be on more than one line by pressing the Return key after the word `unit`. Click OK.

⇨ Resize the box so that both lines of text show.

⇨ Drag the text down a bit:



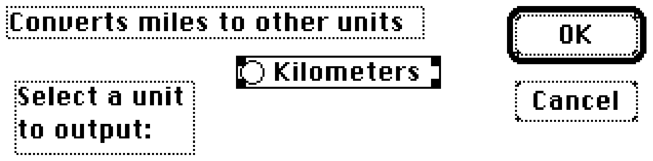
New label added

⇨ Choose New Dialog Item from the Define menu.

⇨ Choose Radio Button. Extend gives this first control group a number of 0.

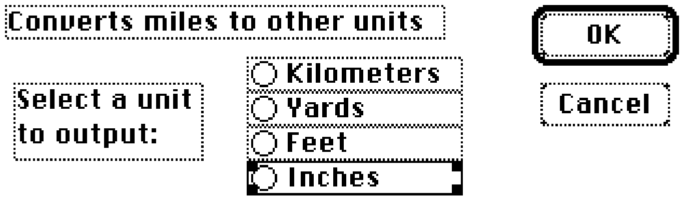
⇨ Enter `Kilometers` for both the button title and the dialog name. Click OK.

⇨ Drag the new button to the right of the text and above it:



New button added

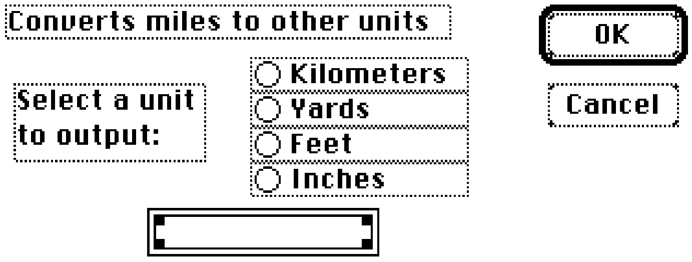
- Repeat the previous four steps to add buttons that have titles and dialog names of **`Yards`**, **`Feet`**, and **`Inches`**. The dialog window now looks like:



Four units buttons added

To add entry boxes to display the numbers:

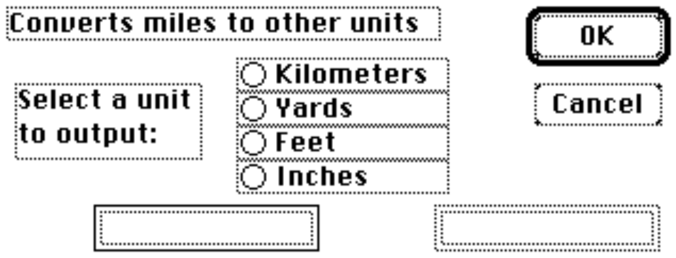
- Choose New Dialog Item from the Define menu.
- Choose Parameter (Number). Enter **`InNum`** for the dialog name. Click OK.
- Drag it below the other items:



First entry box added

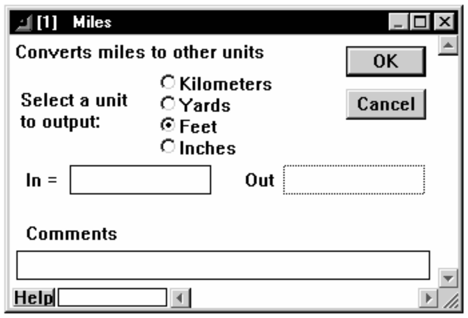
- Choose New Dialog Item from the Define menu.
- Choose Parameter (Number) again. Enter **`OutNum`** for the dialog name.
- Select the Display Only choice (this will cause the parameter value to be displayed in the dialog without being editable). Click OK.

- Drag it to the right of the first entry box:



Second entry box added

- Choose New Dialog Item from the Define menu.
- Choose Static Text (Label). Enter `In =` for the label text. Click OK.
- Move the box down, shrink it, then move it to the left of the first entry box.
- Choose New Dialog Item from the Define menu.
- Choose Static Text (Label). Enter `Out` for the label title. Click OK.
- Move the box down, shrink it, then move it to the left of the second entry box. The dialog window becomes:



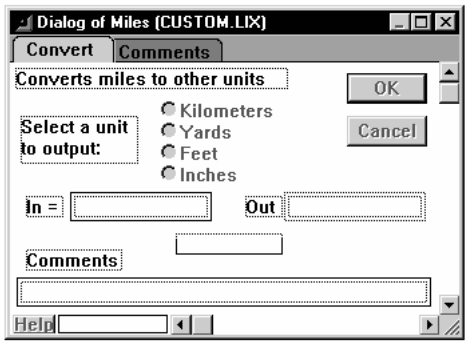
New dialog window (Windows)

Now add tabs to that dialog to separate the comments field from the parameters dealing with the conversion. Dialogs can contain multiple tabs which are similar to pages in the dialog. As you have seen, when a block is first created, it does not contain any tabs. To add a tab:

- Choose Define New Tab from the Define menu (or double-click in the dark grey area at the top of the dialog window).
- Name the tab by entering *Convert* in the name field. Click OK.

To add a second tab for the comments field:

- Double click on the dark grey area next to the Convert tab (this is an alternate to choosing the Define New Tab command).
- Name the tab by entering *Comments* in the name field. Click OK. Click on the Convert tab. The dialog window becomes:



Dialog of Miles block with tabs (Windows)

If you click on the Comments tab, you will see that the only dialog items on this tab are the OK and Cancel buttons. To make the “Converts miles to other units” text appear on both tabs:

- Double click on the “Convert miles to other units” text to open the dialog item.
- Check the Visible In All Tabs checkbox. Click OK.

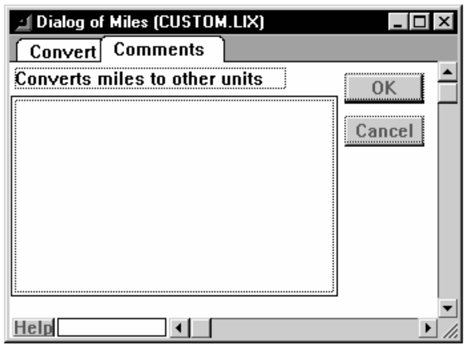
Since the comments field will be on its own tab, the `Comments` label is no longer necessary. To delete the “Comments” label:

- Select the “Comments” text.
- Press delete.

To move the comments field to the Comments tab:

- Select the comments field.
- Select the Move Selected Items to Tab command from the Define menu.
- Select the Comments tab in the “Select tab to move items to:” window. Click Move Items to Tab.

▸ Select the Comments tab. Arrange and resize the comments field to look as shown below:



Converts miles to other units

Comments tab of Miles block (Windows)

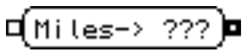
New dialog items will be added to whichever tab is in the front of the dialog window.

Since you have made many changes to the dialog, you should save your changes. However, you are not yet ready to close the block. Choose Save Block from the File menu to save your changes without closing the block.

Note Save Block does not compile the block, so ModL code changes are not saved.

Icon
Now that you have finished changing the dialog, you can change the other parts of the structure. Bring the structure window forward.

You should change the icon since the block now does much more than convert miles to feet. A different icon might be:



New icon

▸ Select the text tool from the tool bar and click on the text box for `Miles->Feet`.

▸ Select "Feet", type `???`, and press Enter.

Connectors
You should change the name of the output connector from `FeetOut` to `UnitsOut` since the number being passed out of it is not necessarily feet. This will help make your code more understandable.

Help text
You should update the help text to explain the choices in the dialog. You may think that they are obvious, but remember that you may give this block to someone else. To be sure that they under-

stand that the block converts from miles to the chosen unit, add new help text. Remember that you can use commands from the Text menu to format your help text.

ModL code

Add the CreateBlock message handler so that you can specify one of the conversions as the default for the block when it is placed in the model. You do not have to type the comments below (comments are text beginning with two slashes or two asterisks), but they are useful for documenting how the ModL code works. For instance, to make feet the default:

```
// comments are optional and begin with two slashes or two asterisks
on CreateBlock    // this gets executed when the user gets a new block
{
Feet = TRUE;      // highlight the Feet radiobutton by making it TRUE
}
```

You can put the CreateBlock handler anywhere in the block's code after the declarations.

For each step in the simulation, you want to:

- Update the input entry box (the first parameter)
- Check which conversion is being performed
- Multiply the numbers
- Update the output entry box (the second parameter which is display only)

To do this, change the Simulate message handler to:

```
on Simulate          // beginning of the SIMULATE message handler
{
InNum = MilesIn;     // Display the value input by setting the dialog
                     // variable "InNum" to the input connector value
if (Kilometers)  // if kilometers radiobutton was TRUE
     UnitsOut = MilesIn * 1.609344;   // set output connector to eqn
else if (Yards)       // if yards radiobutton was TRUE
     UnitsOut = MilesIn * 1760.0;     // set output connector to eqn
else if (Feet)        // if feet radiobutton was TRUE
     UnitsOut = MilesIn * 5280.0;     // set output connector to eqn
else if (Inches)      // if inches radiobutton was TRUE
     UnitsOut = MilesIn * 63360.0;    // set output connector to eqn

OutNum = UnitsOut;  // Display the value output by setting the dialog
                    // variable "OutNum" to the output connector value
}
```

Remember that you can copy the connector and dialog names from the panes at the left of the structure window.

Here is an explanation of what happens in the block's code during a simulate message: if the "Yards" radio button was highlighted in the dialog, the first "if (Kilometers)" evaluates false and its "else" clause is executed. That statement, "if (Yards)", evaluates true and executes its statement, and the "else" clauses following it will not be executed. The other radio buttons behave similarly.

The numbers multiplied by MilesIn can be reals or integers. ModL performs all necessary type conversions for you automatically. However, since connectors are always of type real, you should use reals for values that are used with connectors so Extend will not need to convert integers to reals on each step.

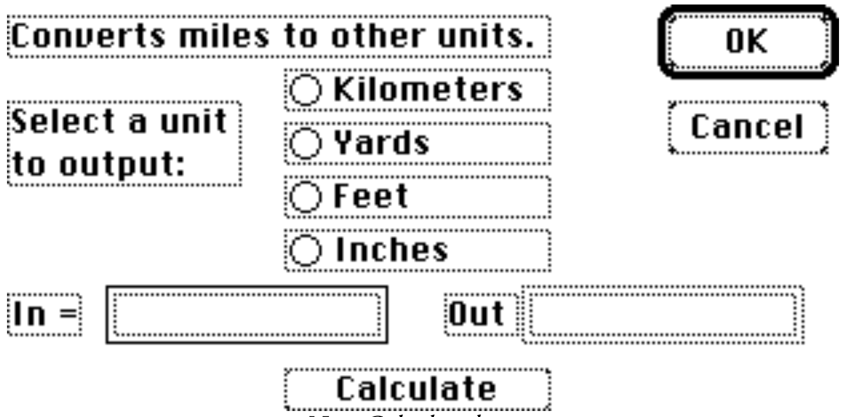
Compile and save the block
After you have finished writing your code, close the block by clicking on the close box in the upper left corner. Click "Save, Compile If Needed" to compile the new ModL code and save the block in the library.

Test the Miles block as you did before, trying all the conversions. Remember that you can now keep the Miles dialog open during the test.

Adding an intermediate results feature

You may have noticed something slightly frustrating about the Miles block: what if you just want to convert a single number without running a whole simulation? As stated earlier, Extend passes dialog messages to a block even if a simulation is not running. It is thus easy to make this block useful even outside of a simulation.

- Stretch the dialog down to make more space.
- Open the block's internal windows again and select the dialog window.
- Choose New Dialog Item from the Define menu.
- Choose Button. Enter `Calculate` for both the button title and the dialog name. Click OK.
- Drag this new button below the others:



New Calculate button

▸ Add the following message handler to the end of the ModL code:

```
on Calculate     // Display the values
{
if (Kilometers)
     OutNum = InNum * 1.609344;
else if (Yards)
     OutNum = InNum * 1760.0;
else if (Feet)
     OutNum = InNum * 5280.0;
else if (Inches)
     OutNum = InNum * 63360.0;
}
```

▸ Save the block and compile the ModL code.

To test this new functionality:

▸ Open the block's dialog. You may need to resize it since you modified it.

▸ Type a number in the "In:" entry box.

▸ Click Calculate.

Note that the number in the "Out:" entry box is updated with the proper conversion.

Adding animation

The icon of the block tells us that it converts miles to something, but what is the something? Displaying text on an icon is a convenient method to show block conditions; in this case, what kind of conversion the block is performing. Note that this kind of animation works even if the user doesn't have animation turned on during the simulation run.

To see how easy this is to do in a block, you now want to add animation to the Miles block so that it displays the output units when the user clicks on the radio buttons in the dialog.

Changing the icon and adding the animation object

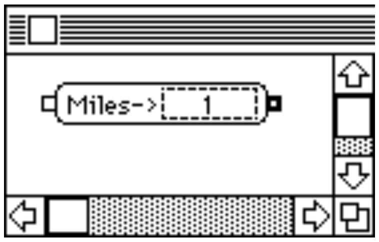
▸ Open the structure window of the Miles block and bring it to the front.

▸ In the icon pane, delete the "???" part of the icon text (Double-click on the "???" to select it, then press the delete or backspace key.)

▸ Move the output connector to the right so that the icon can be lengthened a bit.

▸ Resize the icon so that it extends more to the right (Click on the icon to select it. Then click on the resizing rectangle on the lower right corner of the icon and drag it to the right to stretch it a little.)

▸ In the toolbar, select the Animation Object tool and create an animation rectangle to the right of the text. Since this is the first animation rectangle, it will have a "1" in it. Use the mouse to position it carefully within the icon. You can use the Option (*Macintosh)* or Alt key (*Windows*) to ungrid the rectangle for finer positioning.



Icon with added Animation rectangle

Adding the animation code

‣ Add the following message handlers to the end of the ModL code. These message handlers receive a message when the corresponding radio buttons get clicked:

```
on kilometers     // Kilometers radio button was clicked
{
AnimationText(1, "KM"); // set up animation text value
AnimationShow(1);       // show the animation object
}


on yards          // Yards radio button was clicked
{
AnimationText(1, "Yards"); // set up animation text value
AnimationShow(1);          // show the animation object
}


on feet           // Feet radio button was clicked
{
AnimationText(1, "Feet"); // set up animation text value
AnimationShow(1);         // show the animation object
}


on inches         // Inches radio button was clicked
{
AnimationText(1, "Inches"); // set up animation text value
AnimationShow(1);           // show the animation object
}
```

‣ Next add some code to the “On CreateBlock” message handler to initialize the animation text.

```
on CreateBlock  // this gets executed when the user gets a new block
{
Feet = TRUE;    // highlight the Feet radiobutton by making it TRUE
AnimationText(1, "Feet");  // set up default animation text value
AnimationShow(1);          // show the animation object
}
```

▸ Next add a “On OpenModel” message handler to display the animation text.

```
on OpenModel // this gets executed when the user opens the model
{
if (Kilometers)                     // kilometers is selected
    AnimationText(1, “KM”);          // set up animation text value
else if (Yards)                     // yards is selected
    AnimationText(1, “Yards”);       // set up animation text value
else if (Feet)                      // feet is selected
    AnimationText(1, “Feet”);        // set up animation text value
else                                 // inches is selected
    AnimationText(1, “Inches”);      // set up animation text value
AnimationShow(1);                    // show the animation object
}
```

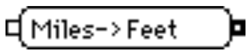
▸ Save the block and compile the ModL code.

Testing the block
To test this new functionality:

▸ Open the block’s dialog.

▸ Click “Feet” and see how the block’s icon changes.

If the dialog hides the block on the model, move it away so that you can observe the changes when you click different radio buttons in the dialog. The icon should look like:



Icon with animation added

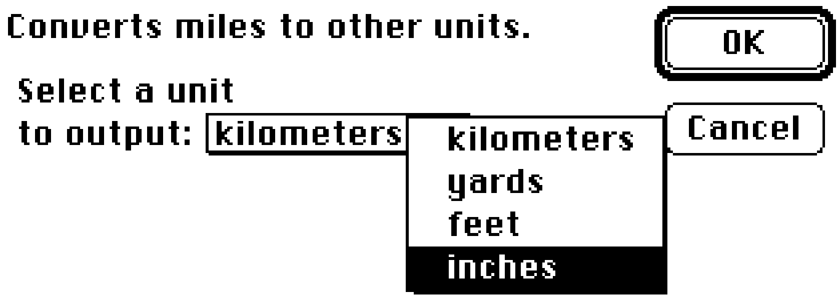
This immediately tells the user what the block is doing without having to open its dialog.

If you want to compare your block to the one we built, the final Miles block is located in the Custom Blocks library (*Macintosh*: Custom Blocks Lib; *Windows*: custom.lix).

Alternatives and enhancements

The preceding example showed you how to build a simple block in Extend. There are alternative features you may want to add to your blocks. For example, you may want to use other Extend features when building your block. Or you may wish to leverage your investment in code from other languages.

Popup menus
As an alternative to using radio button groups, you can use popup menu (see "Types of dialog items" on page P9). In this example, one popup menu can be used to allow the user to select a unit to output. Using a popup menu, the dialog would look like:



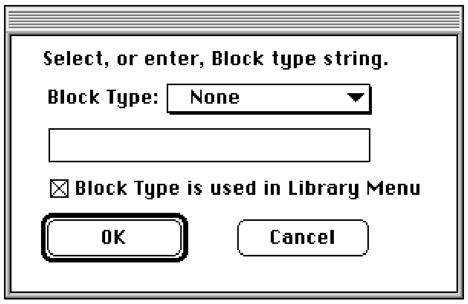
Popup menu as an alternative to radio buttons

Block types
The Miles block is a single block in its own library. It is more common that you will build several blocks to put in a library. You can have all the blocks listed alphabetically in the library, or you can specify a block *type* and have the blocks be grouped into submenus by type.

Each block has a 15 character string associated with it called the block type. The block type is used by Extend to organize the blocks into logical groups of blocks that perform a similar function. For example, the Add block and the Subtract block from the Generic library both perform basic math functions, therefore their block type is Math. Extend uses the block type in two places: 1) to organize blocks within the library menu (as discussed in "Libraries" on page 230) and 2) to organize blocks within the reports (as discussed in "Model reporting" on page 150).

To set or edit a block's type:

▸ While the block's structure window is open, select Set Block Type from the Define menu.


Select, or enter, Block type string.
Block Type: None
☒ Block Type is used in Library Menu

Set BlockType dialog

▸ Either select an existing block type from the popup menu or define a new block type by entering it into the Type field.

▸ Click OK.

The *Block Type is used in Library Menu* checkbox in the Set Block Type dialog is used to determine how Extend will list this block within the library menu. With the box checked, Extend will list the block in a submenu under the block type under the library name. If the box is unchecked, Extend will list the block alphabetically directly under the library name.

Accessing code from other languages
You can use other languages, such as Visual C++, Fortran, C++, etc., to provide functionality that may be missing in ModL, or to make use of a legacy of pre-built code. For example, you may already have thousands of lines of C++ code which perform a specific calculation. You can add this functionality to your block by recompiling your C++ code as a DLL (dynamic link library for Windows) or XCMD (external command for Macintosh). DLLs and XCMDs are the standard interface technologies for communicating between programming environments. When you want to do the calculation, use Extend's built-in function calls to call your DLL or XCMD.

One advantage of this method is that you don't have to program the interface in the external language; you can leverage off Extend's built-in interface and dialog creating capabilities. The result is that, even though you are using an external language, the resulting block will fit seamlessly into the Extend environment and be indistinguishable from other blocks.

For example, the following code in the script pane of the Miles block calls a DLL that performs the same function as the ModL code you saw on page P51. (Note that you could accomplish the same result, but in a slightly different manner, using an XCMD if you are working on a Macintosh.)

```
// Declare constants and static variables here.
```

```
long proc;

on Calculate     // Display the values
{
proc = DLLMakeProcInstance("convert");
if (proc > 0)
     OutNum = DLLDoubleCFunction(proc, inNum, kilometers, yards, feet, inches);
else
     Beep(); // something didn't work right
}


// This message occurs for each step in the simulation.
on simulate
{
InNum = MilesIn;// Display the value input by setting the dialog
                // variable “InNum” to the input connector value
proc = DLLMakeProcInstance("convert");
if (proc > 0)// You must check if proc>0 before you make the call
     unitsOut = DLLDoubleCFunction(proc, inNum, kilometers, yards, feet, inch-
es);
else
     Beep(); // to show that something didn't work right
OutNum = UnitsOut;  // Display the value output by setting the dialog
                      // variable “OutNum” to the output connector value
}
```

The code of the DLL, written in C++, is:

```
double __export __cdecl convert(double x, long kilometers, long yards, long feet,
long inches)
{
     if (kilometers)
          return(x * 1.609334);
     else if (yards)
          return(x * 1760.0);
     else if (feet)
          return(x * 5280.0);
     else if (inches)
          return(x * 63360.0);
}
```

For more information, see “DLLs” on page P238 and “XCMDs and XFCNs” on page P237.

ModL feature review

As you have seen, ModL is very much like C, although it is not quite as complicated and not case sensitive. ModL has a few extensions that will help you create block code:

- Block code is organized mainly as message handlers, rather then just as procedures with passed parameters. Message handler can have local variables.

- The names of connectors and dialog items are treated as static variables that can be read or set from within message handlers or procedures. The dialog changes take effect immediately.
- Blocks can query, control, and send messages to other blocks, even if they are not connected.
- ModL has several string data types. Strings can be up to 255 characters and can be concatenated with the + (plus) operator.
- ModL has subscript checking, causing the code to abort if you try to access an element beyond the size of an array.
- There are almost 600 ModL functions for performing general and simulation-specific tasks. In addition, there are many global variables that can be accessed from ModL code. There are also a few predefined constants.
- Including animation in a block is easy. You do not need to do any graphics in your block code; you can position animation objects on the icon and then show pictures or text within the objects using animation functions.
- You can use multi-dimensional arrays (with up to five sets of dimensions). Arrays can be passed to blocks or functions as entire arrays or as individual elements. ModL arrays can be *fixed* (specified size) or *dynamic* (one dimension undeclared). You can use dynamic arrays to hold values when you do not know how many elements will be needed.
- ModL allows for efficient connectivity with other languages, so you can make use of other features and technologies.

Chapter 3: The ModL Language

A detailed description of ModL, including all of the variables and functions that can be used in your block code

"Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information upon it."
— Samuel Johnson

This chapter reviews ModL language constructs and tells you all the ways that the structure of ModL differs from ANSI C. It also includes a complete list of the system variables and functions that you can use in your ModL code.

As seen in the table below, ModL is structured very much like C. If you know C, you know most of ModL. If you don't know C, the section "ModL and other languages" on page P18 presents a table comparing ModL to Pascal, BASIC, and FORTRAN. In addition, XCMDs, and DLLs provide a method for linking between Extend and languages other than ModL. For more information, see "XCMDs and XFCNs" on page P237 or "DLLs" on page P238.

This chapter is intended as a reference to the ModL programming language; it is not a programming tutorial. Some excellent programming books are listed in the reference section of the Introduction. To get the most out of this chapter, you should first read Chapter 1: Parts of a Block, and Chapter 2: ModL Code Overview and Block Creation.

ModL compared to C

As a convenience, this table is repeated from Chapter 2: ModL Code Overview and Block Creation:

ModL	**C**
case insensitive	case sensitive
real or double (*PowerPC and* ⊞ *Windows*: 16 significant digits)	double
integer or long (32 bit)	long

string or Str255 (255 characters maximum)	typedef struct string {
	unsigned char length;
	unsigned char str[255];
	} string;

str15 (15 characters maximum)	typedef struct string
	{
	unsigned char length;
	unsigned char str[15];
	} string;

str31 (31 characters maximum)	typedef struct string
	{
	unsigned char length;
	unsigned char str[31];
	} string;

ModL	C
// comments to end of line only ** comments also supported	/* comments can be multi-line */
Array bounds subscript checking produces error messages when array bounds are exceeded	No array bounds checking
Functions declared using ANSI declaration only (prototype declarations).	Functions can be declared either K&R or ANSI
Statements cannot be used as expressions. For example “a[++i] = 5;” or “a[i=i+1] = 5;” are not allowed. They must be “i++;” or “i=i+1;” and “a[i] = 5;”. Therefore in ModL, “i++” and “++i” are functionally the same.	Expressions can be statements
The ModL “for” statement is: for (statement; boolean; statement)	The C “for” statement is: for (expression; expression; expression)
^ is used as the exponentiation operator (like it is in BASIC and spreadsheets)	^ is used as the exclusive-OR operator in logical expressions
To concatenate a string use: stringVar = stringVar+"abc";	The C equivalent is: strcat(stringVar, "abc")
To convert a number to a string: stringVar = x;	The C equivalent is: ftoa(x, str);
if (stringVar < "abc") ...	if (strcmp(stringVar, "abc") < 0) ...
Resizable dynamic arrays	Pointers
Use dynamic arrays that have other dynamic arrays as elements (arrays of arrays), using the PassArray and GetPassedArray functions.	Structures
!= or <> are not-equal operators	!= is the not-equal operator
% or MOD are the modulo operators	% is the modulo operator
Bit handling done by functions (such as BitAnd(n, m))	Bit handling done by operators (such as n&m)

Structure

Names

Names for variables, constants, functions and procedures can be up to 63 characters. Names can have letters, numbers, and the underscore (_) character; a name must begin with a letter or the underscore (_). Some names are reserved for system use, as described throughout this chapter (reserved names are listed in Appendix C).

ModL is a case insensitive language. This means that the following identifiers appear the same to the compiler:

```
Myname      MYNAME      MyName
```

Data types

There are five data types in ModL: 1) real or double, 2) integer or long, 3) str15, 4) str31, and 5) string or Str255. For the data types with multiple identifiers, the identifiers are interchangeable; this manual uses *real*, *integer*, *str15*, *str31* and *string*.

Real numbers are stored as Extended IEEE floating point numbers with 16 significant digits (*PowerPC* and *Windows*). Numeric literals containing a decimal point or E notation are assumed by the compiler to be real numbers. The range of real values is approximately ±1e±308.

Integer numbers are stored as 32 bit integers. The maximum value is 2147483647, and the largest negative value is -2147483648.

Str15s may contain up to 15 characters. Str31s may contain up to 31 characters. Strings may contain up to 255 characters. String literals (constants) are sequences of characters enclosed by quotes ("). The quote character itself may appear in strings by using two adjacent quotes. For example:

```
"It's called ""Extend"""
```

is evaluated as:

```
It's called "Extend"
```

If you have a string literal that you want to extend past one line of the code, put a backslash (\) character as the last character on the line. For example:

```
longString = ""This is a very long string \
literal that is on more than one line";
```

ModL treats a non-zero value as TRUE and a 0 value as FALSE. The constant TRUE is defined as 1 and FALSE is defined as 0.

Declarations and definitions

Static and local

Variables and constants must be declared before being used. Static and constant declarations are made at the beginning of a block's code and local declarations are made at the beginning of a message handler or user-declared function.

The values of static variables are stored in the block and are saved in the model file. Thus, they may be used to store data which must be preserved from one run of a simulation to the next. However, they are not initialized and must be initialized in your block code.

Two advantages of using local variables are that they are not saved with the model and they do not use any of the block's memory allocation.

Note The disadvantages of using local variables are that the values of local variables are not remembered after the message handler or function is left, and they can override static variables while the message handler is being executed.

Type declarations

The general forms of the type declarations are:

```
REAL      id, id, ......id;      or    (DOUBLE    id, id, ......id;)
INTEGER   id, id, ......id;      or    (LONG      id, id, ......id;)
STR15     id, id, ......id;
STR31     id, id, ......id;
STRING    id, id, ......id;      or    (Str255    id, id, ......id;)
```

Constant definitions

The general form for a constant definition is:

```
CONSTANT id IS literal;
```

Note that in the constant definition the type is implied by the format of the literal value. This means that if you want a constant to be real, it must either contain a decimal point or be in E notation. Regardless of where they are defined in the ModL code, constants are always static. ModL includes four general-purpose predefined constants: Pi, Blank, True, and False, as discussed below.

Numeric type conversion

Generally, ModL performs all type conversion automatically. Thus, integer values can be assigned to real variables, and mixed type arithmetic can be performed without explicit type conversion beforehand.

- In an operation between an integer and a real, the result is always a real. Note that using integer constants for real expressions is dangerous and can give the wrong results. For example:

  ```
  z = 1/2*a;
  ```

  z will always equal 0: the integer 1 divided by the integer 2 equals 0 because the result of integer operations must be an integer. The statement should be instead written as:

  ```
  z = 1.0/2.0*a;
  ```

- The result of an operation between any type and a string is a string. This makes it easy to make strings that contain numeric results, such as:

  ```
  anOutputString = "The answer is " + aRealNum + ".";
  ```

If a real is converted to a string and that real has a value of BLANK (a NoValue), the string generated is “NoValue”.

- ModL converts the arguments of function calls to the type needed by the function. For example:

  ```
  a = cos(integer);
  ```

  Because the cosine function expects the argument to be a real value, the integer is converted to a real value before the function is called. Note, however, that because this conversion takes time, you should eliminate argument conversions in your blocks to make them run faster.

Arrays

You can use real, integer, or string arrays. Any array can have up to five dimensions, that is, up to five subscripts or indexes. Array indexes are limited to 2 billion elements.

Array declarations

The number of dimensions and the magnitude of each dimension is determined by the array’s type declaration. The magnitude of each dimension appears in the square brackets in the declaration statement, and there must be one set of brackets for each dimension. The subscripts in an array start at 0. The type declarations are of the form:

```
TYPE id [dim1][dim2]...;
```

Thus the declaration:

```
REAL MyArray[3][4];
```

declares an array of real numbers identified by the name MyArray. This is a two dimensional array, with three rows and four columns.

Individual elements of arrays are treated just like variables in ModL. Thus, for an array declared as **`integer a[2][3]`**, you can assign the value **`4`** to the first row in the second column with:

```
a[0][1] = 4;
```

Remember that array subscripts start at 0 and end at 1 less than the number of elements in the declaration (that is, there are n elements in 0 to n-1).

Fixed and dynamic arrays

ModL has both *fixed* and *dynamic* arrays. Fixed arrays have specified sizes that cannot be changed in the code. The leftmost dimension of dynamic arrays are variable in size and can be assigned a size or resized without changing existing data. Dynamic array size is limited to two billion elements, and you can declare up to 254 dynamic arrays per block.

Dynamic arrays can be passed between blocks or used globally. See “Passing arrays” on page P213.

There is another kind of array called "Global Array" that can be set up using functions and is useful for data needed globally throughout the model. See "Global arrays" on page P83.

Dynamic arrays are declared in the same manner as fixed arrays, except that their first dimension value is missing. For example:

```
REAL MyArray [ ] [4];
```

defines a two-dimensional dynamic array of real numbers. There is a varying number of rows and exactly four columns (indexed 0 through 3).

Dynamic arrays must be static variables, not local ones. Thus, you cannot declare a dynamic array variable in a message handler. However, you can resize dynamic arrays inside of a message handler or in a user-defined function. There are three functions for sizing dynamic arrays:

Function	Use
MakeArray	Sets the size of the missing dimension
GetDimension	Returns the size of the missing dimension
DisposeArray	Frees memory when you are finished with the array

The GetDimension function can be used with any array, not just dynamic arrays. GetDimension returns the value of the first (leftmost) dimension.

These functions are described fully in the function listing later in this chapter.

Arrays as arguments to functions

When passing an array name to a function, it is necessary to differentiate between passing the entire array, and passing only an element of the array. In ModL when you wish to pass an entire array, you should supply only the array name, without subscripts. If you wish to pass only a single element of an array, use a subscripted array name.

Many ModL functions take arrays as arguments. To pass an array that has the same dimensions as the function needs, simply use the array's name. For example, the arguments to the AddC function must be arrays with two elements, that is, declared such as `real a[2]`. Assume you had three arrays declared as:

```
real x[2], y[2], z[2];
```

You would call the AddC function as:

```
AddC(x, y, z);
```

If you have an array with more dimensions than are needed by the function, you can specify an *array segment* as an argument. An array segment is an array with fewer dimensions than the full array. Array segmenting can only be done by specifying the leftmost dimensions, not the rightmost dimensions. For example, assume that you had the following declarations:

```
real x[2], y[2];
real z[50][2];
```

To pass the fifth row of **z**, which contains two elements, to a function such as AddC which only wants an array of one dimension, you would use:

```
AddC(x, y, z[4]);
```

Operators

ModL offers a full set of mathematical and logical operators. The assignment operators are:

Operator	Description
id = expression;	assignment statement
id += expression;	equivalent to ***id*** = ***id*** + expression
id -= expression;	equivalent to ***id*** = ***id*** - expression
id *= expression;	equivalent to ***id*** = ***id*** * expression
id /= expression;	equivalent to ***id*** = ***id*** / expression
id++;	equivalent to ***id*** = ***id*** + 1
++***id***;	also equivalent to ***id*** = ***id*** + 1
id--;	equivalent to ***id*** = ***id*** - 1
--***id***;	also equivalent to ***id*** = ***id*** - 1

The standard binary math operators are:

Operator	Description
+	addition and concatenation
-	subtraction
*	multiplication
/	division
^	exponentiation. ModL (unlike C) uses ^ to denote exponentiation, as in **2^4**.
%	modulo
MOD	modulo

The + operator is used both to add numeric values and to concatenate strings. For example:

```
str = "My name"+" is Ralph";
```

returns the string “My name is Ralph”.

```
    str = "Time = " + currentTime;
```

returns the string “Time = 5.65” if currentTime equals 5.65.

The % and MOD operators return the remainder after integer division. For example, 5 MOD 2 returns 1 and -5 MOD 2 returns -1.

Any operation involving a noValue (BLANK) produces a noValue result. NoValue results appear as blank entries in tabular data and are not reflected in plotted traces at all.

The Boolean and magnitude operators are:

Operator	**Description**
AND or &&	combination
OR or ||	conjunction
NOT or !	inverse
!= or <>	not equals
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to

ModL supports the three standard Boolean (logical) operators: AND and &&, OR and ||, and NOT and !. These three operators are not bit-wise logical operators, but only operate on TRUE or FALSE expression values.

The standard magnitude operators return a Boolean value (1 for true and 0 for false) after comparing their arguments. The operators are != and <> (not equal), <, <=, >, >=, and ==.

Operators in an expression are generally evaluated from left to right, however there is a hierarchy of precedence among the operators. The following list is in descending order of precedence; within each group, operators have equal precedence. Note that putting any expression in parentheses insures that it will always be evaluated first.

1) ( )

2) - (negation)

3) NOT or !

4) ^
5) MOD or %, *, /
6) + (addition and concatenation), -
7) <, >, <=, >=, = =, <> or !=
8) OR or ||, AND or &&

Control statements and loops

ModL supports a full complement of structured programming control statements. In this table, “boolean” evaluates to true or false. The control statements are:

Statement	**Use**
ABORT	ABORT;
BREAK	BREAK;
CONTINUE	CONTINUE;
DO-WHILE	DO STATEMENT; WHILE (boolean); // Note semicolon
FOR	FOR (init_assignment; boolean; incr_assignment) STATEMENT;
GOTO	GOTO label; … label: //note the colon …

Statement	Use
IF	IF (boolean)
	STATEMENT;
IF-ELSE	IF (boolean)
	STATEMENT_A;
	ELSE
	STATEMENT_B;
RETURN	RETURN;
	or
	RETURN(value or expression);
SWITCH	SWITCH (expression)
	{
	CASE integerConstant:
	many STATEMENTS;
	BREAK;
	CASE integerConstant:
	many STATEMENTS;
	BREAK;
	DEFAULT:
	many STATEMENTS;
	BREAK;
	}
WHILE	WHILE (boolean) // Note no semicolon
	STATEMENT;

Multiple statements can be grouped with braces:

```
{
statement;
statement;
...
}     // Note no semicolon here
```

Use the IF statement for boolean comparisons such as

```
if (total < 0)
     {
     isNegative = TRUE;
     findings = abs(total);
     }
...
```

You can also use the IF-ELSE construct if you have two paths to choose from:

```
if (total < 0)
     {
     isNegative = TRUE;
     findings = 100 + total;
     }
else
     findings = 100 - total;
...
```

The FOR construct lets you set the initial value, continuing boolean condition, and action to take on each step in the parentheses. For example:

```
for (i = 0; i <10; i++)
     a[i] = i;
```

will set a[0] to 0, a[1] to 1, and so on up to a[9].

The WHILE loop repeats the statement while the expression is true; the expression is evaluated at the beginning of the loop. Thus,

```
x = 3;
y = 3;
while (y<3)
     {
     x++;
     y++;
     }
```

would leave x set to 3 because the loop is never executed.

The DO-WHILE construct tests at the end of the loop, so

```
x = 3;
y = 3;
do
     {
     x++;
     y++;
     }
while (y<3)
```

would leave x set to 4 because the test occurs at the end of the loop, after x is already incremented.

The SWITCH statement is used to check values against integer constants and act on those cases. Note that the integers in the CASE statement must be constants, not variables. For instance,

```
switch (numberOfRepeats)
     {
     case 0:
          UserError("You didn't run it.");
          wasOK = FALSE;
          break;
     case 1:
          UserError("Thank you, it ran once.");
          wasOK = TRUE;
          break;
     default:     // any other number
          UserError("You ran it more than once; please stop.");
          wasOK = FALSE;
          break;
     }
```

The GOTO syntax is supported only within a message handler or user defined function. Control is unconditionally transferred to the statements after the label.

The ABORT statement stops the current message handler. You can use this at any point, even inside loops. For example, in a DialogOpen message handler, the ABORT statement would prevent the dialog from opening. The two most common uses of the ABORT statement are in the CheckData and Simulate message handlers. In CheckData, it can be used to abort if the user's data is bad. In the Simulate message handler, it aborts the current simulation if something goes wrong with the calculations. (There is also an AbortAllSims function that can be used to abort multiple simulations.)

ModL also provides two control words for exiting from loops and other control structures. BREAK immediately exits an enclosing FOR, WHILE, DO, or SWITCH construct. CONTINUE immediately sends control to the next iteration of an enclosing loop.

The RETURN statement is used to exit message handlers and user-defined functions. When returning from a message handler or user-defined function that does not return a value, use:

```
return;
```

When returning from a user-defined function that returns a value, use:

```
return(value or expression);
```

User-defined procedures and functions

The rest of this chapter is a complete listing of the built-in ModL functions and procedures. Extend also allows you to define your own functions and procedures.

User-defined procedures and functions allow you to create tools that can be reused and make your ModL code more readable. They are defined and called as they are in C. They are local to the blocks in which they are defined. See "Passing messages between blocks" on page P222 for a method of defining "global" functions. Also, see "Pass by value and reference (pointers)" on page P213 for information on passing arguments to procedures and functions.

User-defined procedures have the following form:

```
PROCEDURE (or VOID) name (TYPE id, TYPE id,…, TYPE id) // zero or more arguments
{
optional local variable declarations

zero or more statements

RETURN;
}
```

User-defined functions have the following form:

```
TYPE name (TYPE id, TYPE id,…, TYPE id) // zero or more arguments
{
optional local variable declarations

zero or more statements

RETURN(value);
}
```

In these definitions, TYPE can be integer, real or string, and PROCEDURE (or VOID) defines a function that does not return any value. All arguments are optional. Arrays may be passed as arguments.

User-defined procedures and functions must be defined before they are used since the ModL compiler needs to know the types of the arguments for type conversion. Type conversions are discussed above.

If you define a procedure or function that calls another user-defined procedure or function, both cannot be defined first. The method around this problem is:

```
TYPE or PROCEDURE (or VOID) name (TYPE id, TYPE id,…, TYPE id); //Note the ";"
```

This is called a *forward declaration* and tells ModL the types of the function and arguments, and that the function will be defined later. The form is exactly the same as a function or procedure definition, but instead of braces, it ends in a semicolon.

Exit a user defined function at any point with a RETURN statement for PROCEDURE type, RETURN(value) for other types, or the ABORT statement (see ABORT, above).

For example:

```
real MyCalc(real x1, real x2)
{
real sum;        // declare a temp variable

sum = x1+x2;     // the calculation
return(sum);     // return the sum
}
...
y = MyCalc(a, b); // calc using function
...
```

User-defined functions can be recursive, that is, they can call themselves.

Declaring arrays as arguments

If your array size is not fixed, you can declare arrays as arguments to functions with the first (left-most) dimension blank (additional dimensions must have a value). This allows you to pass either kind of array (fixed or dynamic) and any length of array to the function.

The following is an example of functions that add all of the elements of the rows in a variable-sized array and return the sum.

```
real rowSum(real x[])
{
integer length, i;           // Temporary variables
real sum;

length = GetDimension(x);    // Returns first dimension
sum = 0.0;                   // Initialize sum

for (i=0; i<length; i++)     // For all elements
     sum = sum+x[i];         // Add element to sum

return(sum);
}

on Simulate
{
real     valuesArray[36], mySum;
...
mySum = rowSum(valuesArray);
...
}
```

Message handlers

The system and dialog message handlers are described in detail in “Using message handlers” on page P31. They are repeated here in alphabetical order for your convenience:

Message	When sent
AbortDialogMessage	If the user stops the execution of one of your own dialog message handlers or that message handler executes an ABORT statement, this message gets sent to the block. Use this as an “exception handler” to clean up after an error occurs.
AbortSim	After any premature stopping of a run, such as when you stop a simulation using the Stop button in the Status bar or when an error occurs before the simulation is finished. A good use for the AbortSim message is to reset values that may be incorrect due to incomplete calculation. Note that EndSim always gets executed after AbortSim.
ActivateModel	Sent to all blocks in a model when that model is brought in front of a different model. This allows blocks to modify their data or appearance if the model is activated and is useful for IPC communications from outside of Extend.
AnimationStatus	Sent to all blocks in a model when the Animation menu item changes state. This is useful to change a block's appearance according to the value of the AnimationOn variable.
AdviseReceive	Sent to the block when it receives updated data from an advise conversation (see “Interprocess Communication (IPC)” on page P102).
BlockClick	Sent when the user clicks on a block. Use GetBlockPosition() and the User inputs functions on page P122 to determine where the user has clicked on the block and the status of any modifier keys. See the Mandelbrot model (*Macintosh:* Mandelbrot; *Windows*: mandlbrt.mox).
BlockLabel	Sent to the block that just had its Block Label changed, when the label becomes deactivated. The block can then use the new value of the label.
BlockMove	Sent to all blocks that have been moved upon the completion of that move. (See “Scripting” on page P157)
BlockRead	Users should not use this message. It can cause a crash if used improperly. It is used to convert version 3.x block data tables to version 4.0 dynamic data tables. See the Information block. Call only the GetFileReadVersion() and ResizeDTDuringRead() functions in this message handler.
BlockReceive0 through 4	Custom defined message. These are used by the Discrete Event library as system messages. We suggest you use “UserMsg0 through 9” when you define a message for your own purpose.
BlockReport	Sent after the FinalCalc messages. If a block receives this message, it has been selected for a report. To organize the reports by block type (see “Block types” on page P56), Extend cycles through each block type and sends this message to any block selected for a report.
Cancel	When you click the Cancel button.

Message	When sent
CellAccept	Sent to a block when the user finishes editing any cell in any of the block's data tables. You can use this to check the data entered in cells in a data table.
CheckData	After the Run Simulation command and before the simulation begins. This is the best place to check whether all data that is to be used in the block is valid. If the data is bad, you should execute an "abort" statement. Extend will select the block and alert you that the data is missing or bad. During this message, connectors that are connected to something else in the model always have a *true* value and unconnected ones have a *false* value. This makes it easy to check whether a block is connected properly before running a simulation.
ClearStatistics	When a block's statistical variables need to be reset. This message is typically sent by a block in the Statistics Library. See the Activity, Delay block in the Discrete Event library for an example of receiving this message.
ConnectionMake	Sent to all newly connected blocks when the user makes a connection on the model. For hierarchical blocks, this is sent to all internal blocks. To undo the connection, you can call the *Abort* statement during this message handler.
ConnectorName	Connector message. When the named connector on a connected block receives a message from another block using the connector message functions. The Discrete Event library blocks use this message extensively.
CopyBlock	Sent to all blocks selected before a Copy operation. Useful to dispose of a dynamic array, create a dynamic array, or add a Global Array to the clipboard using the GAClipboard() function, before the copy.
CreateBlock	When the block is added to a model. This is the place to set initial values for the dialog. Note that this message is only sent to the block when you add it to a model. If you change the CreateBlock code for a block that already is in a model, the changes won't affect the existing blocks, only new blocks added to the model.
DeleteBlock	Sent when the user deletes a block from the model. For hierarchical blocks, this is sent to all internal blocks.
DialogClick	Sent when the user clicks on a dialog item, before the actual dialog item message is sent to the block. Use the User inputs functions on page P122 to determine where the user has clicked on the dialog and the status of any modifier keys. This message is used, for example, to modify the items in a popup menu at the time it is clicked on but before it opens to the user.
DialogClose	When the OK or Cancel buttons or the close box are clicked
DialogOpen	When the block's dialog is opened. If you want to display any static text labels that can change based on what is happening in the simulation, set them here. If you don't want the dialog to open, execute an ABORT statement (see Plotter I/O block code).

Message	When sent
DragCloneToB-lock	This message gets sent when a user drags a clone onto a block and releases the mouse button when the block is highlighted. If you want to get information about the clones, call the GetDraggedCloneList() function. Used in the Evolutionary Optimizer block.
EndSim	At the end of the simulation. Use this to clean up any memory that you used or to reset values. This message gets sent even if you or the block's code aborts the simulation.
FinalCalc	Sent after the Simulate messages are over and before the BlockReport messages. Used for any final calculations that the block might need before the BlockReport and EndSim messages.
HBlockClose	When a hierarchical block's submodel or structure is closed. For hierarchical blocks, this is sent to all internal blocks.
HBlockHelpBut-ton	This message is sent to all the blocks inside an Hblock when the Hblock help button is clicked. This can be used to intercept the help button message for the hBlock, and do something of your own implementation instead. Aborting this message will prevent the message from getting to the rest of the blocks in the hBlock, and prevent the help text from opening up.
HBlockOpen	When a hierarchical block is opened, this message is sent to all blocks in its submodel. You can use this to correct animation in a hierarchical block that is opening. If you don't want the hierarchical layout and structure windows to open, execute an ABORT statement in one of the blocks in the submodel.
HelpButton	Sent when the user clicks the Help button on the lower left of the block's dialog.
InitSim	Just before the simulation starts. If your block's code uses DeltaTime, you should check it here. Also, allocate any arrays that are dependent on DeltaTime, NumSteps, or any other system variable. This is also a good time to set any static variables, dialog items, or connectors that change when the simulation starts.
MailSlotReceive	This message is sent repeatedly when there are mailslot messages waiting to be picked up. See the mailslot functions "Mailslot (Windows only)" on page P111 for more information.
ModelSave	This message is sent to each block at the beginning of a save. You might use this message handler to dispose of unneeded data before it gets saved.
OK	When you click the OK button. You do not need to do any special handling in this section unless you want to check input data.
OpenModel	When a model is opened or to a new hierarchical block that has just been placed on the model worksheet. You can use this to set some block variables to values that you only want to reset when a model is opened, not at the beginning of a simulation.

Message	When sent
PasteBlock	When the user pastes a block onto the model. For hierarchical blocks, this is sent to all internal blocks.
PauseSimulation	Sent to ALL blocks when the user pauses the simulation. Contrast this to ResumeSim which is sent only to the blocks that have been changed by the user.
Plotter0Close, Plotter1Close, Plotter2Close, Plotter3Close	Sent when the user closes a plotter window.
ProofAnimation	This message is used by the blocks that are interacting with Proof to produce proof animation functionality. See blocks in the Discrete Event, Manufacturing libraries, and the Proof library for information on how to set up your blocks to do Proof animation. Please see “Proof Animation (Windows only)” on page E278 for more details on adding Proof animation to your models.
ResumeSim	When the user either: • clicks dialog buttons during the simulation, or • edits a parameter and chooses Resume from the Run menu or clicks the Resume button in the simulation status bar. This allows the ModL code to respond to changes before continuing the simulation. Note: This message is only sent to those blocks that have had a dialog value changed or a button clicked on.
ShiftSchedule	Sent by a block to all the blocks in the model when a shift schedule has been changed.
Simulate	Every step of the simulation. Note that this message gets sent over and over, not just once. This is where most of the “action” in a block takes place. For instance, you check and change the value of connectors in this message handler. *In continuous simulations*, the first Simulate message gets sent with CurrentTime=StartTime and CurrentStep=0. For subsequent Simulate messages, Extend adds 1 to CurrentStep and adds DeltaTime to CurrentTime. The last Simulate message occurs when CurrentStep=NumSteps-1 and CurrentTime=EndTime, so each block gets NumSteps Simulate messages. *In discrete event simulations*, the first Simulate message gets sent with CurrentTime = StartTime and CurrentStep = 0. For subsequent Simulate messages, the Executive block controls how CurrentTime is advanced.

Message	When sent
StepSize	After all CheckData messages. If your block's code has to have a particular StepSize, you should set it here. Set the DeltaTime system variable to your maximum tolerable step size. Extend queries all blocks and uses the smallest DeltaTime value returned. This message is also used to setup the attribute global arrays in a discrete event model.
TabSwitch	This message is sent to a block when the block's dialog is switched from one tab to another. This will also be sent when the dialog is first opened, as that is basically treated as a click on the last tab that was previously opened.
TimerTick	Sent by the Timer chore started by the StartTimer function. (See "Date and time" on page P165.)
UpdateStatistics	When a block's statistical variables need to be recalculated and updated. This message is typically sent by a block in the Statistics Library. See the Activity, Delay block in the Discrete Event library for an example of receiving this message.
UserMsg0 through 9	User defined message. When a block (connected or unconnected) receives a message from another block using the SendMsgToBlock function.
YourButton	If you have created a button, radio button, or check box named *YourButton*, this message handler is activated when the button is clicked.
YourItem	Whenever the selection in a dialog is in a parameter or editable text dialog item and you click another item or press the Tab key (taking the selection out of the item), the message handler with that dialog item's name is invoked. This is useful if you want to check the value of the item after it might have been changed.

The format of a message handler is:

```
on MessageName
{
local variable declarations
zero or more statements;
}
```

The MessageName must be one of the system message names or a dialog message name. You exit from a message handler with a RETURN statement or an ABORT statement, as described above.

System variables

System variables give you information about the state of the simulation. You can read or write to these variables, but you should be very careful when writing to any of them.

Table of system variables

Name	Description
AnimationOn	Tells the state of the Show Animation command from the Run menu. If it is checked, AnimationOn is TRUE (1), otherwise it is FALSE (0). Note that closed hierarchical blocks always see a value of FALSE until they are opened. This feature speeds simulations by preventing needless animation if the user can't see anything.
AntitheticRandomVariates	If TRUE, Extend's random number functions generate antithetic random numbers.
CurrentSim	Current simulation number. Its value starts at 0 and increments each step up to NumSims-1. It only has a positive value if you set the "Number of runs" option in the Simulation Setup or Sensitivity Setup dialogs to a value greater than 1. CurrentSim has a value of -1 when there is no simulation running.
CurrentSense	Used by the sensitivity analysis feature to determine the number of the simulation run for changing sensitivity variables. It is initially set to 0 and is incremented by 1 during each simulation, after ENDSIM. However, you may choose to change CurrentSense to any value you want for whatever reason during ENDSIM: Extend will simply increment it after ENDSIM and use that for its variable calculations. You should not change CurrentSim, and you can always refer to that variable to find the actual simulation number.
CurrentStep	Current step number. In a continuous simulation, its value starts at 0 and increments each step up to NumSteps. In a discrete event simulation, the value starts at 0 and increments with each event.
CurrentTime	Current time during the simulation. In a continuous simulation, its value starts at StartTime and increments at DeltaTime for each step in the simulation. In a discrete event simulation, the Executive block in the Discrete Event library changes the CurrentTime system variable only when processing an event.
DeltaTime	Time increment per step. This value is initialized by the Simulation Setup dialog and represents the basic increment of time used in the simulation. DeltaTime has no meaning in a discrete event simulation.
EndTime	Ending time for the simulation specified in the Simulation Setup dialog. This is the time at which the simulation ends unless a block stops the simulation with an abort statement or a discrete event simulation runs out of items or events.
MaxAttribs	This variable replaces the constant previously declared in the Discrete Event blocks. It allows a dynamic number of attributes to be set.

Name	Description
ModernRandom	Tells the state of the random number. If the current random number generator is being used, ModernRandom is 1. If the previous version of the random number generator is being used (for backwards compatibility), ModernRandom is 0 (see "Random numbers" in the main Extend User's Guide for a discussion on Extend's random number generator).
MovieOn	Tells the state of the Show Movies command from the Run menu. If that command is checked, MovieOn is 1, otherwise it is 0.
NumSims	Number of times the simulation will be repeated, as specified in the Simulation Setup or Sensitivity Setup dialogs.
NumSteps	The total number of steps that will be executed during a continuous simulation. NumSteps is the number of steps entered in the Simulation Setup dialog. NumSteps has no meaning in a discrete event simulation.
RandomSeed	Sequence number used to initialize the random number generator; it is set in the Simulation Setup or Sensitivity Setup dialogs. When debugging a simulation, it is sometimes necessary to force the random number generator to produce a repeatable sequence of pseudo-random numbers.
SimDelay	Tells the method of simulation order: 0 for Left to right, 2 for Flow order, 3 for custom order.
SimMode	0 for manual mode (deltaTime or numSteps values as entered in the Run Setup dialog), 1 for autostep fast (use entered values unless model calculates smaller deltaTime), and 2 for autostep slow (divide calculated deltaTime by 5), as specified in the Simulation Setup dialog.
StartTime	Starting time of the simulation at step 0. It is initialized in the Simulation Setup dialog.

Global variables

There are three sets of global variables that you can use anywhere in your model; their names start with "Global". These are useful for passing information between your blocks. Extend never changes the values of these variables except to initialize them when Extend is started, so you can use them anyway you want. Global variables are stored within the Extend application and are not associated with any particular model, therefore every block in any open model has access to them.

There is another set of globals that are controlled by the libraries that are included with Extend; their names start with "SysGlobal" and they are reserved for use with the libraries that come with Extend.

Name	Type	Use
Global0 through Global19	Real	General
GlobalInt0 through GlobalInt9	Integer	General
GlobalStr0 through GlobalStr9	String	General
SysGlobal0 through SysGlobal19	Real	Reserved
SysGlobalInt0 through SysGlobalInt29	Integer	Reserved
SysGlobalStr0 through SysGlobalStr9	String	Reserved

The SysGlobal variables are reserved by Imagine That, Inc. You can use the SysGlobal variables (for example when you create your own discrete event blocks), but you should only use them in the same way they are used in the blocks that come with Extend. Never use these variables for your own purposes; always use the general use globals instead.

Global arrays

Global arrays provide a repository, or internal database, for model-specific data. Global arrays are dynamic arrays that, like global variables, can be accessed by any block in the model. However, they differ from global variables in the following ways:

- Global arrays are stored with the model. Therefore, if you have two different models open, each will have its own set of global arrays. By default, the global arrays are saved with the model, so that the arrays are intact when the model is re-opened.
- Global arrays are accessed and managed through a suite of functions (see “Global arrays” on page P151).
- You create and dispose global arrays as they are needed. There is no limit to how many global arrays can be associated with a given model.

See “Working with global arrays” on page P218 for an example of creating, accessing, and disposing of global arrays.

Constants

ModL includes four general purpose predefined constants:

```
PI = 3.14159265358;
BLANK = (noValue);
TRUE = 1;
FALSE = 0;
```

Setting a dialog parameter to “blank” will make it blank. Blank values will show nothing in text in a dialog and are NoValues for all math calculations. Do not compare the constant BLANK with a value to determine if the value is BLANK (that is, NoValue). The result will always be FALSE. Use the NoValue function instead. This function returns a TRUE if the value passed to it is a NoValue.

There are other predefined constants that are specific to functions, such as the pattern constants used with the Animation functions.

ModL function overview

The rest of this chapter is a description of all the functions in ModL, listed by type. Within types, the functions are grouped by category. Within categories, the functions are listed alphabetically.

Function Type	Page	Categories
Math	P85	Basic math, Trigonometry, Complex numbers, Statistical and random distributions, Financial, Integration, Matrices, Bit handling, Equations
I/O	P98	File I/O (formatted), File I/O (unformatted), Interprocess Communication, OLE, Mailslot, ODBC, Serial I/O, Other drivers, XCMDs, DLLs, Alerts and prompts, User inputs
Animation	P123	Animation
Blocks and inter-block communications	P128	Block numbers/labels/names/type/position, Block connection information, Dialog items from other blocks, Block dialogs (opening and closing), Block data tables, Messages to blocks (sending and receiving)
Reporting	P142	Reporting
Plotting	P142	General plotting, scatter plots
Arrays/Queues/Delays	P149	Dynamic arrays, Passing arrays, Global arrays, Queues, Delay lines
Scripting	P157	Scripting
Miscellaneous	P162	Strings, Attributes, Date and time, Time units, Debugging, Help, Models and notebooks, Platforms and versions

ModL functions are also listed in the Help command alphabetically and by type, including their arguments. You can copy a function from there to use in your code.

All ModL functions expect their arguments to be the data type specified in the function definition. If you use another type, ModL will automatically convert the argument to the expected type before the function is called. Throughout this chapter, the following are used as arguments:

```
real      x, y
integer   i, j, n
```

For the functions that take no arguments, the parentheses are still required.

All functions return values that are real, integer, or string; some functions are procedures that do not return values. The type of value returned is indicated in the third column of the tables below as:

Return	Meaning
R	Real
I	Integer
S	String
P	Procedure (no value returned)

Math functions

Basic math

These functions perform numerical operations on their arguments.

Basic Math	Description	Returns
Ceil(real x)	Nearest integral real with a value greater than *x*. For example: Ceil(1.3) returns 2.0, and Ceil(-1.3) returns -1.0	R
Exp(real x)	e^x. e is approximated to 20 digits.	R
FFT(real array[n][2], inverse)	Replaces the array with the real and imaginary parts of the fast Fourier transform (see below).	P
FixDecimal (real value, integer fixFigs)	Sets the number of figures after the decimal point to fixFigs and returns the result.	R
Floor(real x)	Nearest integral real with a value less than *x*. For example: Floor(1.3) returns 1, and floor(-1.3) returns -2.	R
GammaFunction(real x)	Gamma function for *x*. Do not confuse this with the Gamma distribution function.	R
Int(real x)	Nearest integer after rounding toward 0. For example: Int(1.3) returns 1, and Int(-1.3) returns -1.	I
Integerabs(i)	Non-negative integer containing the absolute value of the integer *i*.	I
Log(real x)	Natural (base e) log of *x*.	R
Log10(real x)	Base 10 log of *x*.	R
Max2(real x, real y)	Maximum of the two arguments.	R
Min2(real x, real y)	Minimum of the two arguments.	R
NoValue(real x)	1 (True) if *x* has no value (is blank) or 0 (False) if a value has been assigned to *x*.	I

Basic Math	Description	Returns
Pow(real x, real y)	*x^y*. The same results may be achieved with the ^ operator, as in *x^y*. Note that Pow(0,0) is undefined.	R
Realabs(real x)	Non-negative real number containing the absolute value of *x*.	R
Realmod(real x, real y)	*x* modulo *y* (that is, the remainder of *x* divided by *y*).	R
Round(real x, integer sigFigs)	Rounds *x* to the significant figures specified in the *sigFigs* argument.	R
Sqrt(real x)	Square root of *x*.	R

The FFT function replaces the real array argument with the real and imaginary parts of the FFT. n must be a power of 2. If inverse is TRUE, the inverse FFT is calculated. The real values are contained in array[n][0], and the imaginary values are contained in array[n][1] (these two are denoted together as "array[n][i]"). array[0][i] is zero frequency. array [(n/2)-1][i] is the most positive and most negative frequency. array[n-1][i] is the negative frequency just below 0. See the "four1" routine in Press, *Numerical Recipes in C*, for more information on this algorithm.

Trigonometry

These functions assume that the argument represents an angle in radians.

Trig	Description	Returns
Acos(real x)	Arccosine of *x*, where *x* is any real number between -1 and +1 inclusive.	R
Asin(real x)	Arcsine of *x*, where *x* is any real number between -1 and +1 inclusive.	R
Atan(real x)	Arctangent of *x*, where *x* is a real number. The value returned is between -π/2 and π/2 radians.	R
Atan2(real y, real x)	Arctangent of *y/x*, where x is non-zero. The value returned is between -π and π radians.	R
Cos(real x)	Cosine of angle *x*.	R
Cosh(real x)	Hyperbolic cosine of angle *x*.	R
Sin(real x)	Sine of angle *x*.	R
Sinh(real x)	Hyperbolic sine of angle *x*.	R
Tan(real x)	Tangent of angle *x*.	R
Tanh(real x)	Hyperbolic tangent of angle *x*.	R

Complex numbers

These functions operate on complex numbers composed of two-element real arrays (U, Z, and result), where U[0] is the real part and U[1] is the imaginary part. All arguments and results are complex numbers expressed as two-element real arrays:

```
real U[2], Z[2], result[2];
```

Complex numbers	Description	Returns
AddC(result, U, Z)	result = *U+Z*	P
DivC(result, U, Z)	result = *U/Z*	P
MultC(result, U, Z)	result = *U*Z*	P
SubC(result, U, Z)	result = *U-Z*	P

Statistics and random distributions

These functions can be used both to generate random inputs and to gather statistical information from the results of simulations. In the following functions, the following are used as arguments:

```
real prob, rate, mean, stdDev;
integer nTrials, kthEvent, kthSuccess;
```

Statistics/ distributions	Description	Returns
DBinomial(real prob, integer nTrials)	Number of successes out of *nTrials*, each with a probability of success of *prob*.	R
DExponential(real rate)	Interval between events. *rate* is the expected (mean) number of events per period.	R
DGamma(integer kthEvent)	Waiting time to the *kthEvent* in a Poisson process of mean equal to 1.	R
DLogNormal(real mean, real stdDev)	Positively skewed distribution.	R
DPascal(real prob, integer kthSuccess)	Geometric distribution if *kthSuccess* equals 1. This returns the number of trials needed for the *kthSuccess* of an event with probability of *prob*.	R
DPoisson(real rate)	Number of times an event occurs within a given period. *rate* is the expected (mean) number of events per period.	R
Gaussian(real mean, real stdDev)	Real random member of a Gaussian (normal) distribution with the specified *mean* and standard deviation.	R

Statistics/ distributions	**Description**	**Returns**
Mean(real array[], integer i)	Arithmetic mean of the first *i* members of the single-dimensional array.	R
Random(real i)	Uniform pseudo-random integer in the range 0 to *i*-1 using the random seed specified in the Simulation Setup dialog. For example, Random(6) returns an integer in the range 0 through 5, inclusive. Random(i) assumes that *i* is an integer. For the Integer (uniform) function.	I
RandomCalculate (integer distribution, real arg1, real arg2, real arg3)	Returns a random number given a distribution and up to three arguments. If a given distribution does not use all three arguments, a zero should be entered for the unused argument(s). The following numbers are used to define the distribution: BETA 1 BINOMIAL 2 ERLANG 3 EXPONENTIAL 4 GAMMA 5 GEOMETRIC 6 HYPER EXPONENTIAL 7 INTEGER UNIFORM 8 LOGLOGISTIC 9 LOGNORMAL 10 NEGATIVE BINOMIAL 11 NORMAL 12 PEARSONV 13 PEARSONVI 14 POISSON 15 REAL UNIFORM 16 TRIANGULAR 17 WEIBULL 18	R

Statistics/ distributions	**Description**	**Returns**
RandomCheck-Param (integer distribution, real arg1, real arg2, real arg3, integer reportError)	Checks that the arguments passed in are valid for the given distribution. The distribution argument is defined using the numbers above. Displays an error message if reportError is TRUE and the arguments are not valid. If reportError is FALSE, the function returns the following: 0 successful -1 Mean must be greater than 0 -2 Probability must be between 0 and 1 -3 argument must be between 0 and 1 -4 Shape must be greater than 0 -5 Shape2 must be greater than 0 -6 Most likely value must be between Max and Min values -7 Min must be less than Max	I
RandomGetModel-SeedUsed()	The actual seed used for the model. If the Run Setup dialog had 0 entered for the seed, this will return the actual randomized seed used, not 0. If a non-zero number was entered, this will return that number. This is different than reading the RANDOMSEED global variable, which just shows the actual number entered in the Run Setup dialog, including 0, and doesn't show the actual randomized seed used for running the model.	I
RandomGetSeed()	Current seed value or current state of the random number generator. Used for saving and restoring the random state when using different seeds. See RandomSetSeed() below.	I
RandomReal()	Uniform pseudo-random real number x, in the range {0.0 <= x <1.0} using the random seed specified in the Simulation Setup dialog.	R
RandomSet-Seed(integer i)	Sets the seed value or saved state of the random number generator. Used for saving and restoring the random state when using different seeds. See RandomGetSeed() above.	P
StdDevPop(real array[], integer i)	Population standard deviation of the first *i* members of the single-dimensional array.	R
StdDevSample(real array, integer i)	Sample standard deviation of the first *i* members of the single-dimensional array.	R
TStatisticValue(real probability, integer degreesOfFreedom)	Returns an accurate approximation of the point on the students t-distribution for a given single tail *probability* and number of *degreesOfFreedom*	R

Statistics/ distributions	Description	Returns
UseRandomized-Seed()	Forces a randomized seed for the current simulation run, overriding any fixed seed entered into the Run Setup dialog. Must be called in CHECKDATA message handler.	P

See “Random numbers” in the main Extend User’s Guide for information about how Extend generates random numbers.

Financial

These functions calculate the unknown parameter in loan and annuity calculations given four known parameters. The financial functions use standard financial arguments:

Argument	Meaning
pv	Present value of a cash-flow (real)
fv	Future value of a cash-flow (real)
pmt	Amount of one payment (real)
ratePer	Interest rate per period (real)
nPer	Number of periods (integer)

pv, fv, and pmt treat cash received as a positive value and cash paid out as a negative value.

Note that ratePer must match the length of the periods indicated by nPer. For example, if nPer is the number of months, ratePer is the interest per month.

payAtBegin is a flag variable. If payAtBegin is TRUE (non-zero), payments occur at the beginning of the period. If FALSE (0), payments occur at the end of the period.

Financials	Description	Returns
CalcFV(ratePer, nPer, pmt, pv, payAtBegin)	Future value	R
CalcNPER(ratePer, pmt, pv, fv, payAtBegin)	Number of periods	R
CalcPMT(ratePer, nPer, pv, fv, payAtBegin)	Payment	R

Financials	Description	Returns
CalcPV(ratePer, nPer, pmt, fv, payAtBegin)	Present value	R
CalcRate(nPer, pmt, pv, fv, payAtBegin)	Interest rate. If the rate cannot be calculated using the values of the arguments, the function returns a noValue (BLANK) result.	R

Integration

These functions integrate a stream of values. For instance, you may want to integrate values coming from inputs during a simulation. In the integration functions, the array used in the integration calculations must be declared a static array of four real values:

```
real array[4];
real initConditions, inputValue, deltaTime;
```

Integration	Description	Returns
IntegrateEuler(real array[4], real inputValue, real deltaTime)	Value of an Euler integration in progress. The array must be initialized with the IntegrateInit function. The algorithm is a backward Euler: out = out + *inputValue* * *DeltaTime*.	R
IntegrateInit(real array[4], real initConditions)	Initializes the array for integration. Call this function in the InitSim message handler if you are going to use the integration functions during a simulation. *initConditions* specifies the starting real value for the integration.	P
IntegrateTrap(real array[4], real inputValue, real deltaTime)	Value of a Trapezoidal integration in progress. The array must be initialized with the IntegrateInit function. The algorithm is a first-order trapezoid: out = out + *DeltaTime* * (previous*InputValue* + *inputValue*) / 2.	R

Matrices

Many intricate tasks can be written in just a few matrix operations. For example, solving simultaneous equations, curve fitting data, finding the roots of a polynomial, and coordinate transformations may all be done with matrices. For further information about matrices, see *Matrix Methods and Applications* by Groetsch and King, (Prentice Hall, 1988).

The matrices used by these functions are in the form `matrix[m][n]`, where m is the number of rows and n is the number of columns. Vectors are of the form `vector[n]`, where n is the number of elements in the vector.

The complex numbers returned by the Roots and EigenValues functions are in an array that is declared as `array[n][2]`. This makes array[k][0] the real part, and array[k][1] the imaginary part.

Complex versions of the matrix functions end in the letter C. All arguments to these complex functions are complex.

Declarations for matrices are as follows:

```
real matrix[rows][columns];        // real matrix;
                                   //   also matrixA, matrixB, matrixR
real vector[rows];                 // real vector; also values
real matrixC[rows][columns][2];    // complex;
                                   //   also matrixAC, matrixBC, matrixRC
real vectorC[rows][2];             // complex; also valuesC
real resultC[2];                   // a complex number
integer n, m;                      // dimensions
```

Rows and columns can be bigger than needed. Just specify desired rows and columns when calling functions.

Matrices	**Description**	**Returns**
ConjugateC(matrixRC, matrixAC, integer m, integer n)	Returns the conjugate values of *matrixAC* in *matrixRC*. Both *matrixAC* and *matrixRC* are m by n by 2 (complex) matrices.	P
Determinant(matrixA, integer m)	Value of the determinant of *matrixA*, which is an *m* by *m* matrix. If the matrix is singular, the function returns a NoValue.	R
DeterminantC(resultC, matrixAC, integer m)	Complex version of Determinant which returns its complex result in *resultC*. If the matrix is singular, the function returns a NoValue in *resultC*.	P
EigenValues(ValuesC, matrixA, integer m)	Eigenvalues of *matrixA* (*m* by *m*) are placed into the complex array *ValuesC*. *ValuesC* is of length m by 2 (complex). *matrixA* may be nonsymmetric. (This routine is based on the EISPACK method of creating a Hessenberg matrix and iterating that matrix into a diagonal matrix through similarity transformations). If the matrix is singular, the function returns TRUE.	I
Identity(matrixR, integer m)	Creates an *m* by *m matrixR* with 1s along the diagonal and 0s above and below the matrix diagonal.	P
IdentityC(matrixRC, integer m)	Complex version of Identity.	P

Matrices	Description	Returns
Inner(vectorA, vectorB, integer m)	Value of the inner or dot product of *vectorA* and *vectorB* of length m.	R
InnerC(resultC, vectorAC, vectorBC, integer m)	Complex version of Inner which returns its complex result in *resultC*.	P
LUdecomp(matrixR, matrixA, integer m)	Returns the LU decomposition of input *matrixA* in *matrixR*. If the matrix is singular, the function returns TRUE. Since LU factorization in LUdecomp permutes rows to obtain the best pivots, the LU matrix returned is only directly applicable to diagonally dominant matrices. The result of LUdecomp can be used in general once the permutation is accounted for.	I
LUdecompC(matrixRC, matrixAC, integer m)	Complex version of LUdecomp. If the matrix is singular, the function returns TRUE.	I
MatAdd(matrixR, matrixA, matrixB, integer m, integer n)	Returns in *matrixR* the addition of *matrixA* to *matrixB*. *m* is the number of rows, *n* is the number of columns.	P
MatAddC(matrixRC, matrixAC, matrixBC, integer m, integer n)	Complex version of MatAdd.	P
MatCopy(matrixR, matrixA, integer m, integer n)	Copies *matrixA* (of dimension m by n) into *matrixR*.	P
MatCopyC(matrixRC, matrixAC, integer m, integer n)	Complex version of MatCopy.	P
MatInvert(matrixR, matrixA, integer m)	*MatrixR* is the inverse of *matrixA*, which is an *m* by *m* square matrix. If the matrix is singular, the function returns TRUE.	I
MatInvertC(matrixRC, matrixAC, integer m)	Complex version of MatInvert. If the matrix is singular, the function returns TRUE.	I

Matrices	**Description**	**Returns**
MatMat-Prod(matrixR, matrixA, integer mA, integer nA, matrixB, integer mB, integer nB)	*MatrixR* (*mA* by *nB*) is created from the product of *matrixA* (*mA* by *nA*) and *matrixB* (*mB* by *nB*). Note that *nA* must equal *mB*.	P
MatMat-ProdC(matrixRC, matrixAC, integer mA, integer nA, matrixBC, integer mB, integer nB)	Complex version of MatMatProd.	P
MatScalar-Prod(matrixR, matrixA, integer m, integer n, B)	This matrix scalar product creates *matrixR* by multiplying *matrixA* by *B*. *MatrixA* is *m* by *n*, and *B* is a scalar (single number).	P
MatScalar-ProdC(matrixRC, matrixAC, integer m, integer n, BC)	Complex version of MatScalarProd.	P
MatSub(matrixR, matrixA, matrixB, integer m, integer n)	Returns in *matrixR* the difference of *matrixB* from *matrixA*. *m* is the number of rows, *n* is the number of columns.	P
Mat-SubC(matrixRC, matrixAC, matrixBC, integer m, integer n)	Complex version of MatSub.	P
MatVectorProd(vectorR, matrixA, integer m, integer n, vectorB)	Creates an *m* length vector (*vectorR*) by multiplying *matrixA* (*m* by *n*) by *vectorB* (*n*).	P
MatVector-ProdC(vectorRC, matrixAC, integer m, integer n, vectorBC)	Complex version of MatVectorProd.	P

Matrices	Description	Returns
Outer(matrixR, vectorA, integer m, vectorB, integer n)	Creates an *m* by *n* matrix (*matrixR*) from the product of *vectorA* and *vectorB*. *vectorA* is of length *m* and *vectorB* is of length *n*.	P
OuterC(matrixRC, vectorAC, integer m, vectorBC, integer n)	Complex version of Outer.	P
Roots(real values[][2], real p[], integer n)	Calculates the roots of polynomial *p* of order *n*. These roots are returned in the complex array values. The coefficients of the polynomial *p* are in an array beginning with the coefficient of the second highest power. The coefficient of the highest power is assumed to be 1. For example: p[] -> x^n + p[0]*x^(n-1)+ p[1]*x^(n-2) ...+p[n-1] Note that both values and p can be length n or greater. If the matrix of the polynomial is singular, the function returns TRUE.	I
Transpose(matrixR, matrixA, integer m, integer n)	Creates the transpose of *matrixA* in *matrixR*. The input matrix is *m* by *n* and the result matrix is *n* by *m*.	P
TransposeC(matrixRC, matrixAC, integer m, integer n)	Complex version of Transpose.	P

Bit handling

The bit-handling functions return the integer value result of the operation. In these functions, bit 0 is the most significant bit and bit 31 is the least significant bit of ModL's 32 bit integers.

Declarations for bit handling are as follows:

```
integer     bitNum, Count;
```

Bit handling	Description	Returns
BitAnd(integer i, integer j)	Bitwise AND of the two integers.	I
BitClr(integer i, integer bitNum)	Sets the bit numbered bitNum in *i* to 0 and returns the result.	I
BitNot(integer i)	Bitwise NOT of the integer.	I
BitOr(integer i, integer j)	Bitwise OR of the two integers.	I

Bit handling	Description	Returns
BitSet(integer i, integer bitNum)	Sets the bit numbered bitNum in *i* to 1 and returns the result.	I
BitShift(integer i, integer Count)	Shifts *i* by *Count* bits. If *Count* is positive, this shifts to the left (multiply *i* by 2^*Count*); if *Count* is negative, it shifts to the right (divide *i* by 2^*Count*). 0s are shifted in.	I
BitTst(integer i, integer bitNum)	TRUE if the bit numbered *bitNum* is set to 1, FALSE if *bitNum* is 0.	I
BitXor(integer i, integer j)	Bitwise Exclusive OR of the two integers.	I

Equations

These functions let you create a block in which you can enter equations built on ModL code. See the Equation block in the Generic library for an example of using these functions in building your own equation block.

See the EquationSetStatic() and EquationGetStatic() functions to use “remembered” values in your equations.

Note For cross-platform compatibility, if you build blocks that use the equation functions your code needs to detect if the model is being opened on a different platform. Therefore, in addition to your own specific tests, you should test dynArrayName in the CheckData message handler of any block that uses the equation functions:

```
On CheckData
{
if (getDimension(dynArrayName) == 0)
     EquationCompile(...);
...
}
```

Because Extend will detect a change of platform and will therefore dispose of dynArrayName, the code must check and recompile the equation again when the simulation is run. For a detailed example, see the Equation block in the Generic library.

Equations	Description	Returns
EquationCalculate(real input1, ..., real input 10, integer dynArrayName[])	Calculates an equation compiled by the EquationCompile function. The arguments can be integer or real values (such as input connectors). This function returns the real value assigned to the output variable. This function should be called whenever you need a new value from the equation, and is usually placed in the Simulate message handler.	R

Equations	Description	Returns
EquationCompile(string inputVarName1, ..., string inputVarName10, string output VarName, string equation, integer tabOrder, integer dynArrayName[])	Compiles an equation that is typed into an editable text item or string variable in a dialog. The variables *inputVarName1* through *inputVarName10*, outputVarName, and equation are all strings; *tabOrder* is an integer and *dynArrayName* is an integer dynamic array. Variable names used in the equation are defined in text items in the dialog or in string variables in the block. The user's equation can use any valid ModL function or statement, including defining new variables. The compiler outputs error messages and puts the insertion point at the error in the dialog item identified by *tabOrder* if it is a dialog editable text item. The compiler stores the machine code for the compiled equation in *dynArrayName*. This function should be called only when the equation or variable names are changed. Returns TRUE if there was an error in the equation.	I
EquationCalculate20(real input1, real input2, ... real input19, real input20, integer dynArrayName[])	This works the same as the EquationCalculate function, except it allows 20 inputs.	R
EquationCompile20(string inputVarName1, ..., string inputVarName20, string output VarName, string equation, string equation2, integer tabOrder, integer tabOrder2, integer dynArrayName[])	This works the same as the EquationCompile function, except it allows 20 inputs and two equation strings.	I
EquationGetStatic (integer index)	Used in an Equation type block to allow static values to be "remembered" and used in the equation. Need to define: `Real EquationStaticValues[100];` as a static variable at the top of the ModL script for the Equation block. The user calls this function with an index from 0 to 99 to get the correct static value from this array to use in their equation. Used with EquationSetStatic(), below.	R

Equations	Description	Returns
EquationSetStatic (integer index, real value)	Used in an Equation type block to allow static values to be "remembered" and used in the equation. Need to define: `Real EquationStaticValues[100];` as a static variable at the top of the ModL script for this Equation block. The user calls this function with an index from 0 to 99 to set the value in the static array. Then the user can call EquationGetStatic(index), above, to use that value in their equation.	P
ShowFunctionHelp(integer alpha)	Brings up Extend's help with a list of the functions and arguments available for the equation functions. If *alpha* is TRUE, brings up the alphabetical list of functions. If *alpha* is FALSE, brings up the "Functions by type" list.	P

I/O functions

File I/O, formatted

Use these functions to manipulate formatted text files. These functions let you perform the same tasks as the Import Data and Export Data commands in the File menu as well as lower-level file reading and writing. Text files produced with the output functions must be read in an application that reads text files such as a word processing or spreadsheet program.

In these functions, if the pathname used is the empty string (""), Extend prompts you with a standard open or save dialog. This allows you to determine the correct file name at the time of the simulation. File pathnames for specific files are specified as "volumeName : folder : folder : fileName" (*Macintosh)* or "driveLetter:\directory\directory\fileName" (*Windows*) with each level separated by colons or backslashes, as appropriate. If the volumeName and folder or directory names are left off of the pathname, the file name will come from the current folder or directory. Note that names of files and folders or directories can only be 31 characters long (*Macintosh* and *Windows 95 or NT)* or 8 characters long (*Windows 3.1*).

The Import and Export functions let you define the column delimiter (separator) character in the file to be read or written. In Extend, Excel, Word, and most other tabular data applications, tabs normally delimit columns, and CRs (*Macintosh*: carriage returns) or CRLFs (*Windows*: carriage returns, line feeds) always delimit rows.

The colDelim argument to these functions is a string which specifies the separator character:

""	The empty string (two quotation marks with nothing between them) indicates a tab character
","	A comma character

" "	A space character (multiple spaces are read as one space)
"(any character)"	The character specified will be used as a column separator

There are two sets of functions. The Import and Export functions are used with files that have numerical data; the ImportText and ExportText functions are used with files that have string data. The functions are:

File I/O (formatted)	**Description**	**Returns**
Export(string pathName, string userPrompt, string colDelim, real array[][], integer rows, integer columns)	Writes the contents of a one- or two-dimensional real array or data table to the file. *Rows* and *columns* specify the portion of the array to be written and are integers. The function assumes that columns are delimited by the *colDelim* string character. Single dimension arrays must be read as one column by *n* rows. The function returns the number of rows written to the file, or 0 if there is an error.	I
ExportText(string pathName, string userPrompt, string colDelim, string array[][], integer rows, integer columns)	Writes the contents of a one- or two-dimensional string array or text table to the file. *Rows* and *columns* specify the portion of the array to be written and are integers. The function assumes that columns are delimited by the *colDelim* string character. Single dimension arrays are treated as one column by n rows. The function returns the number of rows written to the file or 0 if there is an error.	I
Import(string pathName, string userPrompt, string colDelim, real array[][])	Reads the numerical data from the file into a one- or two-dimensional real array or data table, then returns the number of rows read (if an error occurs, it returns 0). The function assumes that columns are delimited by the *colDelim* string character. Single dimension arrays are read as one column by n rows. Note that you should initialize the array before using this function. Otherwise, any values beyond what was read from the file will have the old values of the array.	I
ImportText(string pathName, string userPrompt, string colDelim, string array[][])	Reads the string data from the file into a one- or two-dimensional string array or text table, then returns the number of rows read (if an error occurs, it returns 0). The function assumes that columns are delimited by the *colDelim* string character. Single dimension arrays are treated as one column by n rows. Note that you should initialize the array before using this function to prevent any values beyond what was read from the file from having the old values of the array.	I

File I/O, unformatted

These are lower level file I/O functions. You can have up to 200 text (.txt) or HTML (.htm) files open for general reading and writing. The files are specified in the functions with the integer fileNumber that is returned from the FileOpen and FileNew functions. The functions are:

File I/O (unformatted)	**Description**	**Returns**
CreateFolder(string pathName)	Creates a new directory from the *pathName*. For Macintosh, the *pathName* must use colons (:) to separate folder names, "myDrive:Extend:myNewFolder". For Windows, backslashes should be used, "myDrive:\Extend\myNewFolder". Returns FALSE if successful, TRUE if there was an error.	I
FileClose(integer fileNumber)	Closes the file when you are finished writing or reading data to or from the file. Files must be closed before they can be used as data in other applications. Call FileClose in the EndSim message handler to close files when the simulation is finished.	P
FileDelete(string pathname)	Deletes the file. Use this with caution because deleted files are not recoverable.	P
FileEndOfFile(integer fileNumber)	TRUE if the end of file has been reached during the most recent FileRead.	I
FileExists(string pathname)	Returns TRUE if the file exists	I
FileGetDelimiter (integer fileNumber)	Type of delimiter found after the most recent FileRead. Returns FALSE if a column delimiter (such as a tab character) was found after the data, or TRUE if a CR (*Macintosh*) or CRLF (*Windows*) row delimiter was found. Call this immediately after FileRead to find out whether a column delimiter, CR, or CRLF followed the data.	I
FileGetPathName (integer fileNumber)	Returns the file's path name.	S
FileIsOpen(string pathName)	Returns TRUE if the file described by the *pathName* is open.	I

File I/O (unformatted)	**Description**	**Returns**
FileNew(string pathname, string userPrompt)	Opens a new or existing text (.txt) or HTML (.htm) file for writing and returns a fileNumber. If the *pathName* is an empty string (""), Extend prompts for a file name, displaying the *userPrompt* string. If the pathname cannot be found, or the Cancel button has been clicked, FileNew returns FALSE (0). If the file is already open, it returns the file's fileNumber. Note that the FileNew function erases all information from an existing file. To append data to an existing file, use FileOpen. Call FileNew in the InitSim message handler to create files at the beginning of a simulation.	I
FileOpen(string pathname, string userPrompt)	Opens an existing text (.txt) or HTML (.htm) file for reading or writing, and returns a fileNumber for reference. If the *pathname* cannot be found, or the file is unreadable, or the Cancel button has been clicked, FileOpen returns FALSE. If the file is already open, it returns the file's fileNumber. If the file is written to after using File-Open, the new data is appended to the end of the file. Call File-Open in the InitSim message handler to open files at the beginning of a simulation. FileOpen works with aliases of a file. **NOTE:** If you call this function with the following strings (e.g. *.TXT) as the pathname, it will change the types of files that the Standard File Dialog will be looking for: **.TXT* - text files, **.DAT* - data files, **.ATF* - Proof trace file, **.LAY* - Proof layout file.	I
FileRead(integer fileNumber, string colDelim)	Reads and returns a string read from the file, up to *colDelim* (a column delimiter character) or to a CR (*Macintosh*) or CRLF (*Windows*) delimiter. To ignore the column delimiter, set *colDelim* to an unused character (i.e. "@"). Reading past the end of file causes an error message. You should test with FileEndOfFile before calling FileRead. See FileGetDelimiter(), above.	S
FileRewind(integer fileNumber)	Resets the file to its beginning so that it can be reread.	P
FileWrite(integer fileNumber, string s, string colDelim, tabCR)	Writes the string or value into the file. If a number is used for s, ModL will automatically convert the number to a string. If tabCR is FALSE, a column delimiter character is written to the file after s; if TRUE, a CR (*Macintosh*) or CRLF (*Windows*) delimiter is written after s.	P

File I/O (unformatted)	Description	Returns
GetDirectoryContents (string path, stringArray stringdata, longarray longdata)	This function takes two dynamic arrays as arguments, calls MakeArray() for them, and fills them with the names of all the files and subdirectories in the specified directory. The first array will contain the names of all the files/directories, and the second will contain a integer value that will be zero for a file, and one for a directory. It returns the number of row entries in the array. The pathname separator on a mac is a ":", on windows a "/".	I
StripLFs (integer strip)	Sets a flag in Extend that determines if the fileread functions will strip LF characters. This flag defaults to TRUE, so you should call StripLFs(FALSE) if you are finding meaningful LF characters are missing from your data.	P
StripPathIfLocal (String pathName)	Strips off the pathname if the file is in the same directory as the current model. For example, this can remove non-portable path names from a filename returned from FileOpen() function. If the file is in the same folder as the model, no pathname is needed.	S

Interprocess Communication (IPC)

Interprocess communication (IPC) provides a standard way in which one application can directly communicate with another. These functions allow Extend to act as a client application by connecting to and requesting data and services from a server application. The server application must support AppleEvents (*Macintosh*) or dynamic data exchange (DDE, *Windows*). The IPC functions start with "IPC". If you are trying to develop on both Windows and Macintosh using IPC, see "Cross-platform development" on page P243.

IPC & Publish/ Subscribe	Description	Returns
IPCAdvise(integer conversation, string item, integer blockNum, string dialogItem, integer rowStart, integer colStart, integer rowEnd, integer colEnd)	(*Windows* only) Starts a DDE Advise loop with the application that is the other side of the specified *conversation*. This function will return an advise loop id, which needs to be used in the IPCStopAdvise Function when the advise loop is to be terminated.	I
IPCCheckConversation(integer conversation)	Checks the validity of an IPC conversation on Windows. (It always returns TRUE on the Mac.) A TRUE value is returned if the conversation is valid.	I

IPC & Publish/ Subscribe	Description	Returns
IPCConnect(string serverName, string topic)	Initiates an IPC conversation between Extend and a server. The *serv-erName* argument is the DDE or AppleEvents name of the server you are trying to connect to. (This name needs to be in the format that is appropriate for the platform. For example, Excel on Win-dows expects “Excel”, on the Macintosh it expects “XCEL”.) The *topic* argument is typically the keyword “SYSTEM.” You can also use the name of the document you want to communicate with, if you want the conversation to target a specific model. If successfully connected, this function returns an integer value that is used in the other IPC functions as the conversation identifier. If unsuccessful, it returns a zero. Note: the IPCDisconnect function must be used with IPCConnect so that communication is terminated as soon as it is no longer required.	I
IPCDisconnect (integer conversation)	Disconnects the specified *conversation*. This function is necessary when using IPCConnect, and should be called immediately when communication is no longer required. Returns a zero if the discon-nection was successful.	I
IPCExecute(integer conversation, string executeData, string item)	Sends a command to be executed by the server in the specified *con-versation*. The *executeData* argument is the command to be sent. The *item* argument is currently not used and should be set to a blank string (""). This function returns a zero if successful, a -1 for a general error, a -2 for a time-out error, a -3 for an invalid connec-tion, and a -4 for an event not handled by the server.	I
IPCGetDocName()	Returns a string that contains the name of the last file opened with the IPCOpenfile() function. This is mostly used to return the name of a file that the user selected when an empty string ("") was passed into the IPCOpenfile function.	S
IPCLaunch(string appName, integer minimized)	Launches the application *appName* and minimizes it if *minimized* is TRUE.	I

IPC & Publish/ Subscribe	Description	Returns
IPCOpenFile (string fileName)	Opens the file (for example, a spreadsheet) in the Finder using AppleEvents (Macintosh) or DDE (Windows). This function is equivalent to the user double-clicking a file's icon: it causes both the file and the file's application to open. The single argument is a string which is the file name. Note that this function tells the System to open the file (that is, to launch the named application or document) and is not at all related to Extend's FileOpen function. This function will accept a blank string ("") in the *fileName* argument which will cause a File Open dialog to appear. The user is then prompted to select the file which is to be opened. The function returns zero if successful.	I
IPCPoke(integer conversation, string pokeData, string item)	Sends data to the server in the specified *conversation*. The *pokeData* argument is the data to be sent. The *item* argument indicates where the data is to be put. For example, in Excel the item argument could be "R1C1" indicating that the data is to be sent to the cell at row 1 column 1. Note that the syntax of the item argument is dependent on the server. This function returns a zero if successful.	I
IPCPokeArray(long conversation, string item, string delim, string array data)	(*Windows* only) Pokes an array of data. The data from the dynamic array data will be poked to the target application. The return value will be zero for success, and nonzero for failure.	I
IPCRequest(integer conversation, string item)	Returns data from the server in the specified *conversation*. The *item* argument indicates where the data is to be taken from. For example, in Excel the item argument could be "R1C1" indicating that the data is to be retrieved from the cell at row 1 column 1. Note that the syntax of the item argument is dependent on the server.	S
IPCRequestArray (long conversation, string item, string delim, string array data)	(*Windows* only) Requests an array of data. The dynamic array data will be filled with the results from the request. The return value will be the number of rows of data that were rerurned.	I

IPC & Publish/ Subscribe	Description	Returns
IPCSendCalcReceive(integer product, real sendValue, integer sendRow, integer sendCol, string funcName, integer receiveRow, integer receiveCol)	Sends *sendValue* to the *sendRow*, *sendCol* of a spreadsheet file that is already open. It then executes the macro called *funcName*, or just recalculates if *funcName* is an empty string. The function returns the value at *receiveRow*, *receiveCol*. *Product* specifies the spreadsheet you are communicating with: use 1 for Microsoft Excel or 2 for Lotus 123. See also the function IPCSpreadSheetName.	R
IPCSetTimeOut(integer timeOut)	Sets the Timeout value for the various IPC functions. This determines how long Extend will wait for the other application to respond to IPC requests. Values are in milliseconds. The default value is 10000. Putting a –1 into this parameter will request Async behavior.	P
IPCServerAsync (integer async)	If *async* is TRUE, sets a flag in Extend so that Extend will return from further Execute messages immediately instead of waiting for the Execute messages to complete. Useful when used in a Execute message from another application so that the other application can continue to do other tasks while Extend calculates.	P
IPCSpreadSheetName(string spreadSheetName)	Used to establish a default *spreadSheetName*, and is only used in conjunction with the IPCSendCalcReceive function. The IPCSendCalcReceive function does not take a spread sheet name as an argument, and Lotus 1-2-3 requires a specific spread sheet name for DDE communication on Windows.	P
IPCStopAdvise (integer conversation, integer adviseLoopID, integer block)	(*Windows* only) Stops the specified advise loop within the specified block.	I
UpdatePublishers()	Forces an update of publisher information, for all publishers in the model. Note that this departs from Apple's standard interface, where publishers are only updated when the model is saved. (*Macintosh only)*	P

OLE (*Windows only*)

These functions apply to two new types of interface objects that are supported in version 5.0 of Extend. They also control or communicate with other applications via OLE Automation. Both are types of OLE embedded objects, (and/or ActiveX controls,) that can be inserted into the Extend user interface in two different ways.

- You can insert an OLE object at the worksheet level with the Insert Object command in the edit menu. This command will create a new block-like object that represents an instance of the embedded object.
- You can also insert an object into a new type of dialog item called an embedded object. This gives the embedded object a dialog variable name that can be used in these functions.

The ModL functions below will allow you to access and communicate with either type of object. The OLE function calls that contain a *blockNumber* and *dialogItem* refer to an embedded object on either the worksheet or in a dialog. In the dialog case, both arguments need to be used. In the worksheet case, just the *blockNumber* is sufficient, and the *dialogItem* should be an empty string (e.g. ““).

When these functions are used for OLE Automation, you need to start with the OLECreateObject function. This function will create an OLE object of the type you specify, and return an IDispatch interface on that object. From then on you can use the OLEDispatch functions described below to call methods, or set and get properties on the objects.

Please note that what these functions return, and what effects they have is very largely dependent on the implementation of the OLE object/ActiveX Control that you are embedding.

Note There is a sample block in the ModL Tips library (modltips.lix on windows.) that shows the syntax of these commands. The block is called Embed, and can be used as an OLE scripting tutorial, or as a tool for looking at the methods and properties of a given embedded object.

OLE (*Windows only*)	**Description**	**Return**
OLEActivate(integer blockNumber, string dialogItem)	Activates the specified embedded object. *DialogItem* is the dialog variable name in quotes. Returns FALSE if successful.	I
OLEAddRef(integer interfacePtr)	Addrefs the interface specified. Returns the refcount.	I

OLE (*Windows only*)	**Description**	**Return**
OLECreateObject (string objectReference)	This function is the starting point for OLE Automation. It will create an OLE object, (Or provide an interface to an application if it is already running,) and return an Idispatch interface to that object that can be used with the OLEDispatch calls listed below to allow the user to control other applications via OLE Automation. The object reference string is the registry key associated with the object you wish to embed. (As an example, excel would be excel.application.)	I
OLEDeactivate (integer blockNumber, string dialogItem)	Deactivates the specified embedded object. *DialogItem* is the dialog variable name in quotes. Returns FALSE if successful.	I
OLEDispatch-GetHelpContext (long IDispatch-Handle, long dispID)	This uses an Idispatch handle, usually returned by the OLEDispatchResult, or OLECreateObject functions defined above. This handle will be the Dispatch interface to an object that is either associated with an embedded object on the worksheet, or in the block dialog, or has been created via OLE Automation in an remote application. Many embedded objects are simple objects that will not have methods that return Idispatch handles on other objects, but some, like an embedded Excel worksheet, for example, contain 'Subobjects' that will need to be referenced in this way. See OLEDispatchResult() and OLECreateObject(). Same as OLEGetHelpContext(), except uses a handle.	I
OLEDispatchGet-Names (long IDispatchHandle, Str31 names[], integer dispID)	See OLEDispatchGetHelpContext(). Same as OLEGetNames(), except uses a handle.	I
OLEDispatchInvoke (long IdispatchHandle, long dispID)	See OLEDispatchGetHelpContext(). Same as OLEInvoke(), except uses a handle.	I
OLEDispatchPropertyGet (long IdispatchHandle, long dispID)	See OLEDispatchGetHelpContext(). Same as OLEPropertyGet(), except uses a handle.	I

OLE (*Windows only*)	**Description**	**Return**
OLEDispatchPropertyPut (long IdispatchHandle, long dispID)	See OLEDispatchGetHelpContext(). Same as OLEPropertyPut(), except uses a handle.	I
OLEDispatchResult ()	Returns an IdispatchHandle from the last Invoke call. (If a return value is available.) This handle can be used in the other OLEDispatch calls listed above. This handle will be the Dispatch interface to an object that is associated with an embedded object on the worksheet, or in the block dialog. Many embedded objects are simple objects that will not have methods that return Idispatch handles on other objects, but some, like an embedded Excel worksheet, for example, contain 'Subobjects' that will need to be referenced in this way.	I
OLEGetDispatchName (integer blockNumber, string dialogItem, integer dispID)	*DialogItem* is the dialog variable name in quotes. Returns the name associated with the specified dispID.	S
OLEGetDispID (integer blockNumber, string dialogItem, string theName)	*DialogItem* is the dialog variable name in quotes. Returns the Dispatch ID for the function/variable theName.	I
OLEGetDoc(integer blockNumber, string dialogItem, string returnDoc[], integer dispID, integer which)	DialogItem is the dialog variable name in quotes. Returns the internal documentation from the type library in string array returnDoc. returnDoc will be resized as needed if the text is larger then a single string. This functions accesses text that is in the objects Type Library. What information is available, and whether or not any is, is object dependent. *Which* takes the following values. 0: name (DispID property/Method name) 1: doc (Any available Documentation on the DispID.) 2: file name (FileName of the help file associated with the dispID.)	I

OLE (*Windows only*)	**Description**	**Return**
OLEGetFuncInfo (integer blockNumber, string dialogItem, integer index, integer which)	*DialogItem* is the dialog variable name in quotes. Returns function information for the function specified by index. Implementation of this function is object specific, so your results will vary dependent on how the developers of the object have implemented it. if *which* is 0: INVOKEKIND returns 1 for INVOKE_FUNC returns 2 for INVOKE_PROPERTYGET returns 4 for INVOKE_PROPERTYPUT returns 8 for INVOKE_PROPERTYPUTREF if *which* is 1: cParams returns Count of total number of parameters if *which* is 2 : cParamsOpt returns Count of optional parameters if *which* is 3 : memberID returns DispID. Note that *index* is not the same as the dispID. It is just a sequential index value from 0 to n-1 where n is the number of functions supported by the object.	I
OLEGetGUID()	Pops up the standard insert item dialog, and returns the GUID of the object you select as a string. The objects that appear on the standard insert item dialog are just those objects that have been defined as 'Insertable' in the registry, and will not necessarily include all of the objects that you can use with Extend.	S
OLEGetHelpContext (long blockNum, string dialogItem, long dispID)	Returns the help context value of the specified dispID.	I
OLEGetInterface (integer blockNumber, string dialogItem, integer whichInterface)	*DialogItem* is the dialog variable name in quotes. Returns a pointer to an interface on the object. The whichInterface argument currently only supports a zero value for the IDispatchInterface.	I

OLE (*Windows only*)	**Description**	**Return**
OLEGetNames (integer blockNumber, String dialogItem, Str31 names[], integer dispID)	*DialogItem* is the dialog variable name in quotes. Puts the name of the function/variable into the first row of the dynamic array *names*. The later rows contain the names of any arguments to the function. The return value is the number of names returned.	I
OLEInsertObject (integer blockNumber, string dialogItem, string guid, integer xPos, integer yPos)	*DialogItem* is the dialog variable name in quotes. Inserts an object into the indicated location. If you specify the dialog item, and the block number, it will be inserted into that dialog item, ignoring *xPos* and *yPos*. If the dialog item name is an empty string, the object will be inserted onto the active worksheet at pixel location *xPos* and *yPos*. If *xPos* and *yPos* are both -1, the item will be inserted at the "current" position on the worksheet (i.e. the last mouse click or the last created block position).	I
OLEInvoke(integer blockNumber, string dialogItem, integer dispID)	*DialogItem* is the dialog variable name in quotes. Invokes (calls) the method/variable specified by *dispID*. You must call Param functions to set up the arguments to the method. Returns a WIN API error code if it fails, zero if success.	I
OLELongParam (integer value)	Adds a integer *value* to the argument list for the next Invoke call. Note: arguments are listed in back to front order. Returns FALSE if successful.	I
OLELongResult()	Returns an integer value from the last Invoke call, if a return value is available.	I
OLEPropertyGet (integer blockNumber, string dialogItem, integer dispID)	DialogItem is the dialog variable name in quotes. Gets the property specified by dispID. You must call Result functions to retrieve the value.	I
OLEPropertyPut (integer blockNumber, string dialogItem, integer dispID)	DialogItem is the dialog variable name in quotes. Sets the property specified by dispID. You must call Param functions to set up the arguments to the method.	I
OLERealParam (Real value)	Adds a real *value* to the argument list for the next Invoke call. Note: arguments are listed in back to front order. Returns FALSE if successful.	I

OLE (*Windows only*)	**Description**	**Return**
OLERealResult()	Returns a real value from the last Invoke call, if a return value is available.	R
OLERelease(integer interfacePtr)	Releases the interface pointer specified. Returns the refCount.	I
OLEReleaseInterface (integer interfacePtr, integer whichInterface);	Releases the interface pointer returned by OLEGetInterface. Returns FALSE if successful.	I
OLEStringParam (string value)	Adds a string value to the argument list for the next Invoke call. Note: arguments are listed in back to front order. Returns FALSE if successful.	I
OLEStringResult()	Returns a string value from the last Invoke call, if a return value is available.	S
OLEVariantParam (string value)	Adds a variant pointer value to the argument list for the next invoke call. Note: arguments are listed in back to front order.	I
OLEVariantResult (long which)	Returns a string value from the specified variant pointer argument of the last Invoke call. Which specifies which of the arguments of the invoke call you are referring to, it would be one for the first one entered, two for the second, and so on. (If the specified argument was a variant pointer argument.)	S

Mailslot (*Windows only*)

Mailslots are a messaging functionality supported by the windows API. They are unidirectional messages that are sent from a given machine to a specified mailslot.

The MailSlotReceive message handler will be called periodically when there is a waiting message available in one of the mailslots created by these functions.

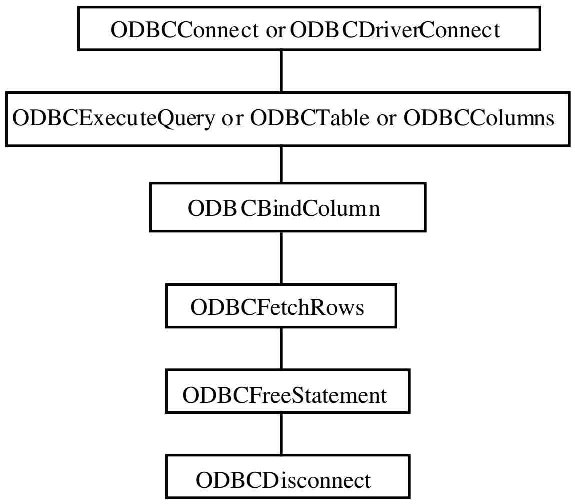
Note If you send out a single message to a mailslot, the mailslot may receive multiple copies of that message depending on your network configuration. This is a expected behavior of the microsoft implementation of mailslots. If you need to uniquely identify messages, the simplest way would be to concatenate a unique identifier onto the beginning or the ending of the message, and then ignore the duplicates.

See the Windows API documentation for more information about mailslots.

Mailslot (*Windows only*)	**Description**	**Return**
MailSlotClose(integer index)	Closes the specifed mailslot. Returns FALSE if successful.	I
MailSlotCreate (string theSlotName)	Creates a mailslot with the specified *name*. Returns the index number of the mailslot, or a zero if the call failed.	I
MailSlotRead(integer index)	Reads the next message from the specified mailslot. This function will return an empty string if there are no messages waiting.	S
MailSlotSend (string theComputerName, string theSlotName, string message)	Sends a *message* to the specified mailslot(s). *TheComputerName* field specifies exactly which machine to send the message to. If this field is a star "*", this message will be broadcast to all mailslots with the specified mailslot name in the primary domain of the sending computer. If this field is a domain name, the message will be sent to all mailslots with the specified name in that domain. The *SlotName* field must contain the name of the specified mailslot, and cannot be wildcarded. Returns FALSE if successful.	I

ODBC

ODBC stands for Open Database Connectivity. This microsoft API allows data connectivity between applications that support it, and databases that support it. The following functions allow MODL access to the ODBC API. The following diagram shows the basic sequence of commands that you would use to connect to a database, and retrieve data.


ODBCConnect or ODBCDriverConnect
ODBCExecuteQuery or ODBCTable or ODBCColumns
ODBCBindColumn
ODBCFetchRows
ODBCFreeStatement
ODBCDisconnect

ODBC sequence of operations

If you are trying to change data in the database, or add data, you could substitute the ODBCInsertRows, and/or ODBCSetRows calls for the ODBCExecuteQuery, ODBCBindColumn, and ODBCFetchRows combination. The information listed here is about the MODL implementation of functions that allow access to the ODBC API.

To use these functions effectively, you will need documentation on ODBC, SQL, and the database you are targeting as well as this documentation. The Microsoft documentation on ODBC can easily be found by doing an Exact Phrase search for "ODBC API Reference" on the MSDN Online Search site.

Note There is a sample block in the ModL Tips library (modltips.lix on windows.) that shows the syntax of these commands. The block is called odbc, and can be used as a ODBC scripting tutorial, or as a test of the ODBC functionality.

ODBC	Description	Return
ODBCBindColumn (integer statement, integer whichCol, string data[])	Associates an array with a column in the specific dataset. *WhichCol* defines which column of the dataset you wish to bind the array data with. The contents of that column will be different dependent on dataset, and how it was derived. This does not actually retrieve the data, just specifies where the data will go when it is retrieved. Returns FALSE (0) if unsuccessful. (See ODBCFetchRows below.)	I
ODBCColAttribute (integer statement, integer column, integer which)	Returns the value of the specified attribute of the specified column. This function just filters directly through to SQLColAttribute, check the ODBC documentation for additional information. This function can be called as soon as there is a defined dataset.	I
ODBCColumns (integer connection)	Executes a standard query that returns a dataset containing the names of all the valid columns in the data source. Returns a Statement Handle. (Note: it is very important to free all statements before disconnecting the connection, otherwise the disconnection may fail.) Returns FALSE (0) if the query fails. See your ODBC documentation (SQLColumns) for a list of the meanings of the columns of the resulting dataset.	I
ODBCConnect (string szDBName, string szUserName, string szPassword);	Connects to an ODBC source. Returns a connection Handle. (Note: it is very important to disconnect this connection before quitting Extend, otherwise a crash may result.) Returns FALSE (0) if the connection fails. You need to have created a valid ODBC Data Source to connect with before using this call.	I
ODBCConnectName ()	Returns the string name of the current connection.	S

ODBC	Description	Return
ODBCCountRows (integer connection, string tableName, string columnName, string whichCondition)	Uses the SQL COUNT statement to return the number of rows in the specified column of the specified table. The statement executed will be SELECT COUNT (*ColumnName*) FROM "tableName" WHERE *whichCondition*. *ColumnName* will default to '*' if it is blank. *WhichCondition* will specify a selection condition, if it is blank, all rows will be counted. See your SQL documentation for additional information about this query.	I
ODBCCreateTable (Integer connection, string tableName, stringarray columnNames, string columnTypes[])	This function will create a table in the specified data base with columns named as the values in the columnnames array, and types as specified in the columntypes array. Returns FALSE (0) if unsuccessful.	I
ODBCDriverConnect(string szConnectString)	Connects to an ODBC source, putting up a system source selection dialog. Returns a connection Handle. (Note: it is very important to disconnect this connection before quitting Extend, otherwise a crash may result.) Returns FALSE (0) if the connection fails. You need to have created a valid ODBC Data Source to connect with before using this call.	I
ODBCDisconnect (integer connection)	Disconnects the specified connection. Returns FALSE (0) if unsuccessful.	I
ODBCExecuteQuery (integer connection, string theQuery)	Executes the specified SQL query string. Returns a Statement Handle. (Note: it is very important to free all statements before disconnecting the connection, otherwise the disconnection may fail.) Returns FALSE (0) if the query fails.	I
ODBCFetchRows (integer statement)	Fetches the data from the dataset, and stores it in the variables that have been bound by ODBCBindColumn. Returns the number of rows. (See ODBCBindColumn above.)	I
ODBCFreeStatement (integer statement)	Frees the specified Statement Handle. Returns FALSE if successful.	I
ODBCInsertRow (Integer connection, string tableName, string columnNames[], string values[])	This Function add a row of data with the specified values to the indicated table. Returns FALSE (0) if unsuccessful.	I

ODBC	Description	Return
ODBCNumResult-Cols (integer state-ment)	Returns the number of resulting columns in the specified dataset. This function will only return useful information after an ODB-CExecuteQuery, ODBCTables, or ODBCColumns call has estab-lished a dataset.	I
ODBCSetRows (integer connec-tion, string table-Name, string columnName, string IDName, string dataArray[], string IDArray[])	Copies the data from the data array into the columnName array, using the IDArray values to determine which cell each item goes into. I.e. the variable named in the IDName field will be compared to the values in the IDArray array, and each row of the column-Name variable will be updated based on the database row selected by the values in the IDArray array. Returns FALSE (0) if unsuccess-ful.	I
ODBCSuccessInfo (integer ShowSuc-cessInfo)	Sets a flag that determines if warning error messages are shown, or not.	P
ODBCTables (inte-ger connection)	Executes a standard query that returns a dataset containing the names of all the valid tables in the data source. Returns a Statement Handle. (Note: it is very important to free all statements before dis-connecting the connection, otherwise the disconnection may fail.) Returns FALSE (0) if the query is unsuccessful. See your ODBC documentation (SQLTables) for a list of the meanings of the col-umns of the resulting dataset.	I

Serial I/O

These functions allow Extend to interact with serial ports. You can, for example, set up a simula-tion that will cause a modem attached to a computer to dial up a remote database, collect data, run a simulation, and send the results to another location over the modem. Likewise, an Extend simulation can be controlled by a modem from a remote location.

Macintosh: The port arguments are 0 for serial port A (modem port) or 1 for port B (printer port).

Windows: The port arguments are from 1 through 4 for COM1 through COM4, respectively.

Note If you use serial functions in your ModL code, and there is any possibility that your blocks will be used cross-platform (ie. on both Windows and Macintosh systems), your code should include checks to allow for the differences between platforms. For more information, see "Extensions and drivers" on page P277.

The SerialRead and SerialWrite functions are asynchronous, meaning that they return immediately without waiting for a response from the modem. SerialRead reads from the input buffer, not from the modem, and SerialWrite writes to the output buffer.

Serial I/O	Description	Returns
SerialRead(integer port)	Returns a string if there is any read data, or an empty string if data is not available. If there are more than 255 characters in the serial port buffer, it returns the first 255 characters. Successive SerialRead calls will return the rest of the characters in the buffer.	S
SerialReset(integer port, integer baud, real stop, integer parity, integer data, integer xonXoff)	Sets up one of the serial ports for communications. *baud* can be 300, 600, 1200, 2400, 4800, 9600, 19200, or 57600; *stop* can be 1, 1.5, or 2; *parity* is 0 for no parity, 1 for odd parity, or 2 for even parity; *data* can be 5, 6, 7, or 8 data bits. Set *xonXoff* to TRUE for XON/XOFF handshaking or FALSE for CTS handshaking.	P
SerialWrite(integer port, string s)	Writes the string *s* to the serial port buffer.	P

Other drivers (*Macintosh only)*

Driver functions allow Extend to communicate with external hardware and software systems. The drivers are stored in resource files you put in the Extend Extensions folder. These driver functions can be dangerous and can cause system errors, lost data, and erroneous results if used improperly. Before using them, consult the driver sections of *Inside Macintosh,* published by Addison-Wesley.

XCMDs (*Macintosh*) and DLLs (*Windows*) provide an effective and safer alternative to drivers. They are described in “XCMDs (Macintosh only)” on page P117 and “DLLs (Windows only)” on page P118.

Note If you use driver functions in your ModL code, and there is any possibility that your blocks will be used cross-platform (for example, on both Windows and Macintosh systems), your code should include checks to allow for the differences between platforms. For more information, see “Extensions and drivers” on page P277.

The buffers used to read and write bytes can be arrays of any type, larger than the number of bytes returned. There are 4 bytes per integer element, 8 (*Windows* and *PowerPC*) or 10 (*68K Macintosh*) bytes per real element, and 256 bytes per string. If a string array is used, the function only uses the first string of the array. The buffer needs to be an array, but in the case of a string, it can be an array with only one element.

In these functions, refNum is the device reference number returned by the OpenDriver function. Error codes returned are negative if an error occurred or 0 or positive if there was no error.

Other drivers	Description	Returns
CloseDriver(integer refNum)	Closes the driver and returns an error value.	P
DeviceControl(integer refNum, integer code, paramsArray)	Sends the control *Code* to the driver and returns an error number. *paramsArray* is any type of array which can have any purpose.	I
DeviceKillIO (integer refNum)	Stops any I/O operation to the device and returns an error number. Can also be specified as DevKillIO.	I
DeviceStatus (integer refNum, integer code, paramsArray)	Sends a status *Code* to the driver and returns an error number. *paramsArray* is any type of array which can have any purpose.	I
FSRead(integer refNum, integer count, bufferArray)	Number of bytes read from the device, or a negative error number if an error occurred, and places the bytes in any type of *bufferArray. Count* specifies the desired number of bytes to be read.	I
FSWrite(integer refNum, integer count, bufferArray)	Number of bytes written to the device, or a negative error number if an error has occurred, and writes the bytes from any type of *buffer-Array* to the device. *count* specifies the desired number of bytes to be written.	I
OpenDriver(string devName)	Opens the device in the *devName* string and returns a device reference number to be used in subsequent device driver calls. Returns 0 if the driver could not be opened.	I

XCMDs *(Macintosh only)*

Extend can interact with XCMDs and XFCNs through the following functions. The XCMDs are stored in resource files you put in the Extend Extensions folder. See “XCMDs and XFCNs” on page P237, for more information on XCMDs.

XCMDs are called by calling the XCMDParam function for each parameter of the XCMD. After all the XCMDParam functions, call the XCMD function. The parameters are not retained in memory after the XCMD call, so the parameters must be reestablished before you call the XCMD function again.

XFCNs are called exactly the same as XCMDs except that you change the type argument to 1.

If you select “Unsupported XCMD callbacks beep” in the Preferences command, Extend will notify you if an XCMD you use has callbacks that Extend doesn’t support. Extend supports the following callbacks: BOOLTOSTR; EXTTOSTR; LONGTOSTR; NUMTOSTR; PASTOZ-ERO; ZEROTOPAS; ZEROBYTES; STRINGLENGTH; STRINGMATCH; STRTOBOOL;

STRTOEXT; STRTOLONG; STRTONUM. Extend does not support the following: GETFIELDBYID; GETFIELDBYNAME; GETFIELDBYNUM; GETGLOBAL; NUMTOHEX; SETFIELDBYID; SETFIELDBYNAME; SETFIELDBYNUM; RETURNTOPAS; SCANTORETURN; SCANTOZERO; STRINGEQUAL. In addition, callbacks specific to Hypercard version 2.0 or greater are not supported.

Note If you use XCMD functions in your ModL code, and there is any possibility that your blocks will be used cross-platform (for example, on both Windows and Macintosh systems), your code should include checks to allow for the differences between platforms. For more information, see "Extensions and drivers" on page P277.

XCMDs	**Description**	**Returns**
XCMD(string XName, integer type)	Returns a string from the named XCMD. The *type* is 0 for a 68K native XCMD, 1 for a 68K native XFCN, 2 for a PPC native XCMD, and 3 for a PPC native XFCN. XName is a string.	S
XCMDArray(array)	Adds any type of array argument to the XCMD parameters. Note that arrays are not part of the XCMD specification and this function can only be used with custom XCMDs for Extend.	P
XCMD-Param(string arg)	Adds a string argument to the XCMD parameters.	P

DLLs (*Windows only*)

Extend can interact with DLLs through the following functions. The DLLs are stored in the Extensions subdirectory in the Extend directory. There are two sets of DLL functions:

- The first set is used for calling existing Windows DLLs. These DLL functions allow a variable argument list and have different calls based on the calling convention of the DLL.
- The second set is designed specifically for cross-platform compatibility with the XCMDs in the Macintosh version of Extend. This set of functions has a simpler interface, but is less flexible in calling convention and argument lists.

Working with DLLs will be a lot easier if you make note of the following:

- DLLs called by Extend must be built for 32 bit execution.
- Variables passed from the ModL code to a DLL are passed by value unless the variables are strings or arrays, in which case they are passed as pointers. Strings are passed to DLLs from ModL as Pascal strings, not C strings. See "DLLs" on page P238, for more information.
- If you use DLL functions in your ModL code, and there is any possibility that your blocks will be used cross-platform (for example, on both Windows and Macintosh systems), your code

should include checks to allow for the differences between platforms. For more information, see "Extensions and drivers" on page P277.

Also see "DLLs" on page P238 for more information about using DLLs and "Accessing code from other languages" on page P57 for a DLL example.

Existing Windows DLLs

These function calls are called with variable argument lists. The first argument in each case is the Proc-Address for the procedure. This is returned by the function DLLMakeProcInstance. It is recommended that you call DLLMakeProcInstance once (in On OpenModel or in On InitSim), and save the return value in a variable, rather then call it each time you call the specific function.

Note: It is critical that you match the argument list and the calling convention in the ModL code with the argument list and calling convention in the DLL, otherwise a crash could result.

DLLs (existing)	Description	Returns
DLLBoolCFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a Boolean (translated into a long by Extend). Accepts a variable argument list. Assumes a C calling convention.	I
DLLBoolPascalFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a Boolean (translated into a long by Extend). Accepts a variable argument list. Assumes a Pascal calling convention.	I
DLLBoolStdcallFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a Boolean (translated into a long by Extend). This function is the same as the other DLL functions except it assumes a StdCall calling convention.	I
DLLCtoPString (string theString)	Converts a C string to a P string that is useable by ModL, as Extend/ModL internally can use only P (pascal) strings. In some cases pre-established DLL's will expect and return C format strings in the passed parameter string pointer, and this function can convert the string type safely within the pointer space. See DLLPtoCString below to convert back to C strings.	P
DLLDoubleCFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a double. Accepts a variable argument list. Assumes a C calling convention.	R
DLLDoublePascalFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a double. Accepts a variable argument list. Assumes a Pascal calling convention.	R

DLLs (existing)	Description	Returns
DLLDoubleStdcall-Function(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a double. This function assumes a StdCall calling convention.	R
DLLLoadLibrary (string pathName)	Loads the specified library. This is for people who need to access routines in a DLL that is not present in the Extensions folder. After loading the library, attempts to access DLL routines that are in that library should succeed.	I
DLLLongCFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a long. Accepts a variable argument list. Assumes a C calling convention.	I
DLLLongPascal-Function(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a long. Accepts a variable argument list. Assumes a Pascal calling convention.	I
DLLPtoCString (string theString)	Converts a P string to a C string that is useable by a DLL, as Extend/ModL internally can use only P (pascal) strings. In some cases pre-established DLL's will expect and return C format strings in the passed parameter string pointer, and this function can convert the string type safely within the pointer space. See DLLCtoPString above to convert back to P strings.	P
DLLLongStdCall-Function(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a long. This function assumes a StdCall calling convention.	I
DLLMakeProcInstance(string procName)	Returns the ProcAddress expected by the other calls as an argument. Requires a procedure name. This function will search all open libraries for the named procedure, so it is advisable to call it once, and save the returned value. Function will return a zero if procName was not found.	I
DLLVoidCFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns nothing. Accepts a variable argument list. Assumes a Pascal calling convention.	P
DLLVoidPascal-Function(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns nothing. Accepts a variable argument list. Assumes a Pascal calling convention.	P
DLLVoidStdcall-Function(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns nothing. This function assumes a StdCall calling convention.	P

DLLs for compatibility
These functions are called by calling the DLLParam and DLLArray functions once for each parameter you want to pass to the DLL. This puts the parameter information into a structure which is passed to the DLL by the DLLXCMD function, which actually executes the DLL.

The parameters are not retained in memory after the DLLXCMD call, so the parameters must be reestablished before you call the DLLXCMD function again.

These function are equivalents of the XCMD, XCMDParam, and XCMDArray functions. They are designed so existing ModL code that calls XCMD functions does not need to be modified to access a DLL instead of an XCMD.

Note: The actual argument list and calling convention of the DLL are not variable. Code should be based on one of the DLLXCMD examples shipped with Extend so that these will be correct.

DLLs (compatibility)	**Description**	**Returns**
DLLXCMD(string procName, integer type)	Executes the procedure referred to by the *ProcName* and returns a string. Passes in the argument values established by prior calls to DLLParam and DLLArray. *Type* is ignored.	S
DLLArray(array)	Adds an array argument to the list of arguments.	P
DLLParam(string arg)	Adds a string argument to the list of arguments.	P

Alerts and prompts

These functions can be used to display results and diagnostics as well as to prompt for input data.

Alerts & Prompts	**Description**	**Returns**
Beep()	Causes the computer to beep using the beep sound selected in the Control Panel.	P
PlaySound(string soundName)	Plays the sound named in the argument. Returns FALSE if no error, TRUE if the sound is not found. See “Sounds” on page P240 for important details.	I
Speak(string s)	(*Macintosh only)* Speaks the string if the speech manager is present.	P
UserError(string s)	Opens a dialog with an OK button displaying the string *s*.	P

Alerts & Prompts	Description	Returns
UserParameter(string prompt, string default)	Opens a dialog to get a value. Displays the *prompt* string and *default* string, then returns either the entry typed, or the *default* string if there was no user entry. After getting a string with UserParameter, you can convert the string to a real with the StrToReal function described later in the section on strings.	S
UserPrompt(string s)	Opens a dialog with an OK and a CANCEL button displaying the string *s*. The function returns TRUE if the OK button is clicked and FALSE if the Cancel button is clicked.	I

Because of the automatic type conversion provided in ModL, and because they provide an easy way to display the contents of variables, these functions are also useful for debugging block code. For example:

```
UserError("X = "+x+", Y = "+y);
```

will display a dialog with something like:

```
X = 5.23, Y = .007
```

This is also an example of using the + operator for concatenating strings.

Also see other useful functions in “Debugging” on page P167.

User inputs

Use these functions to determine the location of mouse clicks and the status of modifier keys during the on blockClick and on dialogClick message handlers.

User inputs	Description	Returns
GetModifierKey (integer whichKey)	Returns a 1 if the key is depressed, a 0 if the key is not depressed. Can be used in the on blockClick and on dialogClick message handlers. *whichKey* 1 = Shift key *whichKey* 2 = Option (*Macintosh*) or Alt (*Windows)* key	I
GetMouseX()	Returns the mouse X position in pixels relative to the model worksheet. Use the GetBlockTypePosition() function to get the coordinates of the block. Can be used in the on blockClick message handler.	I
GetMouseY()	Returns the mouse Y position in pixels relative to the model worksheet. Use the GetBlockTypePosition() function to get the coordinates of the block. Can be used in the on BlockClick message handler.	I

User inputs	Description	Returns
Lastkeypressed()	The value returned will be the ASCII value of the chanracter entered for keys that have ASCII equivalents. For keys that don't have ASCII equivalents, the value returned is the keycode. This is useful for monitering what keys the user hits on the keyboard. Used in conjunction with startTimer (See "Date and time" on page P165), or during a simulation, it will allow live keyboard input.	I
WhichDialogItem-Clicked()	Used in the dialogClick message handler to find the name of the item that received the click. This is used, for example, to modify the items in a popup menu at the time it is clicked on but before it opens to the user.	S
WhichDT-CellClicked(integer rowCol)	Returns the row or column of the cell in the datatable that was clicked on. This function should only be used in the on dialogClick message handler, and usually after the whichDialogItemClicked function has determined that a specific data table was clicked on. *rowCol* 0 = row *rowCol* 1 = col	I

Animation

Use these functions to implement animation in a block. Examples of using these functions are shown in "Animation" on page P184.

To implement Proof animation in your blocks, please see the source code for blocks in the Discrete Event, Manufacturing, and Proof libraries.

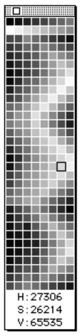
Note: Please see "Proof Animation (Windows only)" on page E278 for more details on adding Proof animation to your models.

Even if Show Animation from the Run menu is not selected, animation may still be available. The Show Animation command only controls whether animation is shown *during the simulation run* and then *only* during the repetitive execution of SIMULATE message handlers. At all other times, animation is always available, even if Show Animation is not selected. For instance, animation is available when the user makes any changes in a block's dialog, whether Show Animation is selected or not and regardless of whether the simulation is running. When Show Animation (in the Run menu) is selected, animation is available at all times.

In the functions, "obj" is the integer object number of the animation object on the icon. As discussed in "Animating hierarchical blocks" on page P187, hierarchical blocks are animated indirectly (by blocks in the submodel) by referencing the negative of the hierarchical block's object number. If a negative number is used and Extend doesn't find that object in the current enclosing HBlock, Extend will search the HBlock enclosing that one and so on, until it finds the object or reaches the top level. The outsideIcon value is a logical that is set to FALSE unless you want to move or stretch outside the original size of the icon (setting this to TRUE slows down the animation). All measurements are in pixels.

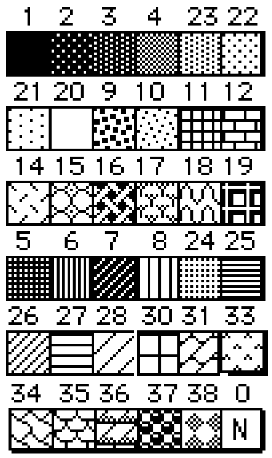
Color palette

AnimationColor specifies the color and pattern of the animated object. The color of an object is set with numbers for hue, saturation, and brightness (value), often called *HSV.* You can see the HSV values for any color on the bottom of Extend's color swatch window:



Color palette

The number for the patterns that appear in the pull-down pattern menu are:


1 2 3 4 23 22
21 20 9 10 11 12
14 15 16 17 18 19
5 6 7 8 24 25
26 27 28 30 31 33
34 35 36 37 38 0
N

Other pattern numbers

Animation	Description	Returns
AnimationBlockTo-Block (integer animationOb-jNum, integer blockNumFrom, integer conNum-From, integer block-NumTo, integer conNumTo, real speed)	This function moves an animation object across the connection lines from one block to another. You specify the sending block and connector, and the receiving block and connector, as well as the number of the animation object. The *speed* value is a relative speed factor. Use a value of 1.0 for normal speed.	P
AnimationColor (integer obj, integer hue, integer sat, integer bright, inte-ger pattern)	Sets the pattern and color of the object using the pattern number, hue, saturation, and brightness.	P
AnimationGetH-eight(integer obj, integer getOrig)	Returns the offset of the height of the object. If *getOrig* is true, gets the original height.	I
AnimationGetLeft (integer obj)	Returns the offset of the left side of the object relative to its original position in the icon.	I
AnimationGetTop (integer obj)	Returns the offset of the top side of the object relative to its original position in the icon.	I
AnimationGet-Width(integer obj, integer getOrig)	Returns the width of the object. If *getOrig* is true, gets the original width before any stretching.	I
AnimationHide (integer obj, integer outsideIcon)	Immediately hides the object.	P
AnimationLevel (integer obj, real level)	Defines the object as a level whose height varies from 0.0 to 1.0 (based on the *level* argument).	P
AnimationMoveTo (integer obj, integer leftOffset, integer TopOffset, integer outsideIcon)	Moves the object relative to its original position in the icon. Note: pixel coordinates start at the top/left corner of the animation object and go down and to the right. *TopOffset* of 0 is the top of the anima-tion object.	P

Animation	Description	Returns
AnimationMovie integer obj, string movieName, real speed, integer play-SoundTrack, integer async)	(Macintosh only) Defines the object as a QuickTime movie, played at the specified speed (relative to 1.0, the normal speed). If *play-SoundTrack* is true, the soundtrack is played. If *async* is true, the movie starts and the function returns immediately; if it is false, the function will not return until the movie is finished. The movie must be a file in the Extend Extensions folder or directory. Unlike other resources, the "movieName" is its file name, not a resource name.	P
AnimationMovieF-inish(integer obj)	(Macintosh only) Returns TRUE if the QuickTime movie is finished or not currently playing; returns FALSE if the movie is still playing. This is useful if you want to wait for a QuickTime movie to finish playing before you restart it. If you call AnimationMovie with async TRUE, use this function to check the status of the movie at any time.	I
AnimationObject-Exists(integer objNum)	Returns TRUE if an animation object with the specified *objNum* exists, and FALSE if it doesn't. For example, you could test if the enclosing hierarchical block has a specified animation object. (i.e. if animationObjectExists(-3) returns a TRUE value, then there is an animation object in the enclosing HBlock, or any number of levels above, with the objNum 3.)	I
AnimationOval (integer obj)	Defines the object as an oval.	P
AnimationPicture (integer obj, string picName, integer scaleToObj)	Defines the object as a picture. If TRUE, the *scaleToObj* argument scales the picture to the animation object size; if FALSE, the picture will not be scaled. See "Picture and movie files" on page P241 for important information about pictures.	P
AnimationPixel-Rect(integer obj, integer rows, integer cols, integer intArray[])	Defines the object as a rectangular pixel map of *rows* height and *cols* width, where each pixel can have a specified color. Extend normalizes the size of the pixels to conform to the animation object size. *IntArray* is declared as a dynamic array: `integer intArray[];`	P
AnimationPixelSet (integer obj, integer row, integer col, integer h, integer s, integer v, integer drawNow)	See AnimationPixelRect, above. Sets a specific pixel to an HSV color (see notes on color, above). *Row* values start from 0 to rows-1. *Col* values start from 0 to cols-1.	P
AnimationRectan-gle(integer obj)	Defines the object as a rectangle.	P

Animation	Description	Returns
AnimationRn-dRectangle(integer obj)	Defines the object as a rounded rectangle.	P
AnimationShow (integer obj)	Shows a hidden object.	P
Animation-StretchTo(integer obj, integer leftOff-set, integer topOff-set, integer width, integer height, inte-ger outsideIcon)	Stretches the object relative to its original position in the icon.	P
AnimationText (integer obj, string msg)	Defines the object as text with the string *msg*. Also see the Text-Width function in *Strings*, below, that is useful in applying the Ani-mationStretchTo function to the object so that the text width is accommodated.	P
AnimationTex-tAlign (integer objNum, integer justification)	This function specifies the text alignment for the specified anima-tion object. *Justification* takes the following values; Left 0 Right -1 Center 1	P
AnimationText-Transparent(integer objNum, String text)	Exactly the same as the animationText function, except the back-ground behind the text will be transparent. It's normally white for the regular function.	P
DialogPicture (String variable-Name, String pic-tureName, integer scalePicture)	This function displays the picture *pictureName* over the static text item *variableName*. This is used to display a picture on a dialog box. Normally you would create an empty static text item, and use that as the location for displaying this picture. *ScalePicture* will take a TRUE or FALSE, and determines if the picture is scaled to the space, or not. Returns FALSE if that picture is not available.	I

Animation	Description	Returns
PictureList (String15 arrayName[], integer type)	Resizes, and fills the dynamic array arrayname with the names of the pictures from the extension folder to be used with the Animation-BlockToBlock() function. Return value is the number of pictures. This function will ignore any pictures with a name that starts with a “@” character. This is to distinguish pictures that are to be used with the AnimationBlockToBlock() function from other pictures. *type*: 0-all AnimationBlockToBlock pictures 1-system AnimationBlockToBlock pictures 2-user AnimationBlockToBlock pictures	I
ProofEncode (String commandLine)	The version of the Proof DLL that ships with Extend requires numeric value calculated by this function to be passed to it periodically. See the Proof library to see how to use this function. Please see “Proof Animation (Windows only)” on page E278 for more details on adding Proof animation to your models.	I
ProofEncodeReset()	Resets counters for Proof copy protection.	P

Blocks and inter-block communications

Block numbers, labels, names, type, position

Blocks, text, and anchor points are numbered uniquely and sequentially from 0 to n-1 as they are added to the model window. The block numbers assigned to each item do not change. If a block or text is deleted, its number becomes an “unused slot” which is available when another block or text is added to the model.

There are two numbers associated with blocks. *Global numbers* access all blocks, including blocks inside of hierarchical blocks. *Local numbers* access a hierarchical block’s internal blocks and are the same for all instances of that hierarchical block, whereas the global numbers are different for each instance. Blocks are numbered from 0 to NumBlocks-1. Block numbers show in a dialog’s title bar.

Block names are the names you give a block when you create it. For example, “Add” is the name of a block in the Generic library. Names appear in the dialog’s title bar. *Block labels* are user-definable names given to a particular block in the model. Block labels are entered in the area to the right of the Help button in the bottom scroll bar of a block’s dialog. Labels appear at the bottom of the block’s icon.

Block type strings are the categories that appear in the library menu. Examples include *Math* or *Holding* from the Generic library.

Block numbers, etc.	**Description**	**Returns**
BlockName(integer i)	Looks in global slot *i* and returns either the name of the block, the first 255 characters of text, or an empty string if it is an unused slot. For example, you can use this function to read text information that you have added or modified in a model.	S
FindInHierarchy (string FindBlock-Name, string Find-BlockLabel, string FindDialogName, long FindDialog)	This function is used internally by several blocks, including the Catch and Throw blocks, to locate the corresponding block associated with the current block. (i.e. a catch for a throw, and vice versa.) Please see the catch and throw blocks for an example of how to use this function. The function returns the block number of the resulting block found, or a –1 if no matching block is found. FindBlockName is the name of the block to be found (i.e. 'Catch', or 'throw'.) FindBockLabel is the block label of the block to be found. FindDialogName is the name of the dialog item you wish to search for. FindDialog is a flag which is set to true for searching for the presence of a specific dialog item, and false for searching for a block label.	I
GetBlockLabel (integer i)	Returns the label string for the global block *i*.	S
GetBlockType (integer i)	Returns the block type string for the global block *i* (e.g. *Math* or *Holding*).	S
GetBlockTypePosition(integer i, integer intArray[4])	Returns the type (not the block type string as in GetBlockType() above) and position for the global block *i*. Types are as follows: 0 - empty slot; 1 - Anchor point; 2 - Text; 3 - Block; 4 - Hierarchical block Declare *intArray* as follows: **`integer intArray[4];`** On return, intarray[0] will contain the top, intarray[1] left, intarray[2] bottom, intarray[3] right position pixel values.	I
GetEnclosingH-BlockNum()	Returns the global block number for the enclosing hierarchical block. Returns -1 if there is no enclosing hierarchical block.	I
GetEnclosingHBlockNum2(integer block)	This is the same as GetEnclosingHBlockNum(), except it refers to a global block number.	I
LocalNumBlocks()	Number of internal blocks in the hierarchical block from which this function is called. Blocks are numbered from 0 to LocalNumBlocks()-1.	I
LocalNumBlocks2 (integer block)	This is same as LocalNumBlock(), except it refers to a global block number.	I

Block numbers, etc.	Description	Returns
LocalToGlobal (integer i)	Converts a local number in the hierarchical block from which this function is called to a global number.	I
LocalToGlobal2 (integer HBlock-Num, integer local-BlockNum)	Similar to LocalToGlobal, this function will return the global block-number for a local one. The difference is that this function allows you to specify which HBlock in the model is used as the context for the local block number via the first parameter HblockNum.	I
MakeOptimizer-Block (long true-False)	This function tags the block as an optimizer block so that the Run Optimization command will send a RUN message to that block, assuming there is a “Run" button in the block that will run the optimization and has an "ON RUN" message handler. Call this function in "CREATEBLOCK." See the Evolutionary Optimizer block in the Optimization library.	P
MyBlockNumber()	Global number of the block in which the function is called. Note that this number is the first number shown in parentheses in the block dialog’s title.	I
MyLocalBlock-Number()	Local number of the block in which the function is called. Note that this number is the second number shown in parentheses in the block dialog’s title.	I
NumBlocks()	Global number of blocks in the model, including text blocks and unused slots. Blocks are numbered from 0 to NumBlocks()-1.	I
SetBlockLabel (integer i, string str)	Sets the label for the global block *i* to *str*. Blocks are numbered from 0 to NumBlocks()-1.	P
ShowBlockLabel (integer i, integer show)	Labels are shown by default. To hide a block’s label, use this function with the global block number *i* and FALSE for *show*.	P

Block connection information

The following functions give you information about blocks and connections in a model. This helps you determine what is connected to a block or to create network lists of your models.

In these functions, “block” is the global block number (see “Block numbers, labels, names, type, position” on page P128), “connName” is the connector’s name (without quotes, not a string), “connString” is the connector’s name (with quotes), and “conn” is the ith connector (0 to n-1) in the list of connectors in the structure window’s connector pane. To determine the connector number for a given connector name, look in the block’s connector pane, counting from the top of the list (which is connector 0). “block” and “conn” are integers.

Block connection	Description	Returns
GetConnectedText-Block(integer block, integer conn)	Returns the block number of the named connection connected to the connector *conn* of block number *block*. Use the GetBlock-Name() function to retrieve the text of the named connection.	I
GetConnectedType (name connName)	Tells the type of connector connected to this connector. This is useful for determining what is connected to a universal connector. You can read the result by number or their associated Extend constants: 13 – ValueConnector; 14 – ItemConnector; 15 – UniversalConnector; 16 – DiamondConnector	I
GetConnectedType 2(integer block, integer conn)	This is similar to GetConnectedType(), except it refers to a global block number and *conn* is the ith connector (0 to n-1).	I
GetConBlocks (integer block, integer conn, integer intArray[][2])	Returns the number of blocks attached to the connector. On return, the rows of intArray are used to index the connected blocks (if there are three blocks connected, there are three rows). The first column of *intArray* contains the global block number of a connected block, the second column contains the connector number of that connected block. Rows and columns are indexed 0 to n-1. *IntArray* is declared as a dynamic array: **`integer intArray[][2];`**	I
GetConName (integer block, integer conn)	Name of the connector.	S
GetConNumber (integer blockNum, string conNameStr)	Returns the Connector number of the connector of the specified name	I
GetIndexedConValue(integer conn)	Gets the connector's current value. For blocks with many similar connectors, use this in a loop instead of lots of statements like "value[22]=con23In". The connectors are indexed from 0; the indexes are the same as the order of the connector names in the connectors pane.	R
GetIndexedConValue2(integer block, integer conn)	This is same as GetIndexedConValue(), except it refers to a global block number.	R
GetNumCons (integer block)	Number of connectors on the block.	I

Block connection	Description	Returns
MakeFeedback-Block(feedBack)	If *feedBack* is TRUE, this function causes Extend to terminate the flow order search for this block. For multiple feedback cases, this function prevents feedback from unexpectedly changing the main flow simulation order. For an example, see the Feedback block in the Utilities library.	P
SetIndexedConVa-lue(integer conn, real value)	Sets the connector's numerical value. For blocks with many similar connectors, use this in a loop instead of lots of statements like "con23Out=value[22]". The connectors are indexed from 0; the indexes are the same as the order of the connector names in the con-nectors pane.	P
SetIndexedConValu e2(integer block, integer conn, real value)	This is same as SetIndexedConValue(), except it refers to a global block number.	P

Dialog items

These functions work with dialog items of the block from which the functions are called. The following section includes functions that work with dialog items of other blocks.

Block dialog items	Description	Returns
AppendPopupLa-bels(String dia-logNameStr, String theLabels)	This function appends the specified string onto the labels associated with the named popup menu. Popup menu labels can be up to 2 strings long (255 + 255 = 510 characters). This function is the method for adding the second string onto the label. *DialogNameStr* is the dialog item name as a string or in quotes.	I
CreatePopupMenu (string string1, string string2, inte-ger initialSelection)	Creates a Popup at the last location the user clicked. This function can be used in conjunction with the on dialogClick message han-dler, the whichDialogItemClicked function, and the whichDT-CellClicked function to create a popup menu that appears on a dialog item, or datatable cell in response to a user's click. See the code of the on dialogClick messagehandler of the Activity, Delay block (Discrete Event library) for an example of how to use this function.	I
DialogItemVisible (long blockNum-ber, string dialogItemName, long clones)	Returns a true value if the specified dialog item is visible. If the Clones value is a one (TRUE) this function checks to see if any clones of the specified item are visible, otherwise it checks to see if the primary item is visible.	I

Block dialog items	Description	Returns
DisableDialogItem (String dia-logNameStr, Inte-ger TRUEFALSE)	Disables, or enables the specified dialog item.	P
HideDialogItem (string dia-logNameStr, inte-ger hideShow)	Hides the named dialog item if *hideShow* is TRUE. *DialogNameStr* is the dialog item name as a string or in quotes.	P
PopupMenuArray (string theArray[], integer initial value)	This function creates a flying popup menu based on the strings in theArray. Other then the fact that an array of strings is passed in instead of two separate strings, it's exactly the same as the Create-PopupMenu function.	I
SetPopupLa-bels(string dia-logNameStr, string theLabels)	Sets the named popup menu items to *theLabels* string. The menu items should be separated by semicolons (;). This function is similar to SetDataTableLabels in that the changes made by this call are not permanent within a block's dialog. Changes made by this call are permanent, however, for cloned copies of popup labels. *Dia-logNameStr* is the dialog item name as a string or in quotes.	P

Dialog items from other blocks

These functions return and set dialog items in other blocks. They work with all dialog items. Each requires the global block number of the block in which the dialog item resides.

NOTE: The block controls (switch, slider, and meter,) can be set or changed via the Get/SetDia-logVariable functions or poke and request functionality just the same way other types of blocks can. In the case of the block meter, this is easy to recognize, as this control has a dialog, and there-fore all the pieces that you need for setting and getting the values are available. (Block number, and dialog item name.) In the case of the slider and the switch it's not so obvious, as these con-trols don't have dialogs, and therefore the dialog item name is not readily available.   The names for these dialog items follow the pattern of the meter, and therefore are as follows:

- Switch: value – This is whether the switch is on, or off, and just takes a value of 0, or 1.
- Slider: value – The current value of the slider. (The position of the thumb.)
    - lower – The value of the lower bound of the slider.
    - upper – The value of the upper bound of the slider.

Block dialog items	Description	Returns
GetDialogNames (integer block, string nameArray, integer typeArray)	Returns a list of the dialog variables in the specified *block*. The *nameArray* argument needs to be a dynamic string array declared as *string nameArray[]*. The *typeArray* argument needs to be a dynamic integer array declared as *integer typeArray[]*. This function returns the number of items in the target block's dialog. For each item in the block's dialog, the string array will contain the name of the dialog item and the type array will contain the type of dialog item. Values for the type of array are as follows: 1 = Button, 2 = Check Box, 3 = Radio Button, 4 = Meter, 5 = Parameter, 6 = Slider, 7 = Data Table, 8 = Editable Text, 9 = Static Text, 12 = Switch, 13 = Text Table, 16 = Popup Menu, 17 = Embedded object.	I
GetDialogVariable (integer blockNum, string variableNameStr, integer row, integer col)	Returns the string value of the named variable (*variableNameStr* in quotes or string). If the variable is a numeric parameter or a control (such as a checkbox, slider, and so on), you would call StringToReal to convert the returned string to a numeric value. The variable can be any dialog item, static variable, global variable or dynamic array. *Row* and *col* apply to the cells of a data table or text table. For Sliders or Meters, *row* must be zero and *col* is 0 for the minimum, 1 for the maximum, and 2 for the value. Row and col are ignored for other types of items.	S
GetDragged-CloneList (integer blockNums[], String variableNames[])	If you put this function call in a DRAGCLONETOBLOCK message handler, it returns the number of clones dragged onto a block. It also fills the dynamic array parameters with the block numbers and variable names of the clones so you can get and set their values with GetDialogVariable() and SetDialogVariable() functions. This function is used in the Evolutionary Optimizer block.	I
OpenAndSelectDialogItem (integer blockNum, Str255 varName)	Opens the block's dialog, highlighting (selecting) the dialog item corresponding to varName.	P
SetDialogVariable (integer i, string variableNameStr, string value, integer row, integer col)	Sets the value of the named variable to the given numeric or string value. The variable can be any dialog item, static variable, global variable or dynamic array. *Row* and *col* apply to the cells of a data table or text table. For Sliders or Meters, *row* must be zero and *col* is 0 for the minimum, 1 for the maximum, and 2 for the value. *Row* and *col* are ignored for other types of items.	P

Block dialog items	Description	Returns
SetDialogVariable-NoMsg(integer blockNum, string variableNameStr, string value, long row, integer col)	Same as SetDialogVariable function, but doesn't send a dialog item message to the block.	P

Block dialogs (opening and closing)

These functions allow the ModL code to control the opening and closing of the block's dialog.

Block dialogs	Description	Returns
CloseBlockDialog-Box(i)	Closes the dialog for any block, where *i* is the global block number. See OpenBlockDialogBox, below.	P
CloseDialogBox()	Closes this block's dialog from within the ModL code. Put this in the EndSim message handler if you want an open dialog to close at the end of the simulation. See OpenDialogBox, below.	P
CloseEnclosing-HBlock()	Closes the hierarchical block in which this block resides, if any.	P
CloseEnclosing-HBlock2(integer blockNum)	This is same as CloseEnclosingHBlock(), except it refers to a global block number.	P
MakeDialogModal (integer theBlock-Number, integer TrueFalse)	Makes a block's dialog modal and should be called when the dialog is open. During the dialogOpen message is OK. Calling this function with the *TrueFalse* flag set to false will turn the dialog back to non-modal behavior.	P
OpenBlockDialog-Box(i)	Opens the dialog for any block, where i is the global block number. You may want to do this if you are telling the user to change a value in a block's dialog. For instance, if you have just put up an alert saying "The value in the 'Height' field of the 'Attic' dialog is negative", you can then open the offending dialog to make the change easier.	P
OpenDialogBox()	Opens this block's dialog to show something visually important to the user. If you want a dialog to always appear in front of any open plotting windows, call this function at the beginning of the Simulate message handler.	P
OpenEnclosing-HBlock()	Opens the hierarchical block in which this block resides, if any.	P

Block dialogs	Description	Returns
OpenEnclosingHBlock2(integer block)	This is same as OpenEnclosingHBlock(), except it refers to a global block number.	P

Block data tables

These functions allow ModL code to control the display of data in block dialog data tables. In the following functions, DataTableName is the name of the dialog DataTable from which you want to retrieve the selection; this name needs to be either a string variable or a string in quotes.

Block data tables	Description	Returns
DisableDTTabbing (integer blockNum, string DTname, integer disableDT)	Disables or enables the tab key functionality for the specified datatable.	I
DTHasDDELink (integer blockNum, string DTname)	Returns true if the specified datatable has an IPC advise link, or a Paste link, associated with it.	I
DynamicDataTable(integer blockNum, string dataTableName, array dynamicArrayName)	This function attaches a dynamic array to a dialog datatable. This allows dynamically resizable data tables, and the ability to change what data is displayed in a datatable without recopying the data. You can attach the array to the datatable at any time, but it will need to be reattached (dynamicDataTable will need to be called) each time that the dynamic array is resized.	I
GetDataTableSelection(string DataTableName, integer integerArray)	*IntegerArray* is a four element array declared as **`integer integerArray[4]`**. On return from the function, *integerArray* will contain the selection information in the following format: integerArray[0] -- top row integerArray[1] -- bottom row integerArray[2] -- left column integerArray[3] -- right column The integer value returned from the function will be the same as integerArray[0] if there is a valid selection. The value will be a -2 if there is no correct selection, and a -1 if an error of some other type occurred (usually an invalid *DataTableName*.)	I

Block data tables	Description	Returns
RefreshDatatable-Cells(integer block-Num, String dataTableName, integer startRow, integer startCol, integer endRow, integer endCol)	This function will redraw the specific cells of the named data table. This needs to be used in conjunction with the dynamic data tables defined by the function above, as they will not automatically update the data displayed during a simulation run when the dynamic array is modified	I
ResizeDTDurin-gRead(string name, integer oldRows, integer oldCols)	This function will allow you to inform Extend that a data table is now a different size then it was in an earlier version. This should only be called in the BlockRead message handler, and is most useful when used with the GetFileReadVersion function to inform Extend when a Datatable has been redefined. (See GetFileReadVersion for more information.)*OldRows* and *OldCols* will contain the original size of the data table.	I
ScrollDTTo(integer blockNum, String name, integer row, integer col)	Scrolls the named datatable to the indicated *row* and *column*. (0 suc-cessful, 1 failed)	I
SetDataTableCor-nerLabel (string datatablename, string15 label)	Sets a label to be displayed in the upper left-hand corner of the datatable. This area of the table is blank by default. The function will return a zero if the call succeeded, and a value of one if there was an error (most likely an incorrect *DataTableName*).	I
SetDataTableLabels (string DataTable-Name, string Label-String)	The *LabelString* variable will be used as a replacement string for the column header string of the specified DataTable. The format for this string is the same as that used in the dialog editor when creating or modifying a DataTable dialog item. The function will return a zero if the call succeeded, and a value of one if there was an error (most likely an incorrect *DataTableName*). Note: The change made by this call is not a permanent change within the block's dialog, you will need to retain the new string in a variable in the block, and call this function multiple times. See the code of the Information block (Discrete Event library) as an example of using this function. The change made by this call *is* permanent for clones of data tables.	I
SetDataTableSelec-tion(name, star-tRow, startCol, endRow, endCol, editCell)	This function selects the cells in the named data table or text table. If *editCell* is TRUE, this function sets up the first cell for entering data.	P

Block data tables	Description	Returns
StopDataTableEditing()	Immediately stops the currently selected data table from being edited. A common use would be to call this function in response to a click on the data table, preventing the user from editing the cell but allowing selection to occur.	P
SetDTRowStart (integer blockNum, String name, integer rowStart)	Sets the named datatable to have its row numbers start at the indicated value. (i.e the first row number, normally zero, will be *rowStart*) (0 successful, 1 failed)	I
SetDTColumnWidth(integer blockNum, String name, integer column, integer width, integer doClones)	Sets the named datatable column number to the *width* in pixels. If *doClones* is TRUE, also instantly updates any visible clones.	I

Block dialog tabs

These functions allow ModL code to manipulate block dialog tabs.

Block dialog tabs	Description	Returns
DisableTabName (String tabName, integer trueOrFalse)	Disables or enables the specified tab.	P
GetCurrentTabName(integer blockNum)	Returns the currently selected tab name.	S
OpenDialogBoxToTabName(String tabName, integer blockNum)	Opens a dialog box to a specified tab.	P
SetDefaultTabName (String tabName, integer blockNum)	Sets the block so that the dialog will open at a specific tab name.	P
VariableNameToTabName(String varName, integer blockNum)	Returns the name of the tab of a dialog that the indicated dialog item is on.	S

Messages to blocks (sending and receiving)

These functions make it easy to communicate information back and forth between blocks and to execute and modify processes between blocks. They can be used to set up different types of simulations that override Extend's simulation engine (i.e. Discrete Event library blocks). Sending messages to blocks can be used to enable global functions. You can put many functions in a custom function block, then use the SendMsgToBlock function to send one of the user defined messages. Use global variables to select a function and pass parameters to and from the function. Blocks can also send and receive connector messages and propagate them correctly, as discussed in "Basic item messaging" on page P201.

In the functions, "connName" is the name of the connector in the calling block (without quotation marks) and "block" is the global block number of the receiving block. (See "Block numbers, labels, names, type, position" on page P128).

Block messaging	**Description**	**Returns**
ConnectorMsg-Break()	Called from a block currently receiving a connector message. It prevents any additional connected blocks from receiving that message. This function does not affect the current message handler.	P
GetConnec-torMsgsFirst (string connName)	Makes the specified connector the first in netlist messages. This is used in blocks where it is critical that the messages be received by a specific block first, no matter what the order of connections. Use this function in INITSIM or CHECKDATA. For example, the Status block uses this function.	P
GetSimulateMsgs(integer ifTrue)	In some cases, you may not want a block to get simulate messages. See "Passing blocks" on page P197. This may be true in some discrete event blocks and in continuous blocks that are used in discrete event simulations. This function prevents blocks from getting Simulate messages and thus speeds up the simulation. The function would usually be called in the InitSim or checkData message handlers. Call this function with FALSE if you do not want the block to get Simulate messages. This is initially set to TRUE for new and existing models, and is always reset to TRUE before a simulation starts.	P

Block messaging	**Description**	**Returns**
MsgEmulationOptimize(integer ifTrue)	See "Value connector messages" on page P207. This specialized function is used in the Executive block in the Discrete Event library to optimize the operation of Generic blocks in discrete event models. It prevents the propagation of redundant messages that are produced by Generic blocks emulating and propagating connector messages from discrete blocks. It does this by not back-propagating (step 1) connector messages to blocks that have ever sent a connector message to that input of this block. This can speed the simulation by reducing the number of messages sent between discrete and generic blocks. This behavior is not a default because other types of modeling will fail if not all messages are propagated. This is initially set to FALSE for new and existing models, and is always reset to FALSE before a simulation starts.	P
RestrictConnectorMsgs (integer restrictMessages)	This function enables/disables a flag that restricts the use of connector messages in InitSim, checkdata, and endsim. This is used in the Discrete Event library Executive, as sending connector messages at the wrong times can cause problems in these libraries.	P
SendConnectorMsgTo-Block(integer i, integer conn)	Sends a connector message to a block that is not necessarily connected to the block sending the message. *i* is the global block number; *conn* is the index for the receiving connector in the connector pane (the index starts at 0).	P
SendMsg-ToAllCons(connName)	Sends a message to all connectors on other blocks that are connected to *connName* on the sending block. The connected blocks will get an "on xxx" message where xxx is the connected receiving block's connector name.	P

Block messaging	**Description**	**Returns**
SendMsgToBlock (integer block, integer message)	Sends a message to the specified global block. The *message* is an Extend constant that corresponds to the message to be sent. The message constants are ABORTSIMMSG, ABORTDLGMESSAGEMSG, BLOCKREADMSG, BLOCKCLICKMSG, BLOCKRECEIVE0MSG, BLOCKRECEIVE1MSG, BLOCKRECEIVE2MSG, BLOCKRECEIVE3MSG, BLOCKRECEIVE4MSG, BLOCKREPORTMSG, CELLACCEPTMSG, CHECKDATAMSG, CLEARSTATISTICSMSG, CONNECTIONMAKEMSG, CREATEBLOCKMSG, DELETEBLOCKMSG,DIALOGCLICKMSG, DIALOGCLOSEMSG, DIALOGOPENMSG, ENDSIMMSG, FINALCALCMSG, HBLOCKCLOSEMSG, HBLOCKOPENMSG, HELPBUTTONMSG, INITSIMMSG, OPENMODELMSG, PASTEBLOCKMSG, PAUSESIMULATIONMSG, PLOTTER0CLOSEMSG to PLOTTER3CLOSEMSG, RESUMESIMMSG, SIMULATEMSG, and STEPSIZEMSG, UPDATESTATISTICSMSG. Note: If the block is a hierarchical block, the submodel blocks will not receive the message. Instead, use the *SendMsgToHBlock* function discussed in this section.	P
SendMsgToHBlock(integer globalBlockNum, integer message)	Sends the message to all internal blocks within the hierarchical block. This function will do nothing if *globalBlockNum* is not a hierarchical block.	P
SendMsgToInputs(connName)	Sends a message to all input connectors on other blocks that are connected to *connName* on the sending block. The connected blocks will get an "on xxxIn" message where xxxIn is the connected receiving block's input connector name.	P
SendMsgToOutputs(connName)	Sends a message to the output connector on the block that is connected to *connName* on the sending block. The connected block will get an "on xxxOut" message where xxxOut is the connected receiving block's output connector name.	P

Block messaging	Description	Returns
SimulateConnector-Msgs(integer true-False)	This is used to disable the “simulation” of connector messages by generic blocks that have no connector message handlers, as described in “Value connector messages” on page P207. “Simulation” of connector messages is enabled by default for new and existing models, and is always reenabled before a simulation starts. This is a special-purpose function that will probably only be of interest to users who are creating their own libraries that are not being used with the Discrete Event library. Use this function in INITSIM or CHECKDATA.	P

Reporting

These functions are used in the BlockReport message handler to organize reports written by blocks.

Reporting	Description	Returns
GetReportType()	Used in BlockReport message handler to get the current report type. Returns 0 for Dialog report, 1 for Statistical report.	I
IsFirstReport()	Returns TRUE if this is the first block of a type getting a BlockReport message. Useful to set up column headers for that block type.	I

Plotting

These functions allow you to provide plotting in any block you create.

Each block can open up to four independent plot pages (numbered 0 to 3), each of which can plot traces (numbered 0, 1, 2 ...) that can be either arrays of points or functions. Each trace can be assigned one of two Y axes, Y and Y2, so that traces of different magnitudes can appear in the same plot.

The following arguments are used in the plotting functions:

Plot argument	Definition
Plot	Number (0 to 3) specifying one of the four possible plot pages
WhichSig	Number (0, 1, 2 ...) specifying one of the plot traces
SigName	Name for the trace
Y	Name of the dynamic array used in the plot. Use MakeArray to allocate Y before using it in a function.
StartTime	X-axis value that corresponds with the first point (y[0]) of the array.

Plot argument	Definition
EndTime	X-axis value that corresponds with the last point of the array.
PlotPts	Number of points in the array ready to plot, which does not need to be the number of points actually declared in the array. If there are no points ready to plot (because they have not yet been calculated), use 0 for this value.
UseY2Axis	If TRUE, the trace is plotted to the limits of the Y2 axis instead of the main Y axis.
Pattern	Pattern of the plot line. You can use the numbers or their associated Extend constants. Note that these are different than the animation patterns. 0 BlackPattern; 1 DkgrayPattern; 2 GrayPattern; 3 LtgrayPattern
Color	Color of the plot traces. You can use the numbers or their associated Extend constants. Note that these are different than the animation colors. 33 BlackColor 205 RedColor 341 GreenColor 409 BlueColor 273 CyanColor 137 MagentaColor
MaxLines	Initial number of rows in the plot's data pane.
IsXLog	If TRUE, the X axis is logarithmic.
IsYLog	If TRUE, the Y axis is logarithmic.
IsY2Log	If TRUE, the Y2 axis is logarithmic.
FunctionName	Name of a real, single argument function. For example, cos(x) should be entered as `cos`.
NumFormat	Format for the numbers in the data pane. 0 General 1 decimal (2 places) 2 integer 3 scientific (exponential)
LineFormat	Format for traces. 0 – connected points 1 – rectangular connected points (no interpolation between points) 2 – dots
SymFormat	Symbols used on traces. 0 – .; 1 – □; 2 – △; 3 – ○; 4 – +; 5 – ■; 6 – ▲; 7 – ● 8 – # (displays the trace's number); 9 – ✕

General plotting

General plotting	Description	Returns
AutoScaleX(integer plot)	Finds the minimum and maximum X axis values of all installed arrays or installed scatter arrays and adjusts the X axis limits accordingly.	P
AutoScaleY(integer plot)	Finds the minimum and maximum Y values of all installed arrays or installed scatter arrays and adjusts both the Y and Y2 axis limits accordingly.	P
BarGraph(integer plot, integer aBins, real aMin, real aMax)	Sets the number of bins, the minimum value, and maximum value for a bar graph.	P
ChangeAxisValues (integer plot, real xLo, real xHi, real yLo, real yHi, real y2Lo, real y2Hi)	Sets or changes the axis values of the specified plotter to the value specified. Specifying a BLANK for a given value will use the existing value. Returns a TRUE if the function executed successfully.	I
ChangePlotType (integer plot, integer plotType)	Changes the defined plot type of a plotter. This is used when creating custom plotters. The call to change the plotter should be done right after the call to install an axis and before any other of the plotter calls. 1 – linear plot; 2 – scatter plot; 3 – error bars; 4 – strip chart; 5 – worm plot; 6 – bar plot	P
ChangeSignalColor (integer plot, integer whichSig, integer color)	Specifies the color for the trace.	P
ChangeSignalSymbol(integer plot, integer whichSig, integer symFormat)	Specifies the symbol for the trace.	P
ChangeSignalWidth(integer plot, integer whichSig, integer width)	Specifies the width (in pixels) for the trace.	P
ClosePlotter(integer plot)	Closes the plot window, if open.	P

General plotting	Description	Returns
ClosePlotter2 (integer blockNum, integer plot)	This function just closes the specified plotter window, if it is open.	P
DisposePlot(integer plot)	Disposes of a plot window and all its saved plots. This does not dispose of any installed data arrays.	P
GetAxis(integer plot, real axisValues[9])	Fills the first nine elements of the *axisValues* array (declared as real *axisValues* [9]) with the current values of isXLog, xLow, xHi, isYLog, yLow, yHi, isy2Log, y2Low and y2Hi.	P
GetAxisName(integer plot, integer whichAxis)	Returns the name of the specified axis. whichAxis: 1 - x 2 - y1 3 - y2	S
GetSignalName (integer plot, integer whichSig)	Returns the name of the signal *whichSig*.	S
GetSignalValue (integer plot, integer whichSig, real xAxisValue)	Finds the value of the installed trace at *xAxisValue*. If the trace is an installed array, the value is interpolated between adjacent points. If the trace is an installed function, the function is evaluated at *xAxisValue*. This does not work for scatter plots because there may be many Y values for a single *xAxisValue*.	R
GetTickCount(integer plot, integer whichCount)	Returns the number of ticks on the plotter axis. *WhichCount* is 0 for the X axis, 1 for the Y axis, and 2 for the Y2 axis.	I
GetY1Y2Axis(integer plot, integer whichSig)	Returns an integer specifying which Y axis the signal is plotted against and whether or not the signal is hidden. 1: y1 2: y2 -1: y1, hidden -2: y2, hidden Note: If you are not concerned whether the signal is hidden, take the absolute value of the result.	I

General plotting	Description	Returns
InstallArray(integer plot, integer whichSig, string sigName, real y, real StartTime, real EndTime, integer plotPts, integer useY2Axis, integer pattern, integer color)	Allows the automatic plotting of arrays. The plot window will plot all the points of this array up to plotPts-1. It should be preceded by InstallAxis and eventually be followed by showPlot. The first time arrays are installed they need to be installed in sequential order. The Install Array call is used in two ways in plotter blocks. The first time it is called, it installs the array and sets parameters such as the color and pattern of the line. The second and subsequent times, the formatting options are ignored, and the only action a call to installArray takes to reinstall the array itself. If you precede a call to InstallArray with a call to RemoveSignal, the formatting information will be used.	P
InstallAxis(integer plot, string title, string xName, integer isXLog, real xLow, real xHi, string yName, integer isYLog, real yLow, real yHi, string y2name, integer isY2Log, real y2Low, real y2Hi, integer y2Pattern, integer y2Color, integer maxLines)	Installs both the X and Y axes. If both the *y2Low* and *y2Hi* arguments are 0, the Y2 axis is not shown.	P
InstallFunction (integer plot, integer whichSig, string sigName, functionName, integer useY2Axis, integer pattern, integer color)	Allows the automatic plotting of functions. The plot window will plot all the points of this function corresponding to the x axis values, even if you changed them. It should be preceded by InstallAxis and eventually followed by ShowPlot. Call InstallFunction every time the plot window is shown (before ShowPlot).	P
NumPlotPoints (integer plot, integer points)	Specifies the number of points to be stored for worm and strip plots	P

General plotting	Description	Returns
PlotNewBarPoint (integer plot, integer whichSig, integer whichBin, real Value)	This function allows you to set the value of one of the bins in a histogram/barchart chart, rather then adding one to it as the PlotNewPoint function will do.	P
PlotNewPoint(integer plot, integer whichSig, integer index, real yValue)	Adds and plots a new *yValue* point to the installed array. It is useful when you want to see points plotted as they are calculated within a loop (such as during simulations). To use this effectively, first use InstallArray with a value of 0 for the plotPts argument. PlotNewPoint increments the plotPts of the installed array when the point is plotted, and the equation "y[index] = yValue" is internally executed, putting the new *yValue* into the installed array. Note that the first value of index must be 0 when calling PlotNewPoint for a newly installed array.	P
PlotSignalFormat (integer plot, integer whichSig, integer lineFormat, integer numFormat)	Specifies the line format and number format for the trace.	P
PlotterSquare(integer plot, integer trueFalse)	If true, forces the window to be square. If false, it removes the squaring restriction.	P
PushPlotPic(integer plot)	Pushes the top plot picture, page 1, down to page 2 of the plot window. The oldest plot (page 4) is discarded. This function only works if the plot is showing when the function is called.	P
RefreshPlotter(integer plot, integer wholeframe, integer openPlotterWindow)	Function that will redraw the plotter without opening the plotter window if it is closed. Used for redrawing the clones of a plotter when a change has occurred.	P
RemoveSignal(integer plot, integer whichSig)	Removes an installed array or function from a plot so that a new one can be installed.	P
RenamePlotter(integer plot, string newname)	Renames the plotter.	P

General plotting	Description	Returns
RetimeAxis(integer plot)	Sets the X axis values to the simulation start and end times. This is useful for plotting simulation results without having to reset the x axis values before a simulation is run.	P
RetimeAxisNStep (integer plot, integer nStep)	This function is used to correct axis values when "Plot every nth Point" is selected in the plotter's dialog. As an example, see the Plotter I/O block in the Plotter library.	P
SetAxisName(integer plot, integer whichAxis, string theName)	Sets the Axis name of the specified Axis. *whichAxis*: 1-x 2-y1 3-y2	I
SetSignalName (integer plot, integer whichSig, string theName)	Sets the name of signal *whichSig*.	P
SetTickCounts (integer plot, integer xCount, integer yCount, integer y2Count)	Specifies the number of ticks on the plotter axes in the plot. Specifying -1 as the parameter will use the current value.	P
ShowPlot(integer plot, string plotName)	Opens and names the plot window. If the plot has never been used, an empty plot is shown. Additional calls bring the window to the front. *plotName* is a string that contains the name of the plot. Note that ShowPlot will not change the name of a plot that was typed in the plot dialog.	P
SwitchPlotterRedraw(integer plot, integer direction)	Specifies the direction in which the plot will be redrawn. A *direction* of 0 specifies left to right, a *direction* of 1 specifies right to left. This only affects the way that the plot lines are redrawn when the plot is complete.	P

Remember to call the ShowPlot function to replot any changes made by these functions. Be sure to call ShowPlot after all calls that change the plot axis limits or formats.

The structure and contents of plots are saved with the model file, whether they were open or not.

Scatter plot functions

To plot scatter plots (XY plots), you need two additional functions. Scatter plot windows combine two traces previously installed into a plot. The traces are combined into an XY trace, where the first trace supplies the X values, and the second trace supplies the Y values. The values of the whichSig argument for the Scatter Plot functions must range from 0 to an even number minus one specifying the highest numbered scatter trace.

Scatter plots	Description	Returns
MakeScatter(integer plot, integer whichSig)	Combines the two traces (the first specified by *whichSig*) into a scatter trace. Both *whichSig* and *whichSig*+1 traces must be the same length and must have been installed by InstallArray before MakeScatter is called.	P
PlotNewScatter (integer plot, integer whichSig, integer index, real xValue, real yValue)	Similar to plotNewPoint but for scatter plots. *xValue* is put into *whichSig* and *yValue* is put into *whichSig*+1.	P

To see how the plotting functions can be used in your own blocks, examine the plotter functions that are used in the code of the various plotters in the Plotter library.

Arrays, queues, and delay functions

Dynamic and non-dynamic arrays

The following functions manipulate arrays.

Dynamic arrays are declared as static arrays with their first dimension missing, such as:

```
real, integer, or string array[], array2Dim[][5];
```

Note If you pass a dynamic array as an argument to a user-defined function, you cannot use this local argument name as the argument to the MakeArray or DisposeArray functions, but must instead use the original static declared name.

Dynamic arrays	Description	Returns
ArrayDataMove (array y[], integer StartIndex, integer rowsToMove, integer targetIndex, integer clearData)	This function moves a specified number of rows of data (rowsToMove) from a specified location (startIndex) in the array to another specified location in the array (targetIndex). If the clearData flag is set to TRUE (1), it will clear out the old data locations in the array. Returns 0 if successful.	I
DisposeArray(array)	Dynamic arrays only. Releases the memory used by a dynamic array. Use this when you have finished using a dynamic array to save memory and model file space. Call this function with the original static declared name of the array (see notes above).	P

Dynamic arrays	**Description**	**Returns**
FindMinimum(real RealArray, integer ResultArray)	Accepts two dynamic arrays; one real, one integer. It searches the RealArray for the minimum values and returns the minimum value. The ResultArray is filled with the position numbers of the elements that contain the minimum value.	R
FindMinimum-WithThreshhold (realArray y, integer-Array y2, real threshhold)	This function is identical to the FindMinimum function defined in the above, with the addition of the threshhold argument. This argument specifies a threshhold below which the real values will be ignored. I.e. the minimum value found will always be at the thresh-hold, or above.	R
GetDimension (array)	The size of the first dimension. If *array* is a dynamic array, this function will return the size of the dynamic dimension. This is useful for determining the maximum subscript (the returned value minus 1) of an array when it is passed to a user-defined function.	I
MakeArray(array, integer i)	Dynamic arrays only. Allocates a dynamic array. *i* is the desired value of the first (blank) dimension. MakeArray does not clear or disturb data already in the array and can be used to make an array larger or smaller. If you need to initialize an array to 0 or BLANK, this must be done by the block's code. Call this function with the original static declared name of the array (see notes above).	P
SortArray(array, integer numRows, integer keyCol-umn, integer increase, integer sortStringAsNum-bers)	Sorts the array up to *numRows*. Uses the *keyColumn* as the sorting column. If increase is TRUE, sorts in ascending order (note that, for purposes of sorting, NoValues are considered larger than any number). If this is a string array and *sortStringAsNumbers* is TRUE, sorts strings as numbers in *keyColumn*. This function works with any kind of array, including data tables and text tables.	P

Passing arrays

These functions let you pass dynamic arrays through connectors or global variables. This is discussed in more detail in “Passing arrays” on page P213. Also see the “Passing Arrays” example (*Macintosh:* Passing Arrays in the ModL Tips folder; *Windows:* PassArry.mox in the ModLTips directory).

Passing arrays	Description	Returns
GetPassedArray(real realVar, array)	Gives access to array values from the input connector or real number *realVar* that holds the location of a passed array. *array* must be declared in this block as a dynamic array of the same type as the passed array. After this function is called, *array* can be used to access and change the values in the passed array. This function returns TRUE if *realVar* is a passed array from another block and returns FALSE if it is not a passed array.	I
PassArray(array)	Returns the real value that represents the location of a dynamic array declared in this block. This is used to pass the array to an output connector or real variable (such as a global variable). Do not call PassArray to pass an array that has been received by GetPassedArray; see "Passing arrays" on page P213, for more information.	R

Global arrays

The Global Array functions define and manage global arrays (see "Global arrays" on page P83). All of the functions except the GAGetXXX functions will return a negative number if they fail. The following numbers have standard meanings:

Value	Meaning
-1	No arrays defined, or array not found
-2	Array index invalid, or array not defined
-3	Row or column reference out of range
-4	Incorrect array type
-5	Name is blank, or too long (> 15 characters)
-7	Wrong type of array
-8	Array length doesn't match global array length

The type of a global array can be defined by the following constants:

Constant	Value
GAReal	1
GAInteger	2
GAString	3
GAString15	4
GAString31	5

Global Arrays	Description	Returns
GAClipboard (integer arrayIndex)	Copies the array with the specified index into the clipboard, where it will be inserted into the selected model with the next paste.	P
GACreate(string name, integer type, integer columns);	This function creates an array with the specified *name*, and *type*. Arrays are created with zero rows. Returns the index number	I
GACreateQuick (string name, integer type, integer columns)	This function behaves exactly the same way as the GACreate function, with the exception that it will not check the name to see if a global array already exists with that name. The only case where you would want to use this function is where you are creating a large number of GA's, speed is of the essence, and you are sure that there will be no duplication of names. If you do create an array with the same name as an existing array, referencing that array by name will only access the older array.	I
GACreateRandom (integer type, integer columns)	Creates a Global Array with a random name. Useful for quick creation of global arrays in cases where you need to create many arrays very quickly.	I
GADataTable (integer blockNum, string dtName, integer arrayIndex)	Associates a Global Array with a dialog datatable in the same way the DynamicDatatable function associates a dynamic array with one. See the DynamicDatable() function description for more information.	I
GADispose(string name)	Disposes the named Array.	I
GADisposeByIndex (integer arrayIndex)	Disposes the Global Array referenced by *arrayIndex.*	I
GAFindStringAny (integer arrayIndex, string findString, integer column, integer numRows, integer numChars, integer caseSensitivity)	This function searches a specific string Global Array, referrenced by *arrayIndex*, for the first string that matches the *findString*. The return value is the index of the array element that contains the first string found. A -1 will be returned if the string is not found.	I
GAGetArray(integer arrayIndex, integer row, array y)	Sets the contents of the array *y* to the values in the Global Array with the specified index. This will copy the contents of the specified *row* of the Global Array into the array *y.* (You need to make sure that the number of elements in the dynamic array match the number of columns in the *row* of the global array.)	I

Global Arrays	Description	Returns
GAGetColumns (string name)	Returns the number of columns defined for the named array.	I
GAGetColumnsBy-Index (integer array-Index)	Returns the number of columns defined for the Global Array referenced by *arrayIndex.*	I
GAGetIndex(string name)	Returns the Index value of the named array.	I
GAGetInteger(integer arrayIndex, integer row, integer column)	Returns the integer value stored at the specific row and column of the array with the specified index.	I
GAGetLong(integer arrayIndex, integer row, integer column)	This function is also known as GAGetInteger. See GAGetInteger for description.	I
GAGetName(integer arrayIndex)	Returns the Name of the array with the specified index.	S
GAGetReal(integer arrayIndex, integer row, integer column)	Returns the real value stored at the specific row and column of the array with the specified index.	R
GAGetRows(string name)	Returns the number of rows defined for the named array.	I
GAGetRowsByIndex (integer arrayIndex)	Returns the number of rows defined for the Global Array referenced by *arrayIndex.*	I
GAGetString (integer arrayIndex, integer row, integer column)	Returns the string value stored at the specific row and column of the array with the specified index.	S
GAGetString15(integer arrayIndex, integer row, integer column)	Returns the string value stored at the specific row and column of the array with the specified index.	S

Global Arrays	Description	Returns
GAGetString31(integer arrayIndex, integer row, integer column)	Returns the string value stored at the specific row and column of the array with the specified index.	S
GAGetType(string name)	Returns the type of the named Array. See table above for type values.	I
GAGetTypeByIndex (integer arrayIndex)	Returns the type of the Global Array referrenced by *arrayIndex.*	I
GANonSaving(integer arrayIndex, integer nonSavingArray)	This function flags the specific array as a non saving array. The array will not be written out to the model file when the file is closed. (By default Global arrays are 'saving' arrays.)	I
GAPtr (integer arrayIndex)	Returns the memory pointer to the data associated with a particular global array. (For use for passing data to dll's only, and it should be called only immediately before the DLL call, as memory can move.)	I
GAPopupMenu (integer arrayIndex, string name, integer rows, integer init, integer flying)	This function copies the strings in the specified Global array into the named Popup menu. This is a utility that allows quick construction of a popup menu. The flying argument is true if the menu is being created "on the fly", as opposed to adding the array to the existing menu. See the code that controls the *Animation* tab of any Discrete Event library block as an example.	I
GAResize(string name, integer rows)	Changes the number of rows defined for a global array.	I
GAResizeByIndex (integer arrayIndex, integer size)	Changes the number of rows defined for the Global Array referenced by *arrayIndex.*	I
GASetArray(integer arrayIndex, integer row, array y)	Sets the contents of the Global Array with the specified index to the values in the array y. This will copy the contents of the array y into the specified row of the Global Array. (You need to make sure that the number of elements in the dynamic array match the number of columns in the row of the global array.)	I
GASetInteger(integer value, integer arrayIndex, integer row, integer column)	Sets the integer value at the specific row and column of the array with the specified index.	I

Global Arrays	Description	Returns
GASetLong(integer value, integer array-Index, integer row, integer column)	This function is alos known as GASetInteger. See GASetInteger for description.	I
GASetReal(real value, integer array-Index, integer row, integer column)	Sets the real value at the specific row and column of the array with the specified index.	I
GASetString(string value, integer array-Index, integer row, integer column)	Sets the string value at the specific row and column of the array with the specified index.	I
GASetstring15(string value, integer arrayIndex, integer row, integer col-umn)	Sets the str15 value at the specific row and column of the array with the specified index.	I
GASetstring31(string value, integer arrayIndex, integer row, integer col-umn)	Sets the str31 value at the specific row and column of the array with the specified index.	I
GASort(integer arrayIndex, integer numRows, integer keyColumn, inte-ger increase, integer sortstringAsNum-bers)	Sorts the Array with the specified index. Sorts the array up to *num-Rows*. Uses the *keyColumn* as the sorting column. If increase is TRUE, sorts in ascending order (note that, for purposes of sorting, NoValues are considered larger than any number). If this is a string array and *sortStringAsNumbers* is TRUE, sorts strings as numbers in *keyColumn*.	I

Queues

These functions store queues in a single-dimensional real, dynamic array, that is, an array that is declared with no dimension, such as **`real a[]`** (see the section above on dynamic arrays). You do not need to use the MakeArray function before calling QueInit. When you have finished with a queue, you should recover the memory it occupies with the DisposeArray function; this is often done in the EndSim message handler.

In queues, the first member is numbered “0”, the next member “1”, and so on. This means that in an array of n members, the last member is numbered “n-1”. n and i are integers, and x is a real value.

Queues	Description	Returns
GetFront(real array[])	Removes and returns item 0 from the front of the queue. If the queue is empty, NoValue is returned.	R
GetRear(real array[])	Removes and returns item n-1 from the rear of the queue. If the queue is empty, NoValue is returned.	R
PutFront(real array[], real x)	Adds *x* to the front of the queue.	P
PutRear(real array[], real x)	Adds *x* to the rear of the queue.	P
QueGetN(real array[], integer i)	Removes and returns the ith member of a queue. If the ith member does not exist, noValue is returned. This compacts the queue after the ith member is removed so that the *i*+1st member becomes the new ith member.	R
QueInit(real array[])	Allocates and initializes the array for the queuing functions. Call this procedure in the InitSim message handler for each queue.	P
QueLength(real array[])	Length of the queue. If the queue is empty, the function returns 0.	I
QueSetAlloc (long alloc, long realloc)	Specifies the allocation and reallocation constants for the queueing functions. This function allows the user to control how much memory is allocated by the queueing functions to initially allocate memory for item storage, and how much additional to add each time they need to be resized bigger. The default values are 200 for alloc, and 500 for realloc.	P
QueLookN(real array[], integer i)	Value of the ith member of a queue without changing the queue order. If the *i*th member does not exist, noValue is returned.	R
QueSetN(real array[], integer i, real x)	Sets the value of the ith member of a queue to x without changing the queue order. If *i* is greater than the length of the queue, an error message informs you and aborts the operation.	P

Delay lines

These functions store delay lines in a single-dimensional real, dynamic array, that is, an array that is declared with no dimension, such as **`real a[]`** (see the section above on dynamic arrays). Delay lines are used in continuous simulations to delay values by a constant amount of time. They are like a pipe with values flowing in one end and out the other; the delay time is analogous to the length of the pipe.

Although the delay line functions store delay lines in dynamic arrays, you must not use the Make-Array function to allocate these arrays. Array allocation is handled automatically by the delay line functions. When you have finished with a delay line, you should recover the memory it occupies with the DisposeArray function.

Delay lines	**Description**	**Returns**
Delay(real array[], real x)	Inserts a new value *x*, and returns a delayed value from a delay line. The returned value will be the value inserted DelayTime ago.	R
DelayInit(real array[], real Delay-Time)	Initializes a dynamic array delay line to *DelayTime*. Call this procedure in the InitSim message handler for each delay line.	P

Scripting

These functions are useful in building or changing models when called from blocks or from other applications. See also the GetDialogVariable() and SetDialogVariable functions under "Dialog items from other blocks" on page P133.

Scripting	**Description**	**Returns**
ActivateWorksheet (string sheetName)	Activates, (selects and brings to the front), the named worksheet.	I
AddBlockToClip-board (integer blockNum)	Adds the indicated block to the clipboard contents without other-wise changing the contents. Returns FALSE if successful.	I
AddBlockToSelec-tion (integer block-Num)	Adds the indicated block/worksheet object to the selection. This is in contrast to selectBlock which makes the indicated block the entire selection. Returns FALSE if successful.	I
ChangePreference (integer preference, integer value)	Allows you to change preference values using ModL code. The *value* argument takes a zero for FALSE or a one for TRUE. The *preference* argument takes the following values: 8 = bitmap plotters; 9 = bitmap blocks; 10 = auto library search; 12 = simulation sound; 14 = right angle connections; 16 = unsupported XCMD callbacks beep (Macintosh only); 20 = updatepubs (Macin-tosh only); 21 = don't show worksheet tool tips; 22 = don't show dia-log editor tool tips; 25 = don't show hierarchical block drop shadows; 26 = plotters use patterns. See GetPreference(), below.	P
ClearBlockUndo (integer theBlock-Number)	Clears the block and adds it to the delete task list to allow undoing. Returns FALSE if successful.	I

Scripting	**Description**	**Returns**
ClearConnection(integer blockFrom, integer conFrom, integer blockTo, integer conTo)	Removes the connection between the specified blocks and connector numbers. Returns TRUE if successful.	I
CodeExecute(string modlCodeString)	Executes the ModL code in the string *modlCodeString*. Local variables may be defined. Global variables can be used to input and return values.	P
ClearBlock(integer BlockNum)	Clears the specified block from the worksheet.	P
CreateHBlock (string theName)	Makes the current selection into an Hblock with the specified name.	I
DuplicateBlock (integer theBlockNumber)	Makes a copy of the indicated block, and returns the block number of the new block. Returns FALSE if successful.	I
ExecuteMenuCommand(integer commandNumber)	Executes the specified command. This is functionally the same as selecting the command from the menus. Extend will attempt to perform the command on the active window. If there are multiple windows open (including dialog boxes), you may need to call ActivateWorksheet() to insure that the correct model is in the active window. See Appendix H for a list of menu command numbers.	P
ExtendMaximize()	Maximizes the Extend Application.	P
ExtendMinimize()	Minimizes the Extend Application.	P

Scripting	Description	Returns
FindBlock (string searchStr, integer which, integer openDialogs, integer wholeWords, integer justBlockNum)	Finds the first block that match the string *searchStr* and returns it's block number, optionally opening the dialog or selecting the block. *Which* specifies what string in the block you want to compare your searchstring to. It takes the following values: BlockLabel 1 BlockName 2 BlockType 3 TextBlockText 4 *openDialogs* specifies if the block should be just selected, or if the dialog should be opened. It is overridden by the justBlockNum parameter, if it is TRUE. *WholeWords* specifies if you want the text to try for an exact match, or to match a partial string. The final argument *justBlockNum* if set to true will set the function to just return a block number, and neither select the block nor open the dialog.	I
FindNext()	Finds the next block that matches the search string specified. Only useful if called immediately after the findBlock function. (returns a –1 if no matching block is found.)	I
GetAppPath()	Returns a string containing the full path name for the Extend application file. This function is useful for determining the location of files for which only the file's relative location to the Extend application is known.	S
GetPreference (integer preference)	Allows you to get preference values using ModL code. The return value is 1 for TRUE and 0 for FALSE. The *preference* argument takes the following values: 8 = bitmap plotters; 9 = bitmap blocks; 10 = auto library search; 12 = simulation sound; 14 = right angle connections; 16 = unsupported XCMD callbacks beep (Macintosh only); 20 = updatepubs (Macintosh only); 21 = don't show worksheet tool tips; 22 = don't show dialog editor tool tips; 25 = don't show hierarchical block drop shadows; 26 = plotters use patterns.	I

GetWorksheetFrame(integer blockNum, array arrayName)	Returns the frame of the worksheet in the array *arrayName*, which must be declared integer arrayName[4]. Coordinates are in screen pixels and correspond to the information returned by the GetMouseX(), GetMouseY(), and GetBlockTypePosition() functions.
	ArrayName[0] = top;
	ArrayName[1] = left;
	ArrayName[2] = bottom;
	ArrayName[3] = right;
		I

Scripting	Description	Returns
IsBlockSelected (integer blockNumber)	Returns a TRUE if the indicated block is selected, returns false otherwise. This function will return a –1 if the indicated blocknumber is not a valid block.	I
IsLibEnabled(integer library)	Returns TRUE if the given library has been enabled for use in Extend. Library values are: 1 - BPR library 2 - Manufacturing library 5 - Proof library	I
IsMenuItemOn(integer whichItem)	Returns TRUE if the given menu command is currently checked. The whichItem argument uses the same numbers as defined in Appendix H for the ExecuteMenuCommand() function.	I
MakeBlockInvisible (integer globalBlockNum, integer invisible)	Makes the indicated block invisible. (Turns it visible if it was already invisible, and the invisible flag is set to false.) Invisible blocks will not display on the worksheet at all, and cannot be selected by the user. Returns FALSE if successful.	I
MakeConnection (integer blockFrom, integer conFrom, integer blockTo, integer conTo)	Makes a connection line from the From block to the To block.	I
MoveBlock(integer blockNumber, integer xPixel, integer yPixel)	Moves the specified block the specified number of pixels. Coordinates refer to the upper left corner of the block.	I
MoveBlockTo(integer blockNum, integer xLoc, integer yLoc)	Moves the specified block to the specified location. Coordinates refer to the upper left corner of the block.	I
OpenExtendFile (string fullPath)	Opens the Extend model, library, or text file, using the *fullpath* name on the disk. This creates a new front window if opening a model or text file. Returns a zero if successful.	I

Scripting	Description	Returns
PlaceBlock(string blockName, string libName, integer xPixel, integer yPixel, integer neighbor, integer side)	Places the named block from the named library onto the active worksheet at the specified location. If the *neighbor* field is filled in with a block number of a block already on the worksheet, then the *xPixel*, *yPixel* values are relative to the location of the *neighbor*, otherwise they are absolute worksheet coordinates. The *side* argument determines how the new block will be placed relative to the old block. 0: Left 1: top 2: right 3: bottom	I
PlaceBlockInHBlock (string blockName, string libName, integer xPixel, integer yPixel, integer HBlockNum)	Places a copy of the named block from the named library in the Hblock that is specified by the HblockNum argument. See PlaceBlock in your manual for more information. Returns FALSE if successful.	I
PlaceTextBlock (string text, integer xPixel, integer yPixel, integer neighbor, integer side, integer width)	Places a Text Block with the string text as its content onto the active worksheet at the specified location. The width argument specifies the final width of the text block in pixels. See the description of PlaceBlock above for descriptions of the other arguments.	I
SelectConnection (integer blockFrom, integer conFrom, integer blockTo, integer conTo)	Selects the connection line associated with the connection between the specified blocks. This function returns a TRUE if it succeeds, and a FALSE if it fails.	I
SetDirty(integer dirty)	Sets/Unsets the 'dirty' flag on the active worksheet. A common use for this functionality would be to set the dirty flag to 'FALSE' before issuing the ExecutemenuCommand function to close a worksheet. This would have the effect of closing the worksheet without querying the user if they want to save, or not.	P
SetSelectedConnectionColor (integer hue, integer saturation, integer value)	The setSelectedConnectionColor function sets the color value of all selected connections. See "Color palette" on page P124 for a description of the hue, saturation and value arguments. This function returns a TRUE if it succeeds, and a FALSE if it fails.	I

Scripting	Description	Returns
SetRunParameters(real SetEndTime, real SetStartTime, real SetNumSims, real SetNumStep)	Sets the parameters in the Simulation Setup dialog. This function returns the following: 0 Successful -1 End time must be greater than start time -2 Number of steps must be at least one -3 Number of steps must be less than 2000000000 -4 Number of simulations must be at least 1 and less than 32768	I
SetConnectionColor (integer blockFrom, integer conFrom, integer blockTo, integer conTo, integer hue, integer saturation, integer value)	The SetConnection color function sets the color value of the specified connection line. See "Color palette" on page P124 for a description of the hue, saturation and value arguments. This function returns a TRUE if it succeeds, and a FALSE if it fails.	I

Miscellaneous functions

Strings

These functions allow you to change and parse a string into whatever component parts you want. Note that, in ModL, the "+" operator acts as a string concatenation operator.

The functions that require a character position indicate the first character in a string as position 0. In these functions, the arguments s, findString, and replaceString are strings.

Strings	Description	Returns
FormatString(integer numArgs, string formatString, String value1, String value2, String value3, String value4, String value5, String value6, String value7, String value8)	This function creates formatted strings using string value arguments. The *formatString* argument is used to define the format of the returned output string. The definition of the format string is based on the C function sprintf. Please refer to a standard C reference for more information on how to define the format string. *NumArgs* defines the number of value arguments that contain meaningful values.	S

Strings	Description	Returns
FormatStringReal (integer numArgs, string formatString, real value1, real value2, real value3, real value4, real value5, real value6, real value7, real value8)	This function creates formatted strings using real value arguments. The *formatString* argument is used to define the format of the returned output string. The definition of the format string is based on the C function sprintf. Please refer to a standard C reference for more information on how to define the format string.	S
NumToFormat(real x, integer maxchar, integer sigFigs, integer format)	Returns the number formatted as a string. *X* is the input number, *maxchar* is the maximum number of characters in the returned string, *sigFigs* is the number of significant figures desired. *Format* is 0 for general, 1 for currency, 2 for integer, 3 for scientific notation.	S
RandomString(integer n)	Returns a randomly generated string of *n* upper case alphabetic characters.	S
RealToStr(real value, integer sigFigs)	Converts the *value*, rounded to *sigFigs* significant figures, to a string.	S
StrFind(string s, string findString, integer caseSens, integer diacSens)	Character position of *findString* within string *s*. The first position of a string is 0, so if the *findString* is not found, StrFind returns -1. If *caseSens* is TRUE, case is considered in the search. If *diacSens* is TRUE, diacritical marks are considered in the search.	I
StrGetAscii(string s)	First character of *s* as an integer value corresponding to the ASCII value.	I
StringCase(string str, integer lowerCase)	Returns *str* converted to lower case if *lowerCase* is TRUE, upper case if *lowerCase* is FALSE.	S
StringCompare (string s1, string s2, integer caseSens, integer diacritical)	Returns -1 if *s1* < *s2*, 0 if *s1* == *s2*, 1 if *s1* > *s2*. If *caseSens* is TRUE, uses case. If *diacritical* is TRUE, uses diacritical marks (ä, é, ö, etc.).	I
StringTrim (string input, long which)	This function trims leading and trailing blank spaces off the input string. Spaces, CR, LF, *and TAB characters will be trimmed. The resulting string is returned. The argument, ‘which’, takes the following values: 0-both leading and trailing blanks trimmed. 1-leading blanks trimmed. 2-trailing blanks trimmed.	S

Strings	**Description**	**Returns**
StrLen(string s)	Number of characters in the string *s*.	I
StrPart(string s, integer start, integer i)	Substring of string *s*, starting at character position *start*, *i* characters long. Note that string length is 256 character maximum.	S
StrPutAscii(integer i)	String of length 1 corresponding to the ASCII value of *i*. This is useful for putting non-printing control characters into a string.	S
StrReplace(string s, integer start, integer i, string replaceString)	The substring of string *s* starting at start of length *i* is replaced with *replaceString*.	S
StrToReal(string s)	Real value converted from string *s*, ending with the first space or letter found that is not part of the number. If *s* does not represent a number, the function returns a NoValue.	R
TextWidth(string theString, integer font, integer face, integer size)	This function returns the width the specified string would draw at in pixels. If font, face, and size are all zero the function will use the default values for the animation text functions.	I

Attributes

Use these functions in discrete event blocks to work with attribute strings. Attribute strings are formatted as: **AttrName1=val1;AttrName2=val2;...**

Note These function are provided for backwards compatibility with version 3.x. New blocks built in version 4.0 should not use these functions.

Attributes	**Description**	**Returns**
GetAttributeValue(string attrString, string attrName)	Returns the value of the attribute *attrName* or NoValue if *attrName* isn't found. If you pass in a blank (empty) string for the *attrName* variable, function returns the value of the first attribute in the attribute string.	R
RemoveAttribute(string attrString, string attrName)	Returns the attribute string after removing attribute *attrName*.	S
SetAttribute(string attrString, string attrName, real value)	Adds *attrName* and value or if it already exists, changes the *attrName* to value. Returns the attribute string after adding or changing the attribute value.	S

Date and time

Extend stores dates and times in an integer that is the number of seconds since midnight, January 1, 1904. See "Dates and times" on page P235 for more information.

Date and time	**Description**	**Returns**
CalcDate(integer year, integer month, integer day, integer hour, integer minute, integer second)	Returns the numeric time value for the specified information. The arguments are all integers. The *year* argument is entered as the entire year (e.g. 1993) where years range from 1904 to 2040; *months* are entered from 1 to 12 (4 for April); *days* are 1-31; *hours* are based on the 24-hour clock and go from 0 to 23 (2 in the afternoon would be 14); *minutes* and seconds are 0-59. For example, February 1, 1994 at 2:00 PM would be (1994, 2, 1, 14, 0, 0).	I
DateToString(integer i)	String containing the date in the localized date format (usually MM/DD/YY) corresponding to integer *i*.	S
DiffDate(integer firstDate, integer secondDate)	Returns the difference between two date values as a real number which represents the number of days between the two dates.	R
GetBlockDates(integer blockNum, integer whereFrom, integer whichDate)	Returns the modified and created dates of the block. The *whereFrom* argument takes a zero for the block or a one for the library. The *whichDate* argument takes a zero for created or a one for modified. Use the dateToString & timeToString functions to get the string values of the date & time.	I
ModifyDate(integer oldDate, real dateModifier)	Returns the old date value plus the date modifier value. The *oldDate* value is an Extend integer date value, similar to that returned from the GetBlockDates function, and the *dateModifier* value is a real number representing a number of days.	I
Now()	Number of the current date and time.	I
StartTimer(integer blockNum, integer waitTicks)	Starts a timer chore that will periodically send messages to either the specified block, or every block in the active model if the blocknum is zero. This chore is performed when the CPU is idle, so you cannot count on it happening exactly every period, unless the CPU is idle. (I.e., other user/program actions will potentially interrupt the execution of the chore.) If the cpu is idle, the chore will be performed every waitTicks time intervals. Ticks are 60ths of a second, so if you enter 60 the chore will attempt to send out it's message every second. The message sent is TIMERTICK. See the Breakout.mox model for an example of timer events.	P
Stoptimer()	This procedure stops the idle timer chore started by startTimer above.	P

Date and time	Description	Returns
TickCount()	Clock count (in 60ths of a second) since the computer was powered up. This is useful for timing operations.	I
TimeToString(integer timeVal)	String containing the time in the localized time format (usually Hour:Minute:Second) corresponding to integer *timeVal*.	S
WaitNTicks(integer numTicks)	Waits for the number of ticks (60ths of a second) before returning. This is useful for slowing simulations down and for synchronizing communication protocols with real time.	P

Time units

These functions convert local time units defined within blocks to the global time unit defined in the Simulation Setup (see "Units of time" in the main Extend User's Guide). GetTimeUnits and ConvertTimeUnits use the following values for Time Units.

Time Unit	Value
Generic Time Units	1
Seconds	3
Minutes	4
Hours	5
Days	6
Weeks	7
Months	8
Years	9

Time Units	Description	Returns
ConvertTimeUnits (real value, integer fromType, integer toType)	Converts a value from one type of time unit to another.	R
GetTimeUnits()	Returns the Currently selected Time Units from the Simulation Setup dialog.	I
SetTimeConstants (real hInADay, real dInAWeek, real dInAMonth, real dInAYear)	Sets the time unit conversion values. These are specified in the Run Setup dialog.	P

Time Units	Description	Returns
SetTimeUnits(integer value)	Sets the Time Unit parameter in the Simulation Setup dialog.	I

Debugging

Also see the Abort statement in “Control statements and loops” on page P70 and UserError in “Alerts and prompts” on page P121.

Debugging	Description	Returns
AbortAllSims()	Aborts the simulation and all multiple simulations. Note that the ABORT statement only aborts the current simulation.	P
AbortSilent()	Aborts the simulation run without giving any error messages.	P
DebuggerBreakpoint (long trueFalse)	If called with a true value will act as if a source debugger breakpoint has been set at the line of code where the function is called. This function is useful in debugging the CREATEBLOCK message and in putting a global breakpoint for all blocks of a type.	P
DebugMsg(string errorString)	Operates the same as the UserError function. The difference is that this function will flag the block as having debugging code. The next time a library containing any blocks having these functions is opened, Extend will issue a message warning that the blocks still have debugging code. This function automatically displays the simulation time and the block number of the block from which the function was called.	P
DebugWrite(integer fileNum, string errorString, delimStr, integer tabCR)	Operates the same as the FileWrite function. The difference is that this function will flag the block as having debugging code. The next time a library containing any blocks having these functions is opened, Extend will issue a message warning that the blocks still have debugging code.	P
FreeMemory(integer memoryType)	Returns the amount of memory available to Extend. The *memoryType* argument determines what type of memory will be checked. Macintosh: 1 - Total memory 2 - Contiguous memory Windows: 1 - Total memory 2 - Resources (only for Windows 3.1 and Windows 95)	I
PauseSim()	Pauses the simulation until you choose Resume from the Run menu or click the Resume button.	P

Debugging	Description	Returns
SelectBlock(integer trueFalse)	Selects the block and scrolls to it if the argument is TRUE, unselects it if the argument is FALSE.	P
SelectBlock2(integer block, integer trueFalse)	This is same as SelectBlock(), except it refers to a global block number.	P

Web and Help connectivity

See also the ShowFunctionHelp function which brings up a list of Extend's functions and arguments. This function is listed with "Equations" on page P96.

Help	Description	Returns

CallHelp(string fileName, integer command, integer data, integer html)	Calls the WinHelp function. Currently only implemented on windows. See your Microsoft documentation for the WinHelp and HTMLHelp Windows API calls for more information.
	The html flag takes the following values;
	0) Calls WinHelp. (Opens compiled help files. Typically have an extension of .hlp.)
	1) Calls HTMLHelp. (Opens compiled html help files. Typically have an extension of .chm.)
	This function can be used to bypass the standard Extend help system via the following simple code in the 'on helpbutton' message handler;
	`On helpbutton`
	`{`
	`CallHelp("C:\helpfile.chm", 1,1,1);`
	`Abort;`
	`}`
		P

OpenURL(string theURL)	Access the specified URL using your computers default browser. Returns non-zero error code if it fails.	I
ShowBlockHelp (integer block)	Opens the help dialog, showing the on-line help for any global block. For example, use the function GetEnclosingHBlockNum to get the global block number of an enclosing hierarchical block to show its help under ModL code control.	P
ShowHelp()	Opens the help dialog, showing the on-line help for the block.	P

Models, notebook

Models, notebook	Description	Returns
GetGlobalSimulationOrder(integer blockNum)	Returns the actual simulation order index of a block.	I
GetLibraryInfo (integer blockNum, integer specify)	Function to return whether a library is of a certain type. specify takes the following values: 1: RunTime (others will be defined in the future)	I
GetLibraryPathName (integer blockNum, integer specify)	Function to return the pathname of a library file, given a block in that library. specify takes the following values; 1: pathname without library name 2: just library name	S
GetModelName()	Returns the string that is the model's file name. This is useful when writing out debugging information or in certain user alerts.	S
GetModelPath (string modelName)	Returns the pathname to the specified model, but does not include the model name. The model needs to be open.	S
GetSerialNumber()	Returns the serial number of the unit as a string.	S
GetSimulationPhase()	returns the phase of the simulation. 0 - Not currently running a simulation 1 - CheckData 2 - StepSize 3 - InitSim 4 - Simulate (main simulation loop) 5 - FinalCalc 6 - BlockReport 7 - EndSim 8 - AbortSim	I
GetWindowsHndl (long which)	(Windows Only) returns the Windows API handle of the main window in the Extend application. Currently the *which* parameter is unused. It should be set to 0 for the Main window handle. This handle is used to pass to a Windows DLL. It is not used in ModL functions.	I
OpenNotebook()	Opens the model notebook.	P
ResumeSimulation()	Resumes the simulation and returns immediately.	P

Models, notebook	Description	Returns
RunSetup(trueFalse)	Opens Run Simulation dialog and sets up default OK button instead of Run Now button. Returns TRUE if OK is clicked. If *trueFalse* is TRUE, show RunNow button. If *trueFalse* is FALSE, hide RunNow button.	I
RunSimulation(trueFalse)	Runs the model. If the argument is TRUE, the function puts up the Simulation Setup dialog, then runs the simulation if the user chooses OK; if the argument is FALSE, the model is run directly. This function returns TRUE if the simulation ran to completion with no errors and was not stopped, otherwise returns FALSE. See the SetRunParameters() function in Scripting.See "Scripting" on page P157.	I
SaveModel()	Saves the model and updates any publishers.	P
SaveModelAs(string fileName)	Saves the active model under the name and path defined in fileName. Returns a zero if successful, a negative number otherwise.	I
SetBlockSimulationOrder (long blockNum, long newOrder)	Sets the simulation order value of the specified block. This is only useful, or allowed if the model simulation order has been set to custom simulation order. This function returns a zero if successful, and the following error codes if it fails. -1: can't be set during a simulation. -2: neworder must be greater then zero. -3: no active model document. -4: sim order is not set to custom. -5: no such block.	I
SetModelSimulationOrder (long neworder)	Sets the type of simulation order to be used in the model. Types are: 0: left to right 1: not used 2: flow 3: custom This function returns a zero if successful, and –1, -2, and –3, as described under SetBlockSimulationOrder.	I
SpinCursor()	Spins the beachball cursor. Used to alert the user that a time consuming calculation is taking place.	P

Platforms and versions

These functions allow you to determine the version number of Extend and the platform on which it is running.

Platforms and versions	**Description**	**Returns**
GetCurrentPlatform()	Determines the operating system under which Extend is currently running. This function returns 0 for 68K Macintosh, 1 for Power Macintosh, 2 for Windows 3.1/Win32s, 3 for Windows NT, and 4 for Windows 95.	I
GetExtendVersionString()	Returns the version number of Extend as a string.	S
GetExtendType ()	Returns the type of the Extend application. normal: 0 player: 1 student: 2 runtime: 3 demo: 4	I
GetFileReadVersion()	This function will return the version of the file being read, or previously read. It can be used in the On BlockRead message handler to determine the version of the file that is currently in the process of being read. In conjunction with the ResizeDTDuringRead function, it will allow you to inform Extend that a DataTable has changed size from the size it was in an older version. (This is useful for using the Dynamic datatable function without breaking existing models.)	R
GetFileReadVersionString()	Returns the version of the file being read, or previously read, as a string. This is similar to the GetFileReadVersion function, with the difference that the result is a version string, not a real number. Please note that this function returns the complete version string, in the form ‘major version. minor version.bug fix version', unlike the GetFileReadVersion function which just returns the major version. This function will return an empty string for files earlier then version 4.0.	

Chapter 4: ModL Programming Techniques

Procedures and suggestions for how to create and modify ModL code

"In baiting a mousetrap with cheese, always leave room for the mouse."
— Hector Hugh Munro

Writing ModL code takes much of the same talents as writing C programs. However, Extend provides tools to help make writing block code easier and safer. This chapter covers techniques you can use when creating blocks to make your models run more efficiently.

Running a simulation

It is useful to understand the steps that Extend takes when it runs a simulation so that you can get a feeling for what parts of the process are relevant to your models. The following sections give a step-by-step breakdown of how Extend runs simulations and how it decides what to do next. The first section covers continuous simulations; the second section discusses discrete event simulations.

How Extend runs continuous simulations

Extend keeps many global variables handy so that it can compute how to run the model. Five of the options in the Continuous tab of the Simulation Setup dialog become variables that are used to determine how the model is run:

Option	Variable
End simulation at time	EndTime
Start simulation at time	StartTime
Number of runs	NumSims
Time per step	DeltaTime
Number of steps	NumSteps

Before anything else happens in the simulation run, these variables are used to calculate an initial value for the DeltaTime or NumSteps variables. The variable that is calculated depends on what is selected in the Simulation Setup dialog: if “Number of steps” is selected, Extend calculates the related variable DeltaTime; if “Time per step” is selected, Extend calculates NumSteps. The formulas used are:

```
DeltaTime = (EndTime-StartTime)/(NumSteps-1);

NumSteps = Floor (((EndTime - StartTime) / DeltaTime) + 1.5)
```

This initial value is the value that the blocks see during the CheckData and StepSize simulation messages.

Pseudocode of the simulation loop

The following is a *pseudocode* description of Extend’s continuous simulation loop. Pseudocode is used by programmers to describe the controlling logic of a program in a form that is a combination of English and programming concepts.

```
Calculate simulation order of blocks;

for CurrentSim = 0 to NumSims-1
     {
     Send CheckDataMsg to all blocks;// Validate block variables
     if (CheckDataMsg aborts)
          Abort Simulation and select bad block;

     Send StepSize to all blocks;// Continuous models can change stepsize
     Send InitSimMsg to all blocks;// Initialize block variables

     CurrentTime = StartTime;

     for CurrentStep = 0 to (NumSteps-1)     // In discrete event models,
          {                                  // simulation time is controlled
          Send SimulateMsg to all blocks;    // by the Executive block, not
                                             // by this loop.
          if (Abort was encountered)
               {
               Send AbortSim to all blocks;
               Jump out of loop to ExcecuteEndSims;
               }

          CurrentTime = CurrentTime+DeltaTime;
          }


     Send FinalCalc to all blocks;// Performs final statistical calculations
     Send BlockReport to all reporting blocks;// Report results
     ExecuteEndSims:
     Send EndSimMsg to all blocks;// Clean up variables, dispose arrays
     }
```

The next several sections discuss the meaning of this pseudocode.

Simulation order

This section calculates the simulation order, either flow or left-to-right, as discussed in "Simulation order" in the main Extend User's Guide.

Initialization

The pseudocode:

```
for CurrentSim = 0 to NumSims-1
```

instructs Extend to execute the contents of the "for" loop NumSims times. You set NumSims in the "Number of runs" entry box in the Simulation Setup dialog. The value of the variable CurrentSim varies from 0 to NumSims-1. This is the logic that defines the number of simulations that will be run. If an "abort" statement is executed during a simulation, the abort will halt the current simulation, increment NumSims by 1, and run any remaining simulations.

The first thing done inside this loop is to send the CheckData system message to all the blocks in the simulation. The CheckData message handlers in the various blocks should check the values of dialog items and connections to see if they are valid. If any of them abort, Extend recognizes that fact, and the simulation aborts with the block selected.

Next, the StepSize message is sent to all blocks. In continuous models, StepSize is used to manipulate the DeltaTime variable. The smallest value of DeltaTime specified by any block is found and DeltaTime is set to that value. Note that this DeltaTime overrides the value initially calculated. If AutoStep Slow was selected, the returned value of DeltaTime is then divided by 5 for more accurate simulation results. The blocks that use the StepSize message in this way are in the Electronics and Filter libraries because they need a calculated value of DeltaTime to get accurate results.

Finally, in continuous simulations, a new value of NumSteps is determined based on DeltaTime, StartTime and EndTime.

The formula is:

```
NumSteps = Floor(((EndTime - StartTime) / DeltaTime) + 1.5)
```

This value is used to govern the actual simulation loop. Because DeltaTime is used to calculate NumSteps and a rounded value is used for the number of steps, it is possible for the EndTime of the simulation to be exceeded slightly.

The InitSim message is then sent to all blocks. This message handler is used to initialize variables within each block.

Step loop
The inner loop:

```
for CurrentStep = 0 to (NumSteps-1)
```

is where Simulate messages are sent to all of the blocks in the order determined by the connections. In continuous simulations the loop is governed by CurrentStep and NumSteps only. NumSteps is the variable that is used to end the simulation, and the actual number of Simulate messages (steps) that occurs is NumSteps. That is, if 2 is entered for the “Steps” option in the Simulation Setup dialog, NumSteps becomes 2, the simulation step loop sends Simulate messages to the blocks for CurrentStep equal to 0, and then 1. (If you want to run a simulation for only a single step, set “Steps” in the Simulation Setup dialog to 1).

CurrentTime is incremented by DeltaTime for each step of the loop, but otherwise has no effect on this loop. This means that you can change CurrentTime and DeltaTime without affecting the loop (although changing DeltaTime can have unexpected results, as discussed in the Note below). It also means that you can change CurrentStep or NumSteps to keep the loop running longer or stop it prematurely.

You can set CurrentTime in any block to any value you wish, although you might want to have some logic in the ModL code to prevent conflicts between blocks setting CurrentTime after other blocks have already changed it. Use Extend's global variables to help establish some safeguards against these conflicts.

Note Changing CurrentTime can have unexpected results. In most cases, it is preferable to allow Extend to control the CurrentTime variable.

Final messages

Upon completion of the simulation, Extend sends out the FinalCalc system message to all blocks. This message handler is used to perform final calculations for all time dependent statistics.

Extend then sends the BlockReport system message to all blocks that have been selected for a report. This message handler is responsible for writing all report data to the report text file.

Finally, Extend sends the EndSim system message to all blocks. The EndSim message handler is used for any "clean up" activities such as disposing of any dynamic arrays to free up memory.

Aborting multiple simulations

If the Abort statement is executed during the simulation, it only aborts the current simulation. If you want to abort all simulations (for example, when you have specified in the Simulation Setup dialog that more than one simulation should be run), use the AbortAllSims() function instead of the Abort statement.

How Extend runs discrete event models

For discrete event models, the DeltaTime and NumSteps variables (described on page P174) are ignored, because the Executive block calculates CurrentTime based on the time of the next event.

Pseudocode of the simulation loop

The following is a pseudocode description of Extend's discrete event simulation loop:

```
for CurrentSim = 0 to NumSims-1
     {
     Send CheckDataMsg to all blocks;
     if (CheckDataMsg aborts)
          Abort Simulation and select bad block;

     Send StepSize to all blocks;
     Send InitSimMsg to all blocks;

     CurrentTime = StartTime;

// simulation Phase is controlled by the Executive block
Do until simulation end reached (currentTime >= EndTime or number of events
     {
     Determine the block(s) with the next event time
     Set currentTime to the next event time
     Send message to every block whose nextTime == CurrentTime
     Send message to every block that has posted a 0 time event (rescheduled)
```

```
    }

    Send FinalCalc to all blocks;
    Send BlockReport to all reporting blocks;
    ExecuteEndSims:
    Send EndSimMsg to all blocks;
    }
```

The next several sections discuss the meaning of this pseudocode.

Initialization

The pseudocode:

```
for CurrentSim = 0 to NumSims-1
```

instructs Extend to execute the contents of the "for" loop NumSims times. You set NumSims in the "Number of runs" entry box in the Simulation Setup dialog. The value of the variable CurrentSim varies from 0 to NumSims-1. This is the logic that defines the number of simulations that will be run. If an "abort" statement is executed during a simulation, the abort will halt the current simulation, increment NumSims by 1, and run any remaining simulations.

The first thing done inside this loop is to send the CheckData system message to all the blocks in the simulation. The CheckData message handlers in the various blocks should check the values of dialog items and connections to see if they are valid. If any of them abort, Extend recognizes that fact, and the simulation aborts with the block selected.

Next, the StepSize message is sent to all blocks. In discrete event models, this message is used to setup the attribute global arrays (see "Attribute global arrays" on page P199).

The InitSim message is then sent to all blocks. This message handler is used to initialize variables within each block.

Simulation Phase

The Discrete Event library relies on the Executive block rather than Extend to control CurrentTime and NumSteps. Discrete event simulations are event based, as opposed to continuous simulations which rely on time being incremented in an orderly fashion. In discrete event simulations, NumSteps is modified constantly and the Executive block monitors and manipulates CurrentTime based on the flow of items and events through the simulation.

At each simulation step, the executive searches through a list of event scheduling blocks and finds the nearest future event time. The Executive then sends a message to each of the event scheduling blocks whose event time is equal to the nearest event time. This may cause other blocks to post zero time events (rescheduling themselves). The Executive sends a message to each one of the blocks on the zero time event list (see "Timing for discrete event models" on page P195 for a detailed discussion).

Final messages

Upon completion of the simulation, Extend sends out the FinalCalc system message to all blocks. This message handler is used to perform final calculations for all time dependent statistics.

Extend then sends the BlockReport system message to all blocks that have been selected for a report. This message handler is responsible for writing all report data to the report text file.

Finally, Extend sends the EndSim system message to all blocks. The EndSim message handler is used for any "clean up" activities such as disposing of any dynamic arrays to free up memory.

Aborting multiple simulations

If the Abort statement is executed during the simulation, it only aborts the current simulation. If you want to abort all simulations (for example, when you have specified in the Simulation Setup dialog that more than one simulation should be run), use the AbortAllSims() function instead of the Abort statement.

Equation block programs

You saw in "Equation editor" in the main Extend User's Guide, how to use the Equation blocks from the Generic and Discrete Event libraries (*Macintosh*: Generic Lib and Discrete Event Lib; *Windows*: generic.lix and de.lix) for entering simple equations. The Equation blocks are much more powerful than that, however. You can enter short ModL programs into the blocks, including variable definitions. Think of the Equation blocks as a method for including small programs on the fly without having to design a new block.

You can also create your own blocks that take in equations using the EquationCompile and EquationCalculate functions described on page P96. For example, you might want to do this if you are building scientific or engineering blocks and can't predict ahead of time what formulas you will need.

Advanced dialog techniques

In the past few years, computer users have become accustomed to having dialogs be more interesting and informative.

With a bit of creativity, you can make dialogs that give a great deal of information about the state of a simulation. Also, you can use dialogs to help you debug your models-in-progress. This section gives many ideas about how you can use dialogs to make your models more exciting.

Changing dialog text as the simulation runs

The Miles block example in "Building a block that converts miles to feet" on page P38, showed the usefulness of keeping a dialog open and watching the numbers in the entry boxes change as the simulation progresses. This is useful for watching numbers, but it could also be used for displaying text or messages that might change during the simulation. This is often more useful than displaying alerts since alerts stop the simulation until you click on one of their buttons.

For example, assume that you are modeling a factory that has three shifts. The name of the shift (day, night, swing) changes depending on the current time. Assume that you have a block with a static or editable entry box named “shift” that you want to modify. Also assume that the simulation times are in minutes.

You could display the time in the “Shift” dialog item as a number, but then you have to convert the time to the shift name in your head:

```
Shift = (CurrentTime / 60) mod 24;
```

Instead, you could convert the time in the block’s code and display text in the dialog:

```
...
string name[2];
name[0] = "Day"; name[1] = "Night"; name[2] = “Swing”;
...
Shift = name[((CurrentTime / 60) mod 24) / 8];
```

Changing dialog text in response to a user’s action

ModL lets you change labels, text, datatable column titles, and button titles in a dialog at any time. This lets you decide what to show based on what is done in the dialog, such as clicking on a control item (button, radio button, checkbox, switch, or slider). An example of this is the Input Random Number block in the Generic library (*Macintosh*: Generic Lib; *Windows*: generic.lix”), which displays different parameter labels depending on the statistical distribution that is selected.

When a control item is clicked, two things happen:

- The value of the radio button’s variable name is set to TRUE; switches and checkboxes are toggled from FALSE to TRUE and vice versa; and sliders are changed in value. Plain buttons don’t have a value but their titles can be changed.
- Extend sends a message to any message handler with the control item name. This allows the ModL code to take a special action.

For example, assume that you want to show values as either octanes or cetanes. There are two radio buttons that let you decide which unit to show:

```
// ShowOctaneValues radio button was clicked
on ShowOctaneValues
{
     // set static text label above data table
SetDataTableLabels(“dataTable”, “Octane Values”);

for (row=0; row<numRows; row++)
     for (col=0; col<numColumns; col++)
          dataTable[row][col] = octaneVals[row][col];
}
```

```
// ShowCetaneValues radio button was clicked
on ShowCetaneValues
{
     // set static text label above data table
SetDataTableLabels("dataTable", "Cetane Values");

for (row=0; row<numRows; row++)
     for (col=0; col<numColumns; col++)
          dataTable[row][col] = cetaneVals[row][col];
}

// When the dialog is opened by the user
on DialogOpen
{
   // Static labels are not "remembered" when dialog is closed
   //   so when the user opens the dialog, restore the titles here

if (ShowCetaneValues)                 // ShowCetaneValues was TRUE
     SetDataTableLabels("dataTable", "Cetane Values");   // restore title
else                                  // else ShowOctaneValues TRUE
     SetDataTableLabels("dataTable", "Octane Values");   // restore title
}
```

See also "Buttons" on page P27 and "Static text (label)" on page P28.

Note If the octane and cetane arrays were dynamic, the DynamicDataTable function could be used to switch the data table between the two dynamic arrays. This would execute more quickly and require less code.

Changing the title of a radio button or checkbox

If you add a radio button or checkbox to a dialog and leave the title blank, then superimpose a static text item over where the button's title would normally appear, you can set the static text item to any string in the ModL code and it will appear as the button's title. The ModL code can thus change the title, and meaning, of the control at any time. For example:

```
...
staticTextTitle = "Interest in percent";
...
staticTextTitle = "Interest in decimal";
...
```

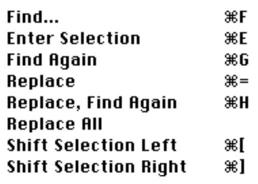
Note The contents of static text dialog items are not stored with the block on the model. Their values are stored with the block structure when building the block. Therefore, if you want to change the value of static text, you must set it whenever a block or model is opened using the on OpenModel and on DialogOpen message handlers.

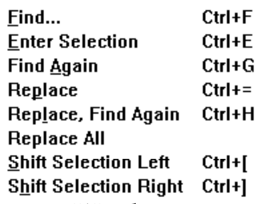
Formatting and maintaining ModL code and help

The following sections give tips on how to edit and format block help and code and discuss using include files to simplify programming tasks that are repeated in multiple blocks.

Searching and replacing

The Define menu has many commands that help you edit your block's code. The editing commands at the bottom of the Define menu are:

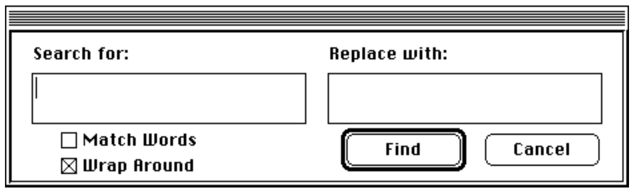



Find... Ctrl+F
Enter Selection Ctrl+E
Find Again Ctrl+G
Replace Ctrl+=
Replace, Find Again Ctrl+H
Replace All
Shift Selection Left Ctrl+[
Shift Selection Right Ctrl+]

Macintosh Windows

Editing commands

The Find command lets you define a string to find as well as a replacement string for the Replace command. The dialog looks like:



Find dialog

The Enter Selection command gives you an easy method for finding the next instance of some text that is in your code:

- Select the text.
- Choose Enter Selection from the Define menu or (*Macintosh:* press ⌘-E; *Windows*: press CTRL-E).
- Choose Find Again from the Define menu (*Macintosh:* press ⌘-G; *Windows*: press CTRL-G or F3).

The Enter Selection command places the selected text in the Search for: box. The Replace command uses the Search for: and Replace with: strings in the Find command's dialog. Thus, you can set up a search and a replace in one dialog. If there is no string in the Replace with: entry box, the text you searched for is deleted (that is, replaced with nothing).

Using the Go To Line command causes the code pane to scroll to that line number.

Formatting tips

The Shift Selection Left and Shift Selection Right commands (*Macintosh:* press ⌘-[and ⌘-]; *Windows*: press CTRL-[and CTRL-]) in the Define menu let you change the tabbed indentation on lines in your ModL code. For example, if you copy lines from another block's code that are at a different indentation level, select them and use the appropriate command to move them to the correct level. These commands only work if you have used the Tab key for indentation.

You can copy names from the variables pane of a block's structure window into the code or the help panes with the Copy and Paste commands or with drag and drop editing (see "Drag and drop editing" in the main Extend User's Guide for a detailing discussion). You can also copy and paste from Extend's Help dialog. This lets you copy function definitions and arguments directly into your code.

You can change the default font and size for the text of the ModL code using the Preferences command in the Edit menu. The text on your icons can also have character formatting.

Your help text can have character formatting. Simply select the text and give commands from the Text menu. For example, to make a phrase bold, select it and choose Bold from the Text menu. You can also specify a color for the text from the color area of the toolbar.

Include files

Extend allows you to use include files in your ModL code. Include files are standard header files that you can put in any ModL code. Include files can contain any ModL commands such as definitions, assignments, and functions. The purpose of an include file is to help simplify maintenance of a library of blocks that use similar variable definitions and functions.

To start a new include file, be sure a structure window is active and choose the New Include File command from the Define menu. This opens an untitled text file window. Type the statements you want in this window and choose the Save Include File As command from the File menu. This saves the include file into the *Extensions* folder. For all platforms, the include file's name *must* end with ".h" (like standard C include files) and the file *must* be in the *Extensions* folder that resides in the same folder or directory as Extend.

To reference an include file from a block's code, enter a line in the format:

```
#include "filename.h"
```

or

```
#include <filename.h>
```

For example, if the name of the include file you created is "New_Defs," you would use the command:

```
#include "New_Defs.h"
```

You can have as many different include files as you wish, as long as all of them reside in the *Extensions* folder.

Animation

The ModL animation functions listed in “Animation” on page P123, make it easy to add animation to your blocks. This section discusses the general procedures in creating animated blocks, then shows some examples of how you might add animation to the blocks you build.

Note Even if Show Animation from the Run menu is not selected, animation may still be available. The Show Animation command only controls whether animation is shown *during the simulation run* and then *only* during the repetitive execution of SIMULATE message handlers. At all other times, animation is always available, even if Show Animation is not selected. For instance, animation is available when the user makes any changes in a block's dialog, whether Show Animation is selected or not and regardless of whether the simulation is running. When Show Animation (in the Run menu) is selected, animation is available at all times.

This is a very powerful feature because it lets you build blocks which show their initial status or final values without having to turn on animation. For example, if a checkbox is clicked, the dialog can send an “on myCheckBox” message to the block and the block can animate the change on its icon, as seen in the Miles block in “Adding animation” on page P53.

You do not have to have Show Animation selected to show QuickTime movies (Macintosh only).

Overview

As described in detail below, the basic steps for adding animation to a block are:

- Decide how you want the block to animate, including the shape, color, and pattern that the animation objects should have.
- Create the animation objects in the icon pane of the block’s structure window.
- Initialize the animation objects in the CheckData or InitSim handlers.
- Add code to update the animation object at the correct times, including (if necessary) code to slow the display so that it isn’t too fast to be seen. For continuous blocks, this code will be in the Simulate message handler. For discrete event blocks, this code will be in the procedure or message handler appropriate for the data that you want animated (for example, if you are animating an item entering a block, the code would go in the ItemIn message handler).

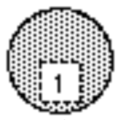
Deciding how to animate the icon

There are several ways to animate a block’s icon. You can show and hide text or a shape (such as a rectangle, rounded rectangle, oval, or level), move a shape within or outside of the icon (even along the connection lines), show a changing level, stretch and reduce a shape’s size, show a picture or a movie, or change colors, patterns, or text. Some animations (for example, moving or stretching) will cause models to run slower than other types (for example, changing color or text).

While it is common that icons are animated, it is also possible to animate in the area around icons. See the Planet Dance model (*Macintosh:* Planet Dance in the Science & Engineering folder; *Windows:* Planets.mox in the Sci&Eng directory) for an example of a model that shows animation outside of the icon. Also see the blocks in the Discrete Event library for examples of animating from one block to the other, along connection lines.

Creating animation objects

All animation is done through the use of one or more animation objects that you put in the icon pane. The animation object itself is a resizable rectangle with dotted sides, identified by a unique number. To create an animation object, open a block's structure window and click on the animation tool ( )in the toolbar. Then click and drag in the icon pane to create a rectangular object the size you want. Extend puts the object's number in the item, such as:



Animation object number

You will use that object number in all of the animation functions. Animation objects are always rectangles in the icon pane; the animation functions you use in the block's code determine the characteristics of the object, for example, the shape and color that will show as the block is animated.

Initializing animation objects

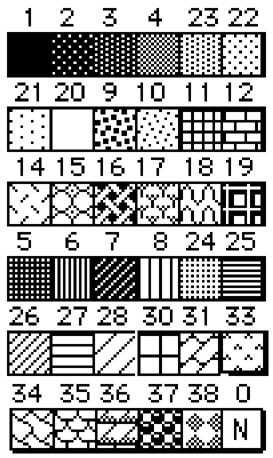
In the block's code, initialize the object in the InitSim message handler. You may also want to put the same code in the CreateBlock handler if you always want the animation object visible, even before an animation is run. The initialization will generally consist of a call to one of the object definition functions (AnimationRectangle, AnimationRndRectangle, AnimationOval, AnimationText, AnimationLevel, AnimationPicture, AnimationMovie, and AnimationPixelRect), a call to AnimationColor (to set the color and pattern for the object), and perhaps a call to AnimationShow so that the object is visible at the beginning of the simulation (for example, for showing initial text or color). For example, you might have:

```
on initSim
{
AnimationOval(1);                          // Set 1 to oval
AnimationColor(1, 0, 65535, 65535, 1);     // Set 1 to red (HSV)
                                           //   solid (pattern 1)
AnimationShow(1);                          // Optional
                                           //   makes the object visible
}
```

Note that if you define the animation object in the code as an oval, and resize it on the icon as a square (as was done here), it will show as a circle. If you define it as an oval and resize it as a rectangle, it will show as an oval.

The color of an object is set with numbers for hue, saturation, and brightness (value), often called HSV. You can use the color swatches window to determine the HSV values of any color.

The pattern is defined from the patterns in the toolbar. The number for the patterns that appear in the pull-down pattern menu are:



Numbers for patterns

Once animation is performed, animation objects show on the block, covering the icon. If you want your block to see whether or not the Show Animation command from the Run menu is checked (for example, to hide the animation object if animation is not on), use the AnimationOn system variable in the CheckData or InitSim message. It is set to TRUE (1) if animation is on and FALSE (0) if it is not. For example, instead of the above initialization, you might have:

```
on initSim
{
AnimationOval(1);                      // Set 1 to oval
AnimationColor(1, 0, 65535, 65535, 1); // Set 1 to red (HSV)
                                       //   solid (pattern 1)
if (AnimationOn)                       // Animation is on
     AnimationShow(1);                 // Optional: show object 1 now
else                                   // Animation is not on
     AnimationHide(1, FALSE);          // Hide object; it is not
                                       //   outside of icon
}
```

Updating the animation object
As mentioned earlier, if Show Animation is not selected, animation is still available except during the Simulate message. The Show Animation command only affects animation that is specified in the Simulate message. During Simulate, Extend blocks check the state of the Show Animation command in the Run menu to determine whether or not to perform animation. In the Simulate message handler (for continuous blocks) or in the procedure or message handler appropriate for

the data being animated (for discrete event blocks), put the code that checks whether you want to change the animation object, for example its position, color, or text. To speed execution of the block, be sure to only call an animation function when the object has changed. If the animation object shows too fast to be seen, you may want to include a call to the WaitNTicks function. Note, however, that this will also slow down the simulation.

Animating hierarchical blocks

To animate a hierarchical block's icon, you have to include in the submodel a block that contains code for the animation, but no animation object. You create the animation object on the hierarchical block's icon. The block in the submodel is used to read the values from other submodel blocks, and control the animation of the hierarchical block's icon based on those values. In order to do this, it references the hierarchical block's animation object using the negative of the object number. For example, if the number of the animation object on the hierarchical block's icon is 2, the block in the submodel would reference animation object -2 throughout the ModL code. Note that you can animate any number of levels up to an enclosing hierarchical block, and that animating hierarchical blocks is the only time when you would use negative values to reference an animation object. The Animate Value and Animate Item blocks in the Animation library contain code which allows them to animate hierarchical blocks. This use is described in “Adding animation” in the main Extend User's Guide.

Showing and hiding a shape

You might want an animation object to show if an input value met some condition or became true, or if an item arrived in the block or was being processed. Hiding the animation object would then indicate the opposite condition. You might want to have a small object near the connector to indicate item arrivals or meeting a condition, like the select blocks in the Generic library. Or you could have a larger object indicating a true value or processing, such as the activity blocks from the Discrete Event library.

For this example, you use the animation object drawn above. You want a solid red circle to appear in the icon when a value is true (greater than 0.5) and you want to hide it when the value is false. To do this, initialize the animation object and hide it in InitSim. Then put code in the Simulate message to indicate when and how the object should change:

```
on initSim
{
AnimationOval(1);                       // Set 1 to oval
AnimationColor(1, 0, 65535, 65535, 1);  // Set 1 to red (HSV),
                                        //   solid (pattern 1)
AnimationHide(1, FALSE);                // Hide object; it is not
                                        //   outside of icon
}
```

```
on Simulate     // Or appropriate procedure or message handler
{
...;
if (ConditionIsTrue)
     AnimationShow(1);                    // Show object now
else
     AnimationHide(1, FALSE);             // Hide object; it is not
                                          //   outside of icon
}
```

Moving a shape

Assume that you want the animation object drawn above to move between the original position and a position 20 pixels higher than the original depending on the values received. The input at the connector called "con1in" takes in real values from 0 to 1, with 1 indicating the highest desired input. Since in screen coordinates up is considered negative relative to the starting position, you must call AnimationMoveTo with a negative number to move the object up. Your code might be:

```
integer     obj1Loc;                     // Save the object's location
real        clipped;
integer     testLoc;
. . .

on initSim
{
AnimationOval(1);                        // Set 1 to oval
AnimationColor(1, 0, 65535, 65535, 1);   // Set 1 to red (HSV)
                                         //   solid (pattern 1)
if (AnimationOn)                         // Animation is on
     AnimationShow(1);                   // Show object now
else                                     // Animation is off
     AnimationHide(1, FALSE);            // Hide object; it is not
                                         //   outside of icon
}

on simulate     // Or appropriate procedure or message handler
{
clipped = Min2(con1in, 1.0);             // Highest position is 1
clipped = Max2(clipped, 0.0);            // Lowest position is 0

testLoc = int(clipped*-20.0);            // Scale to range, upwards
if (testLoc != obj1Loc)                  // Do nothing if not changed
     {
     obj1Loc = testLoc;
     AnimationMoveTo(1, 0, Obj1Loc, FALSE); // Move to new location
     }
}
```

Changing a level

Sometimes you want to display a changing level, such as in a water tank. For this example, you use the animation object drawn above. A simple block that displays a changing level when its input connector varies between 0.0 (lowest level) and 1.0 (highest level) might look like:

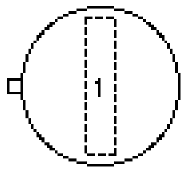
```
on InitSim
{
// Initialize animation object number 1 as a "level" shape
AnimationLevel(1, 0.0);                    // Initialize level to low
AnimationColor(1, 0, 65535, 65535, 1);     // Set 1 to red (HSV),
                                           //   solid (pattern 1)
if (AnimationOn)                           // Animation is on
     AnimationShow(1);                     // Show level now
else                                       // Animation is off
     AnimationHide(1, FALSE);              // Hide level; it is not
                                           //   outside of icon
}

on simulate     // Or appropriate procedure or message handler
{
AnimationLevel(1, con1in);     // Input connector value (0.0 to 1.0)
                               //   controls the height of the level
}
```

If you want the level to reflect values other than 0 to 1, you need to scale the input values to correspond to that range. See the Accumulate block in the Generic library for an example of changing a level.

Stretching a shape

You can stretch an animation object horizontally or vertically, or both at the same time (circular), relative to its original position on the icon. For example, you might want to do this to show the relative size of an item (for instance, based on an attribute value) or to indicate a direction of flow.

The following code stretches the object vertically inside the icon. This method uses the exact size of the animation object as it is drawn on the icon to determine the boundaries for the stretch. The input at the connector called "con1in" takes in real values from 0 to 1, with 1 indicating the highest desired input. For this example, draw an animation object like:



Animation object for stretching

```
integer     origWidth, origHeight;         // Boundaries of the object
integer     pixels;                        // How far to stretch
real        clipped;                       // Amount to change
. . .
```

```
on InitSim
{
AnimationRectangle(1);                     // Set 1 to rectangle
AnimationColor(1, 0, 65535, 65535, 1);     // Set 1 to red (HSV),
                                           //   solid (pattern 1)
origWidth = AnimationGetWidth(1, TRUE);    // Get original width
origHeight = AnimationGetHeight(1, TRUE);  // Get original height

if (AnimationOn)                           // Animation is on
     AnimationShow(1);                     // Show object now
else                                       // Animation is off
     AnimationHide(1, FALSE);              // Hide object; it is not
                                           //   outside of icon
}

on simulate     // Or appropriate procedure or message handler
{
. . .
clipped = Min2(conlin, 1.0);               // Highest position is 1.0
clipped = Max2(clipped, 0.0);              // Lowest position is 0.0
pixels = clipped * origHeight;             // Calculate stretch
// Subtract the pixels to go upward (negative is up relative to base)
AnimationStretchTo(1, 0, origHeight - pixels, origWidth, pixels, FALSE);
}
```

If you want to have the shape stretch outside of the icon, you would set the last argument in AnimationStretchTo to TRUE and increase the value for pixels so it will stretch outside the icon. Stretching or moving outside the icon slows animations considerably and might cause the icon to flash.

Showing a picture or movie on an icon

You can also have a picture or a QuickTime movie (Macintosh only) show on an icon in response to some occurrence in the block. You can even alter the speed of the movie during the simulation and play the sound (if the movie has sound). For example, the code for a picture might look like:

```
on initSim
{
AnimationPicture(1, "MyPicture", FALSE); // Set 1 to picture, not scaled
AnimationHide(1, FALSE);                 // Hide object; it is not
                                         //   outside of icon
}
on simulate     // Or appropriate procedure or message handler
{
. . .
if (ConditionIsTrue)
     AnimationShow(1);                   // Show picture
else
     AnimationHide(1, FALSE);            // Hide object; it is not
                                         //   outside of icon
}
```

The code for a movie might look like:

```
on simulate             // Or appropriate procedure or message handler

. . .
if (ConditionIsTrue)
    {
            // optionally wait here until the previously started movie is done
    while ( ! AnimationMovieFinish(1))
            // AnimationMovieFinish(1) returns FALSE if it is still playing
          ; // loop here until AnimationMovieFinish(1) returns TRUE

    // Start "MyMovie" in animation object number 1 at normal speed,
    // sound on, async
    AnimationMovie(1, "MyMovie", 1.0, TRUE, TRUE);
    }
}
```

In order to use a picture or a movie, it must be a resource in the System, in Extend, or in a file in the Extend's extensions folder or directory as discussed later in this chapter.

Moving a picture along the connection line between two blocks

You might want to show a picture traveling along the connection line between two blocks. This is an effective way to animate individual items as they flow through a model. As shown above, you can assign a picture to an animation object. By calling AnimationBlockToBlock, you can move the animation picture from one block to the next along the connection line.

Arguments for AnimationBlockToBlock include the animation object number and the block numbers and appropriate connector numbers of the two blocks you wish to animate the picture between. Typically you do not know how blocks will be connected until the model is built. Therefore you must determine the values of these parameters by using function calls to MyBlockNumber, GetConNumber, and GetConBlocks. You can call the functions from any block, provided it knows which block is sending and which is receiving.

The following code example illustrates how to determine the appropriate parameter values and make a call to AnimationBlockToBlock. In this example, AnimationBlockToBlock is called from the “sending” block, or the block from which the picture will start moving. The picture will move to the “receiving” block connected to the con1Out connector of the “sending” block.

```
integer array[][2];
...
on InitSim
{
AnimationPicture(1, "MyPicture");     // Set 1 to picture
myNumber = MyBlockNumber();                 // determine "sending" block #
outCon = getConNumber(myNumber, "con1Out"); // determine "sending" connector #
getConBlocks(myNumber, outCon,array);       // what blocks are connected to
                                     //    con1Out?
rBlock = array[0][0];               // determine "receiving" block #
rConn = array [0][1];                      // determine "receiving" connector #
}

on simulate
{
AnimationBlockToBlock(1,myNumber,outConn, rBlock, rConn, 1.0); // animate picture
```

```
                                                                //   between blocks
}
```

Changing a color

If you want to use changing color instead of motion, you might include a call to AnimationColor with a variable for one of the hue, saturation, or brightness arguments:

```
on InitSim
{
AnimationOval(1);                           // Set 1 to oval
AnimationColor(1, 0, 65535, 65535, 1);      // Set 1 to red (HSV),
                                            //   solid (pattern 1)
if (AnimationOn)                            // Animation is on
     AnimationShow(1);                      // Show object now
else                                        // Animation is off
     AnimationHide(1, FALSE);               // Hide object; it is not
                                            //   outside of icon
}

on simulate     // Or appropriate procedure or message handler
{
. . .
hueVal = hueVal+1000;                       // Different colors
AnimationColor(1, hueVal, 65535, 65535, 1);
. . .
}
```

Changing text

You can also animate by changing text in the animation object created at the beginning of this section:

```
string          objText;

on InitSim
{
AnimationText(1, "");                      // Set 1 to blank text, can also use
                                           //   animationTextTransparent()
AnimationColor(1, 0, 0, 0, 1);             // Set 1 to black (HSV),
                                           //   solid (pattern 1)
if (AnimationOn)                           // Animation is on
     AnimationShow(1);                     // Show object now
else                                       // Animation is off
     AnimationHide(1, FALSE);              // Hide object; it is not
                                           //   outside of icon
}
```

```
on simulate     // Or appropriate procedure or message handler
{
. . .
if (temp < 1000)                        // Less than 1000
     objText = "Cool";
else if (temp < 1500)                   // Between 1000 and 1500
     objText = "Med";
else                                    // Greater than 1500
     objText = "Hot";
AnimationText(1, objText);              // Set the text, can also use
                                        //   animationTextTransparent()
. . .
}
```

Animating pixels

You can generate a rectangle of pixels where each pixel can be a different color based on the value for that pixel. For example, you would use this to generate a contour map or give visual display of temperatures over a two-dimensional space. For an example of this, see the Mandelbrot model.

Connectors

Most blocks have connectors. You add and change connectors using the connector tools at the right of the toolbar.

Deleting connectors or changing connector types

Extend keeps an ordered list of the connectors on a block. If you delete a connector on a block used in a model, it changes the connector order. This may cause unexpected results: the connectors could become disconnected, or connected incorrectly. This is not a problem if the block isn't currently in a model, but for blocks in existing models the results could be disastrous.

You can change the type of a connector by selecting it in the icon pane and clicking on the desired connector tool. Thus, if you only need to change connector types, *do not delete the connector*. Instead, simply select it and click on the correct connector tool in the toolbar. If you have to delete a connector on a block which is used in a model, carefully examine the block's connections.

Bidirectional connectors

All Extend connectors are bidirectional. This is very useful if you want blocks to communicate back and forth.

For example, you may want to simulate a network or a bus in which blocks need to both send to, and receive information from, the other members of the network. Because Extend does not allow multiple output connectors to be connected together, you could use only input connectors for all the blocks in the network; output connectors are optional.

For example, you may also want source and sink connections, where a source block can have its output value changed by the sink blocks that are connected to it. This is useful in simulations where one block's reserves are depleted by the other blocks connected to it.

To implement these features in blocks, you can assign or modify the value of an input connector. This feature makes input connectors bidirectional. When the value of an input connector is modified, all connected blocks will see this new value when they get their next Simulate message.

The concept of using an input connector for both input and output is shown in the "Bidirectional Flows" model (*Macintosh:* Bidirectional Flows in the ModL Tips folder; *Windows:* Flows.mox in the ModLTips directory). In the power grid model, a power station supplies energy to cities. Each city block has a single input connector. The block code for the cities subtracts an amount from the input, called "PowerIn", with the statement:

```
PowerIn = PowerIn - amountUsed;
```

This affects the Power Station block by reducing the output power value stored in its output connector. The power station can check its output connector value and see if it went to 0 or became negative, signifying that its power reserves have been depleted by the towns:

```
if (PowerOut <= 0.0)    // check the reserve power
     {
                        // You are out of power. Tell when this occurred.
     UserError("Brownout occurred at time= "+CurrentTime);
     abort;             // Stop the simulation.
     }
```

Initializing connectors

All connectors are initialized by Extend to 0.0 after the CheckData message handlers are executed. If you want to initialize connectors before the simulation runs, do that within the InitSim handler. The value of a connector is used to determine whether the block is connected to another block during the CheckData message handler, and thus cannot be changed there. You can also initialize connectors in the Simulate message handler. You can use an "if" statement such as:

```
if (CurrentStep == 0)      // The first step of the simulation
     conOut = initialValue;
```

The blocks in the Discrete Event library initialize their connectors in the InitSim handler.

How discrete event blocks and models work

Discrete event blocks use some fairly complex code. The following explanation should be useful to anyone trying to program their own discrete event blocks or to alter the blocks that come in the Discrete Event library. (We strongly recommend making copies of blocks from the Discrete Event library, and altering the copies rather then the originals.)

The Make Your Own block in the Make Your Own DE library (*Macintosh:* Make Your Own DE lib; *Windows:* MYO_DE.lix)has sample code and comments that explain how the blocks in that library work. You can use this block as a template for your own discrete event blocks. Also, see the code in the Activity Delay block for an example of how to process items with time delays.

In the following sections, we will examine the messaging system used to schedule events, transfer items through the model, and resolve logic issues. To understand the details of the different types of messages, some understanding of the simulation engine, the underlying data structures, and event handling is necessary.

Timing for discrete event models

To make a model discrete event, you must add the Executive block to the model worksheet. This block does two things: first, it maintains a data structure of information about the items in the model; second, it takes control of the time clock from the Extend application, scheduling events, sending messages to the blocks that scheduled the event, and moving the clock forward to the appropriate time for the next event.

In order to provide true discrete event operation, the simulation clock must move to the exact time of each event. In order to do this, the Executive block creates two dynamic arrays that are used to store future events in the model. The first array is called TimeArray and contains the event times; the second, called TimeBlocks, contains the block number of the block that posted the event. The Executive block passes the TimeArray and TimeBlocks arrays to sysGlobal0 and sysGlobal7 respectively with the statements:

```
sysGlobal0 = passArray(TimeArray);
sysGlobal7 = passArray(TimeBlocks);
```

Note An important consequence of the Executive block controlling the time clock in the model is that the variable numSteps, which in a continuous model represents the number of simulation steps that will occur in the total run, is updated only by the Executive block, and will always have a value of one greater than the number of simulation steps that have actually occurred.

Residence and passing blocks

There are two types of discrete event blocks: *residence blocks* and *passing blocks*:

- Residence blocks are able to contain or hold items for some duration of simulation time. Some residence blocks post events and some do not. Examples of residence blocks are the Queue, FIFO or Activity, Delay blocks.
- Passing blocks pass items through without holding them. These blocks implement modeling operations that are not time based. Passing blocks do not post events and do not get either BlockReceive0 or Simulate messages. Examples include setting an attribute or selecting a path (Set Attribute or Select DE Output blocks).

What messages a block gets depends on the type of block and whether or not it posts events.

Residence blocks that post events

Each residence block that posts events places its block number in a slot in the global TimeBlocks during the initSim message handler. The slot number is an index number assigned in the check-

Data message handler, called “myIndex”. Each residence block also places the time of its next event in a slot in the global TimeArray. For example:

```
TimeArray[myIndex] = nexttime;
```

At the start of a simulation event, the TimeArray (the event list) is searched to find the next event time. A third dynamic array, CurrentEvents, is used to store all of the block numbers that have posted an event at the next event time. The simulation clock is then advanced to the next event time and a *BlockReceive0* message is sent to each block, one at a time, within the currentEvents array. After receiving the message in its “On BlockReceive0” message handler, each block searches its own internal list for the item with the current event time and then attempts to push the item out into the next block in the process. The Activity, Delay and Generator blocks are residence blocks that post events.

Residence blocks that do not post events but receive the Simulate message
As mentioned above, not all residence blocks post events. Some of the residence blocks which do not post events, such as queues and resources, also attempt to move items at event times. They do this in response to a Simulate message. The purpose of this type of block is to pull in items and hold them until there is a downstream capacity. If, at any given event, no items have been pulled in or pushed out, the block will go to “sleep”, ignoring any Simulate messages. It will “wake up” when an item is either pushed in or pulled out and will attempt to process additional items. The Queue, FIFO and Resource blocks are residence blocks that do not post events.

Residence block that do not post future events:
As mentioned above, not all residence blocks post events. Some of the residence blocks which do not post events, such as queues and resources, also attempt to move items at event times. They do them in response to two mechanisms.

The first mechanism is in response to a Simulate message. The purpose of this type of block is to pull in items and hold them until there is downstream capacity. If, at any given event, not items have been pulled in or pushed out the block will go to "sleep" turning off any simulate messages to the block. It will "wake up" when an items is either pushed in or pulled out and will attempt to process additional items.

The second mechanism is the CurrentEvents array in the Executive. If a residence block needs to return from a message but may also need to attempt to pull in or push out items with the same time step, it will send a message to the Executive posting itself on the CurrentEvents array. The Executive sends a BlockReceive0 to all of the blocks listed in the CurrentEvents array. This ensures that the residence block will receive an additional message before the next time step so that additional items can be processed.

The Queue FIFO and Resource blocks are residence blocks that do not post events.

Passing blocks
Passing blocks do not receive Simulate messages. They may use the CurrentEvents array to reschedule themselves. In this case they will receive a BlockReceive0 message. Rescheduling is necessary when the passing block needs to return from a message. However, before the simulation clock advances, it also needs to perform some additional processing. In this case a message is sent to the Executive and the block number is posted on the CurrentEvents array. The Executive sends a BlockReceive0 message to every block entered in the CurrentEvents array.

Item data structures

The Executive block stores information about each item in a discrete event model in a set of four dynamic arrays which it passes into global variables sysGlobal3, sysGlobal4, sysGlobal6, and sysGlobal9 (see "Global variables" on page P82), as well as a set of two global arrays (see "Global arrays" on page P83). The information in all of the arrays is available to every block that needs to access it.

Each item in the model is identified by a unique number that is the index number used to look up the item's information in the dynamic arrays. This is the item's *index*. The arrays contain information about the items, such as values, priorities, attribute information, and so on. When an item is passed through an item connector, it is really the index of the item that is passed. Items are indexed from 1 to n-1, where n is the number of slots in the array. Index 0 is not used as an item index, since its presence at a connector indicates that no item is present.

Passing the index numbers through item connectors allows the blocks to pass a great deal of information with a single number. It also means that all blocks in the model can access any item. Each of the global variables is received into a local array within each block and, after being received, is available in exactly the same way that data in any local array is available.

For example, if there are currently ten items in a simulation, the index number 5 might be passed from one block in the model to the next. When a block received index number 5, it would access the information in the Executive block's arrays for item number 5. For instance, itemArrayC[5][0] would tell if the item could accumulate a cost while itemArrayR[5][1] would tell the item's priority. When a block is done with an item, it passes the index value of that item to the next block.

The Executive block maintains two arrays of real information (itemArrayR[][3], itemArrayC[][10]), an array of integer information (itemArrayI[][5]), and an array of string information (itemArrayT[]). The arrays are:

Real array
The real array itemArrayR[][3], passed through sysGlobal3, contains the following information:

Slot #	Description
0	Value: The number of items that the current item represents. This is used, for example, by the Set Value block in the Discrete Event Library. Item values are used to copy items in queues and resource blocks. They are also used by certain input connectors (such as on the Activity, Service block) to convey additional information to a block.
1	Priority: Used by the Set Priority, Get Priority, and Queue Priority blocks. Note that in Extend the lower the number (including negatives), the higher the priority.
2	User defined real value. This value is left untouched by the blocks in the Discrete Event library.

Cost array

The real array itemArrayC[][10], passed through sysGlobal9, contains information concerning cost resources that are batched with the item. This information is used by residence blocks to calculate the accumulated cost based on the cost rates and the amount of time the item spent in the block.

Slot #	Description
0	Item type: When considering cost, all items can be classified as an item which can accumulate costs (1) or a resource (2).
1	Resource Rate 1: The cost per time unit of a resource batched to the item using the batch block.
2	Batch Number 1: Stores the amount of resource 1 batched to the item.
3	Resource Rate 2: The cost per time unit of a resource batched to the item using the batch block.
4	Batch Number 2: Stores the amount of resource 2 batched to the item.
5	Resource Pool Rate: The accumulated cost per time unit of resources batched to the item using the Queue, Resource Pool block.
6	AGV Rate: The cost per time unit of any AGVs (Manufacturing lib) batched with the item.
7	Original Cost: Used in calculating cost when unbatching items using the Unbatch, Variable block (Manufacturing lib).
8	Unused
9	Unused

Integer array
The integer array itemArrayI[][5], passed through sysGlobal4, contains the following information:

Slot #	Description
0	Free row flag. This is used by the Executive block for memory management. It has a value of 1 if the row is free (that is, has no existing item associated with it), and a value of 0 if the row is in use. See the Exit block for an example of how to use this to delete an item.
1	Batch ID. This is used by the batching and unbatching blocks to keep track of which items are part of what batch.
2	User defined integer value. This value is left untouched by the blocks in the Discrete Event library.
3	Second batch ID, used during batching and unbatching. This second level of batch IDs is required to maintain infinite levels of batching depth.
4	Reserved for future use.

Timer array
The timer array itemArrayT[], passed through sysGlobal6, is for the Timer block. This array is only available for use if you have a Timer block in your model. See the Make Your Own block in the Make Your Own DE library to see how to test for the presence of this array. It contains the string of unique timer tags the timer uses to identify each item in the simulation. Each item is given a unique value defined as "inputNumber.blockNumber,".

String attribute array
The string array itemArrayA[] was previously used to store an item's attribute values. The current system uses global arrays, as discussed in the section following. The string attribute array is retained in some blocks for backwards compatibility and is not to be used when building new blocks.

Attribute global arrays
There are two global arrays that are responsible for attributes. The first stores attribute names; the second stores attribute values.

The attribute names are stored in a String15 type global array called "_AttributeList". This array is created whenever a block which uses attributes is placed in the model. Its single column contains an alphabetized list of the attribute names entered into the blocks. All blocks that reference attributes use a popup menu to allow the user to select from a list of attributes that have already been defined for the model or to create a new attribute. When the popup menu is clicked, these blocks reference the AttribList global array to ensure that all of the attributes defined in the model are available for selection in the popup menus. Each time a new attribute is added, this global array is increased in size by one, the attribute name is appended to the list, and the list is sorted.

While building a model, it is possible to define attributes that end up not being used or referenced in the model. Cleanup of unused attribute names is done at the start of the simulation. The Executive clears the AttribList global array in its stepsize message handler. Each block in the model that references an attribute name then adds all of its referenced attributes to the AttribList global array in their stepsize message handlers.

In the initsim message handler, the Executive sorts the new AttribList global array (which now includes only attributes which are used in the model). In doing this, each attribute name is assigned a sequential index (the global array index) in alphabetical order. Each block then calls the BuildAttribList procedure which searches the AttribList global array for the attribute that is reference by the calling block and sets its popup menu to the index value of the attribute (see the code of the Get Attribute block in the Discrete Event library for an example of this procedure). This index is used when referencing the attribute value during the simulation.

In addition, if either of the two costing attributes ("_cost" and "_rate"; see "Activity-based costing" in the main Extend User's Guide) are used in the model, they are assigned the index values at the end of the list of user defined attributes. The attribute with index 0 stores the animation object for the item.

The second global array used for attributes stores the attribute values for each item. In the Executive's initsim message handler, the two-dimensional global array "_AttribValues" is created to store the values of the attributes during the simulation. The number of columns is calculated as the number of user defined attributes plus one for the animation attribute plus two for the costing attributes (if costing is used in the model). The number of rows corresponds to the number of items in the model and is increased if additional items are allocated. The attributes' values can be referenced by using the attribute index as the column and the item index as the row. For example: you would set an attribute with the following ModL code:

```
AttribValueIndex = GaGetIndex("_AttribValues");
GaSetReal(Value1,attribValueIndex,itemIndex,AttribIndex);
```

where:

Value1 = the value of the attribute

AttribValueIndex = The index to the "_AttribValues" global array

ItemIndex = The index of the item

AttribIndex = The index of the attribute

Basic item messaging

The actual moving of items between blocks is done through a messaging communication structure using item connectors and connections. This messaging system allows modelers to place blocks in a more intuitive sequence.

Discrete event blocks send messages to each other during the course of a simulation run. These messages are used for communication regarding whether items are available, whether they have been taken, and whether a block is free to receive items.

Messages are sent using the functions *SendMsgToInputs(connectorname)* and *SendMsgToOutputs(connectorname*). Messages are received in message handlers that have the name of the connector that *received* the message. For example the "On ItemIn" message handler is called when a message is sent to the "ItemIn" connector.

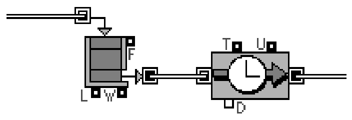
The discrete event blocks have a function in their code called SendMsg. This function is just included for clarity. It calls the two ModL functions mentioned above.

The *SendMsgToInputs* and *SendMsgToOutputs* functions only send messages. In discrete event models, so that more information can be sent with each message, the global sysGlobalInt3 is used as an argument to the messages, and the global sysGlobalInt0 is used as a return code value. Note that some globals are used to perform different functions during the initialization (CheckData and InitSim) phases of the simulation run.

During the Simulate phase of the run, the meanings of the various values of sysGlobalInt0 and sysGlobalInt3 are as follows:

	Sending (sysGlobalInt3)	**Returning (sysGlobalInt0)**
0 (rejects)	(Not sent)	Block rejects item
1 (wants)	Does block want item?	(Not returned)
2 (taken)	Item has been taken	(Not returned)
3 (needs)	Item needs to be taken	Block needs item
4 (query)	What is the index of the next item?	Item index for the next item (0 if no item is found)
5 (notify)	Item has been sent	(Not returned)
6 (blocked)	Is the downstream block blocking us?	(Not returned)
7 **(init)**	Used during initialization	(Not returned)

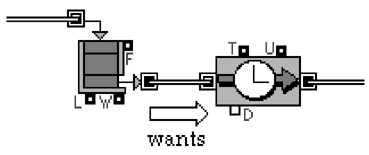
Depending on how the message sequence is initiated, items can be either pushed or pulled through the model. It is easiest to illustrate this with a series of simple examples. The figure below shows a Queue, FIFO block connected to an Activity, Delay block with a single item capacity.



Queue, FIFO connected to an Activity, Delay

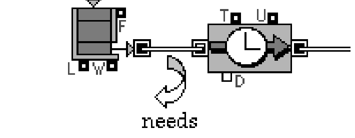
Push mechanism
When an item is pushed through the model, the upstream blocks (in this case, the queue) try to push their items out into any downstream blocks.

For example, assume that an item has just arrived at the queue and the queue is attempting to pass that item along into the activity. The first action is for the queue to send a *wants* message through its output connector to the activity. This is done by setting SysGlobalInt3 to a value of 1 (wants) and calling the function *SendMsgToInputs*. This indicates that the queue wants to send an item to the activity.



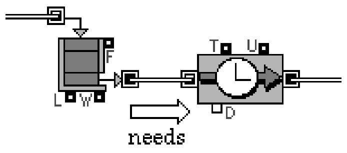
Wants message sent to activity

The Activity, Delay will receive the message in its “on itemIn” message handler. If the activity in the above example is not currently processing an item (idle), it will return a *needs* value to the queue by setting SysGlobalInt0 to 3 (needs), indicating that the item can be accepted.



Needs value returned from activity

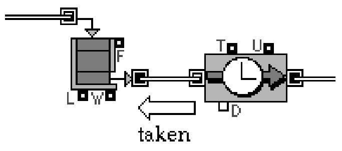
If a *needs* value has been returned from the activity, the queue then sends a *needs* message back to the activity by setting SysGlobalInt3 to 3 (needs) and calling the function *SendMsgToInputs*. At this point the item would be committed to moving from the queue to the activity.



Needs message sent to activity

However, if the activity is currently busy processing another item, it will return a *rejects* value by setting SysGlobalInt0 to 0 (rejects). If the item is rejected, the message sequence will be terminated.

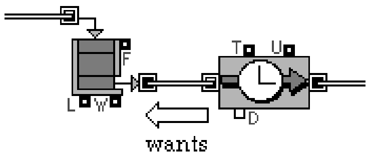
The item is logically moved from one block to the next by transferring its item index over the connection between the blocks. To do this, the queue sets its output connector value to the item index. When a block sets its connector value to the item index, the connector value of any connected blocks will automatically be set to that same item index value. Since the output connector of the queue is connected to the input connector of the activity, the two connectors will share the item index value. The activity then sets its input connector to a negative number and sends a *taken* message back to the queue to indicate that the item has successfully moved. This is done by setting sysGlobalInt3 to 2 (taken) and calling the function *SendMsgToOutputs*. In response to the *taken* message, the queue will update any internal statistics related to the departure of the item.



Taken message sent to queue

Pull mechanism

In addition to being pushed, as in the preceding example, items can also be pulled through the model. If they have remaining capacity, downstream blocks try to pull items into their inputs. For example, when the activity finishes processing the item, it will attempt to pull in another item. To do this, the activity first sends a *wants* message to the queue indicating that it is requesting an item. The *wants* message is sent by setting SysGlobalInt3 to 1 (wants) and calling the *SendMsg-ToOutputs* function.



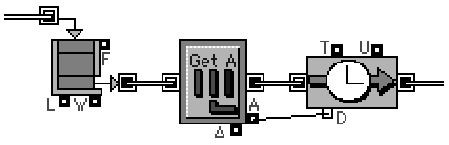
Wants message sent from activity

If an item is available in the queue, the queue's output connector will be assigned to that item's index value. The activity will pull in the item and then send a *taken* message back to the queue by setting sysGlobalInt3 to 2 (taken) and calling the *SendMsgToOutputs* function. If an item is not available in the queue, a *rejects* value (0) will be returned and the message chain will be terminated.

Passing blocks

Both of the blocks in the above examples are residence blocks and can hold items for some period of time. Passing blocks do not have this ability and must pass the item through in 0 time.

The following example shows a Get Attribute block reading an attribute value that will be used to set the delay of the activity. The Get Attribute block does not affect the messaging communication between the queue and the activity. Since it is a passing block, it transfers the initial *wants* and *needs* messages between the queue and activity. Thus, any number of passing blocks can be between any blocks that can hold items. Once the item moves into a passing block, it will send a *taken* message to the upstream block, as shown below.



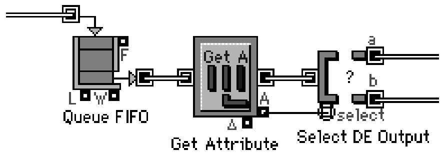
Passing block between queue and activity

Blocked and query messages

One of the more complex message communication subsystems in this architecture is the communication between blocks when a block needs information about an item that has not yet arrived. This occurs when an item needs to know if it can move downstream before it starts to move, or when it needs to determine which path it will take before it gets to the block that contains the different paths.

Traditional discrete event architectures would require dummy resources to overcome these problems. The Extend discrete event architecture, however, is able to determine the path of an item before it moves into the decision logic and is able to block through decision points without any additional modeling components.

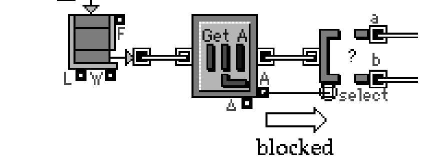
In the example below, the model section reads an attribute on an item (Get Attribute) and selects one of two paths for that item to follow (Select DE Output) based on the value of the attribute. Without the ability to send a Query message, the Get Attribute block would read the value of the attribute on the item, but only after the item has already entered the block. In this case, this could cause a problem, as the Select DE Output block may be blocked down the path that the item will need to travel. If the Select DE Output is blocked, and the item has to move into the Get Attribute block to present its attribute value, then the item will be stuck in the Get Attribute block. And since attribute-manipulating blocks are only meant to pass, not hold, items the Queue, FIFO will understate the number of items available.



Select controlled by a Get Attribute

The solution to this problem is to set up the Get Attribute block to be aware that it is in this situation. The Get Attribute block can then look upstream to see what the next item coming along will be, and not pull in the item until the downstream path is free.

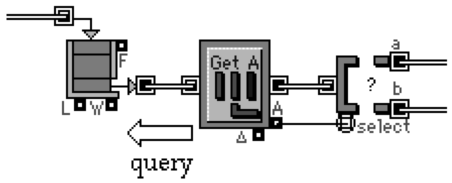
Blocked messages
The first part of this process involves sending a *blocked* message downstream from the Get Attribute to see if there are any blocks that could cause this situation. This is accomplished by setting SysGlobalInt3 to 6 (blocked) and calling the *SendMsgToInputs* function. The Get Attribute does this the first time it gets an incoming message (i.e. the first time the Select DE Output block requests an attribute value from the A connector.) The Get Attribute sends a blocked message from its item output connector in its on AttribOut message handler. (The on AttribOut message handler is called when the A connector on the Get Attribute block gets a message.) If the block downstream is a potential blocker, it will return a TRUE value by setting SysGlobalInt6 to 1 (TRUE). If a TRUE value is returned in response to the message, then the Get Attribute block sets a flag which records that it is blocked.



Blocked message sent to Select DE Output

Query messages

If it has been determined that blocking can occur, each time there is a request for an attribute value, the Get Attribute block sends a *query* message upstream by setting SysGlobalInt3 to 4 (query) and calling the *SendMsgToOutputs* function. This message is essentially a request for information about the next item that is available. The message will be propagated upstream by the blocks until it reaches a block that can contain items, such as a queue.



Query message sent to queue

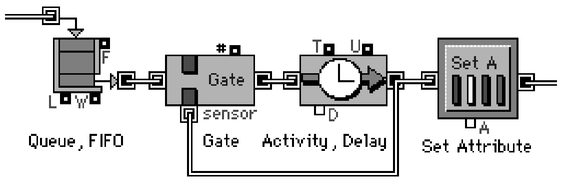
The block responding to the *query* message will check to see what the item index of the next item to be released will be and will return that value to the querying block by setting the global sysGlobalInt0 to the item's index value. The Get Attribute block will then access the global array _AttribValues to get the attribute value for that item. From this point on, each time that an attribute value is requested from the Get Attribute block, it will send out the *query* message and check the attribute value of the next item. It will not need to re-send the *blocked* message again.

Extend will notify users if there are any logical ambiguities that will not allow the model to operate properly. When this occurs, an error message is issued that recommends a course of action that will resolve the ambiguity.

The notify message

The final message used by this system is the *notify* message. This message is used to notify other blocks that an item has just passed by a specified point in the model. A special sensor connector receives this message. Sensor connectors do not pass items, they monitor the message stream, processing only the notify message. Only a few blocks have sensor connectors, although all of the blocks that process items will send the *notify* message through their item output connector by setting SysGlobalInt3 to 5 (notify) and calling the *SendMsgToInputs* function.

In following example, the Gate block limits the number of items in the section of the model between its output connector and the activity's output connector. It uses its sensor connector to determine when an item has passed the activity's output connector.



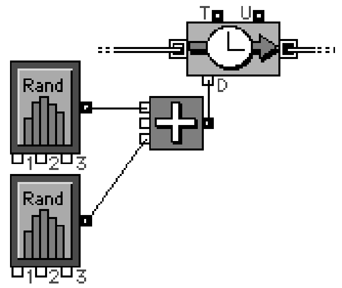
Situation that requires a Notify message

As an item travels from the activity to the Set Attribute, a *taken* message will be sent to the activity. In response to this message, the activity will send out the *notify* message. The normal item input connectors in the blocks will ignore the *notify* message, but sensor connectors will respond to it and start processing information about the item. In the above example, when a *notify* message is received by the sensor connector, the Gate block knows that another item has passed by and can allow an additional item into the model section.

Value connector messages

In addition to item connectors, value connectors are used to relay model information. Value connectors pass a single number from one block to another. Examples include a value output such as length of a queue or an input such as a delay time. The use of these connectors allows the combining of blocks which perform numerical calculations (continuous or generic blocks) to provide a control structure and logical information for the discrete event blocks. This provides additional modeling flexibility without requiring user programming or complicated interfaces.

In the example below, two Input Random Number blocks are added together to specify the delay for an activity. Whenever an item arrives to the activity, the Input Random Number and Add blocks will need to be recalculated. Whenever a discrete event block detects a condition where an update to the value of a connector is needed (in this case, an item arriving to the activity), it sends a message out its value connector (in this case, value input connector “D” in Figure 9) using the *SendMsgToOutputs*() function.



The sum of two random variables specifying an activity’s delay

Message emulation

A default feature for generic blocks called “message emulation” will propagate the message throughout all of the generic blocks used in the calculation. Message emulation is used whenever the generic block contains no message handlers for any of its connectors. In that case, the “on simulate” message handler is used as a default message handler for all connectors.

The code below illustrates an On Simulate message handler assigning the output connector (Con1Out) to the input connector (Con1In) plus one. Whenever either connector receives a message, a message will be sent out the other connector through message emulation. This will cause all connected blocks to execute their On Simulate message handlers, propagating messages where appropriate.

```
On Simulate
{
   Con1out = Con1In + 1;
}
```

Explicit connector messages

Overriding message emulation gives you, as a block programmer, more flexibility in the behavior of the block. If a generic block has one or more connector message handlers, message emulation is automatically disabled and the connector message handlers are used to perform the calculation instead. In this case, the generic block must send out messages to other value input and output connectors explicitly. For example, a message must be sent out the output connector whenever a message is received on the input connector, and a message must be sent out the input connector whenever a message is received on the output connector.

The figure below shows the code necessary to use message handlers to make a block behave equivalently to the message emulation used in the above example. Both examples will perform identically in model operation. Because message handlers have been explicitly specified for the connectors in the example below, message emulation has been automatically disabled.

```
On Con1In
{
   Con1Out = Con1In + 1;
   SendMsgToInputs(Con1Out);
}

On Con1Out
{
   SendMsgToOutputs(Con1In);
   Con1Out = Con1In + 1;
}

On Simulate
{
}
```

The “Messages” model (*Macintosh:* Messages in the ModL Tips folder; *Windows:* messages.mox in the ModLTips directory) uses animation to illustrate the messaging communication structure described in the sections “Basic item messaging” on page P201, “Blocked and query messages” on page P204, “The notify message” on page P206, and “Value connector messages” on page P207.

Functions in discrete event blocks

The following are not all the functions to be found in discrete event blocks. They are the ones found in most of the blocks and help give an understanding of how the code is organized.

Function	Description
SendItem	Attempts to pass an item out of the block. First it checks certain block variables to see if an item is available, then it will output the index value of the item and send a message to the receiving block (it calls SendMsg).
GetItem	Attempts to get an item once a block determines that it is ready to get an item. First it checks to see if an item is available, then it gets the index value, negates the connector, and sends a message to the sending block.
PassItem	Performs the actions of both the GetItem and SendItem functions. It is used in blocks that pass items through without delaying them (passing blocks).
GetArrays	Receives the global arrays maintained by the Executive block into local dynamic arrays. This is used in any blocks that need to modify the item information on items that pass through it. This function should be called immediately before any of the item arrays are referenced.
SendMsg	Sends the messages out through the connectors. It sends out messages based on the values of its arguments.

Globals in discrete event models

Several system globals are used in discrete event models. The following table lists the global variables with a brief description of their use during the Simulate message (most have undefined values during the CheckData message). For more information, examine the code of the Make Your Own Block in the Make Your Own DE library or the Executive block and other blocks in the Discrete Event Library.

Use of Global variables during Simulate message

Global	Definition during Simulate message
SysGlobal0	Global used to access the TimeArray of posted events
SysGlobal1	Report, used by the Report block in the Generic library
SysGlobal2	Debug, used by the Debugging Trace block in the Generic library
SysGlobal3	Global used to access the Real item array of discrete event item data
SysGlobal4	Global used to access the integer item array of discrete event item data
SysGlobal6	Global used to access the string item array of discrete event timer data
SysGlobal7	Used to access the TimeBlocks array of event posting blocks.
SysGlobal8	Used in communication between the Resource Pool; Queue, Resource Pool; and Release Resource Pool blocks
SysGlobal9	Global used to access the cost item array for discrete event item data
SysGlobal15	Used in V3.2 and earlier to pass the ItemArrayA attribute array. It is only used when the *String Attributes* option is selected in the Executive.
SysGlobalStr0	Name of attribute being checked upstream
SysGlobalInt0	Return code from the messages that discrete event blocks send to each other
SysGlobalInt1	Index value for the first free row in the item arrays maintained by the Executive block
SysGlobalInt2	Total number of rows of data that have been allocated to the item arrays. This will always be a small amount higher that the number of items in the model.
SysGlobalInt3	Argument to the messages that discrete event blocks send to each other
SysGlobalInt4	Tells whether the priority is being checked
SysGlobalInt5	Global batch count
SysGlobalInt6	Specifies whether or not there is a blocked block (see the Get Attribute block)
SysGlobalInt7	ID of the item currently being disposed
SysGlobalInt8	Used to pass the block number to the Executive when the block is rescheduling itself in the CurrentEvents array.
SysGlobalInt9	Flag when the plotter is sending a message. (See the code of the Plotter DE and the Input Random Number blocks for more information.)
SysGlobalInt10	Used in communication between the Resource Pool; Queue, Resource Pool; and Release Resource Pool blocks

Global	Definition during Simulate message
SysGlobalInt11	Used to control whether or not a new random value is generated in a connected Input Random Number block.
SysGlobalInt12	Passes item index in Throw and Catch blocks (Discrete Event library)
SysGlobalInt13	Set to TRUE if a new random number should be generated for a select block.
SysGlobalInt14	Used by Throw and Catch blocks (Discrete Event library) to determine which Throw block is sending the message
SysGlobalInt15	Used in communication between the Resource Pool; Queue, Resource Pool; and Release Resource Pool blocks
SysGlobalInt16	Set to TRUE during CheckData message if discrete event model is calculating item costs
SysGlobalInt17	Holds the number of attributes in a discrete event model
SysGlobalInt18	Holds the number of system attributes in a discrete event model

Use of Global variables during CheckData or InitSim messages
The table above describes how system globals are used during the Simulate message. Although most of these variables are only used during the Simulate message, some are also used in the CheckData and InitSim message handlers, and have a different meaning during those message handlers:

Global	Different definitions during CheckData/InitSim messages
SysGlobalInt0	Number of blocks posting events for the time array.
SysGlobalInt1	Block number of the Executive block.
SysGlobalInt8	Used by the Executive block during CheckData to check for duplicate Executive blocks
SysGlobalInt11	Used to control random seed initialization

Creating blocks for discrete event models

The Make Your Own block is useful for creating blocks that will be used in discrete event models that have item inputs and outputs. However, you may want to create blocks that have value inputs and outputs, but are meant to be used in discrete event models. In that case, do not use the Make Your Own block as a template. Instead, you only need to follow these two rules:

- Add SendMsgToInputs(connName) and SendMsgToOutputs(connName) functions to your Simulate message handler for all input and output connectors on your block. Remember that the argument to SendMsgToInputs is the name of the *output* connector on the block you are creating; likewise, the argument to SendMsgToOutputs is the name of the *input* connector.

- If you want to control how the block receives and sends messages, you need to add at least one message handler (that can be empty) with the name of one of your connectors. Otherwise, the block will emulate connector messages. Thus, if the name of one of the output connectors is "G1Out", you would add a message handler such as:

```
on G1Out
{
...
}
```

Note Using options in the Debugging command from the Run menu, you can cause your models to display block messages as they run. This will give you a feel for how messaging works in discrete event models.

Working with arrays

Since you will typically use arrays to store any data which is more complex than a single variable, you will probably use them fairly often. Extend provides lots of features and functions that use arrays, especially regarding discrete event modeling. Note that, while ModL does not directly support structures, it emulates them using arrays of arrays. You will find that working with arrays in Extend is simpler and safer than using the data structures that are available in C.

Memory usage of variables, arrays, and items

For most models, you do not need to worry about how much memory your variables take up. You may need this information in some circumstances, especially if you are using huge arrays. Note that there is no overhead for using arrays.

Type	Memory used
Real	10 bytes for Macintosh; 8 bytes for Windows and PowerMacintosh
Integer	4 bytes
String	256 bytes per string regardless of contents
Str15	16 bytes per string regardless of contents
Str31	32 bytes per string regardless of contents

The total of all static arrays and variables in a block, including any data tables in the block's dialog (which are real arrays, 10 bytes per element), cannot exceed 32,767 bytes. When a user-defined function or procedure is called, its local arrays can have up to 32,767 bytes. The size of dynamic arrays are not included in any of the above calculations and can have up to 2 billion elements each.

If your arrays are small, use fixed dimension static arrays since they are easier to declare. If your arrays are large, use dynamic arrays.

Pass by value and reference (pointers)

In C, you can pass variable arguments to functions or procedures by value or by reference (pointers). When you pass a variable by value, the value of that variable in the outside environment is not affected by anything that function or procedure does. When you pass by reference, however, the function or procedure can modify the contents of the variable and those modifications are seen by the outside environment.

In ModL, all non-array variables and single elements of arrays (such as myArray[i]) are always passed by value and are therefore never modified. Arrays, however, are always passed by reference (such as “myArray”, with no subscripts) and therefore can have their contents changed by a ModL or user-defined function. If you are writing user-defined functions, this feature makes it easy to return more than one value. Simply pass an array to your user-defined function and change the values in the array. All changes you make to the array in your function can be seen in the message handler when the function returns.

Passing arrays

Essentially, a passed array is a pointer assigned to a real variable with the PassArray function and it is read from that real variable and converted back to an array with the GetPassedArray function. (In ModL, pointers have more information than just the address of data, so real variables are used to hold them.) These functions are listed in “Passing arrays” on page P150. The “Passing Arrays” model (*Macintosh:* Passing Arrays in the ModL Tips folder; *Windows:* PassArry.mox in the ModLTips directory) illustrates how to pass, receive, and modify an array.

Passed arrays must be dynamic arrays but they can be any type (real, integer, or string). A passed array has the same properties as an array that you use in a single block.

Note *If you pass arrays, those arrays can only be resized or disposed in the same block as they were created.*

You can use the SendMsgToBlock function and the BlockReceive message handler to tell the originating block to resize the array. An example of this is the Executive block in the Discrete Event library.

It is important to note that any changes made to data in a passed array affects all blocks that reference that array, including the block that originated the array. If you want to make a change to a passed array that is not reflected in previous blocks in the model, copy the values from the passed array to a new array, make changes to that new array, and pass that new array. (You can copy an array quickly with a “for” loop, as described later in this chapter.)

Passing arrays through connectors

The blocks built in this manual pass single values through their connectors. You cannot pass arrays quite as easily as you can single values, but it is not very difficult to add the few functions that let you pass arrays through the connectors.

Because a connector is a real variable, you can pass arrays through connectors. To read a passed array from a connector, use the GetPassedArray(connector, array) function. For example, assume that you are receiving an array through the input connector called ArrayConnectorIn. Your message handler might look like:

```
real theArray[];
...
on Simulate
{
     if (GetPassedArray(ArrayConnectorIn, theArray))  // is it passed
          {
          . . .     // Yes, use the array
          }
          else
          {
          . . .     // The array did not arrive yet
                    //   or it is not a passed array
          }
     . . .
}
```

All of the values in the passed array can now be accessed and changed by using the variable `theArray` in your code.

The process for passing multidimensional arrays is exactly the same, with the exception that you need to confirm that the fixed dimensions are the same for both the passing and receiving blocks.

There are two different ways to pass an array to an output connector, depending on whether the array was passed to the block or it originated in the block. If the array was passed to the block, simply set the output connector to the same value as the input connector:

```
ArrayConnectorOut = ArrayConnectorIn;    // pass the old pointer on
```

To pass an array that originated in the block, use the PassArray(array) function. You can assign the value of this function to an output connector (or to a real variable in your ModL code that is then assigned to an output connector). Assume that you had changed the values of `theArray` in the example above and wanted to pass them out the connector named ArrayConnectorOut. You would use:

```
ArrayConnectorOut = PassArray(theArray);   // pass the new pointer
```

If you are just passing an array through a block without looking at it, you should not use GetPassedArray. Simply assign the output connector to the value of the input connector, and the real number that holds the passed array will be handled with no overhead:

```
ArrayConnectorOut = ArrayConnectorIn;    // pass the old pointer on
```

Passing arrays through global variables
You can use any real variable to hold passed arrays. This means that you can pass arrays through the global variables Global0 through Global19 which are available to all blocks. If you want a global array, pass it to one of the global variables and use GetPassedArray to interpret the global variable when you want to get at the array values. Discrete event blocks are an example of this.

Precautions when passing arrays
The *GetPassedArray* function needs to be called before accessing a passed array. If an array is created at the beginning of the simulation and the size is never changed and the array is not disposed of during the simulation, then GetPassedArray only needs to be called once, at the beginning of the simulation.

Note *If you pass arrays NOT during a run, call GetPassedArray() in any message handler that uses those arrays before accessing those arrays.*

However, GetPassedArray must be called again before accessing any passed array which may have changed in size or been disposed of. Trying to access a passed array after the creator block has disposed of or resized it can cause a crash if the GetPassedArray function is not called immediately before access is attempted. When you resize or dispose of an array, its memory location may change or otherwise become invalid. A crash can occur because the block that received the array has a pointer to a specific location in memory, and will try to access that memory point even if it no longer is a valid array location. Calling GetPassedArray immediately before accessing the array will relink to the correct memory address if the array has been resized and will return a FALSE value if the array has been disposed of.

Since GetPassedArray is extremely fast (because it only links the pointer of the array), the safest action is to call it and test its return value before every series of accesses to the array.

An example of what <u>not to do</u> is the following:

Block #1

```
integer x[];
on checkdata
{
Makearray(x, 10);       // create the array
global9 = passarray(x);
}

on endsim
{
DisposeArray(x);        // dispose of the array
}
```

Block #2 //this is not safe!

```
integer x[];
on initSim
{
GetPassedArray(global9, x);
}

on endSim              //this is not safe!!!!
{
for(i = 0; I<10; I++)  // This is dangerous! Is the array still available?
     x[i] = 0;
}
```

This could cause a crash if the code in Block #2 is executed later in the simulation than Block #1. The problem occurs because Block #2 tries to access the array it thinks is in the variable x, while the array referenced by x has actually already been disposed of by Block #1.

It would be safer to use the following approach in Block #2:

```
integer x[];
on initSim
{
GetPassedArray(global9, x);
}

on endSim
{
if (getPassedArray(global9, x)) // This is safer. Get and test the array first.
     {
     for(i = 0; I<10; I++)
          x[i] = 0;

     }
}
```

For the same reason, it is important to call the function GetArrays() (see "Functions in discrete event blocks" on page P209) in your custom discrete event block code each time before you access the ItemArrays.

Using passed arrays to make structures

ModL does not support structures directly. It does, however, allow you to emulate structures using arrays of arrays. This is safer than using pointers and structures in C.

Suppose you want to pass an array consisting of real, string, and integer values. Since arrays can be passed to any real variable, many arrays can be passed into a real array, and that array can be passed through a connector or global variable. The receiving block can get the passed structure array and then get the individual passed data arrays from that array.

The following shows an example of using passed arrays to make structures.

```
          // This block makes a structure and passes it.
          // Declare the arrays at the top of the code.
real       structureArray[];     // This is the structure
string     stringValues[];       // Holds the strings
real       realValues[];         // Holds the reals
integer    integerValues[];      // Holds the logical values


on InitSim
{
          // Give the arrays a size.
MakeArray(structureArray, 3);     // Holds reals, strings, long.
MakeArray(stringValues, 10);      // 10 elements for each array
MakeArray(integerValues, 10);
MakeArray(realValues, 10);

          // Pass the arrays to the real structureArray.
          // This needs to be done only once each simulation run.
structureArray[0] = PassArray(realValues);
structureArray[1] = PassArray(stringValues);
structureArray[2] = PassArray(integerValues);
}


on Simulate
{
          // Put data into the realValues, stringValues,
          //   and integerValues arrays.
          // For example, set one element of each array.
realValues[0] = 98.6;
stringValues[0] = "Octane rating";
integerValues[0] = TRUE;

          // The data arrays were already passed to the structureArray.
          // Now, pass the structureArray to the connector.
          // This has to be done here because connectors are active
          //   only during "On Simulate".
DataOut = PassArray(structureArray);
}
```

The following is the receiving block's code. This gets the passed structure array, then gets the individual arrays from the structure array for use in the block:

```
          // Declare the arrays at the top of the code.
real       structureArray[];     // This is the structure
string     stringValues[];       // Holds the strings
real       realValues[];         // Holds the reals
integer    integerValues[];      // Holds the logical values
```

```
on Simulate
{
     if (GetPassedArray(ConnectorIn, structureArray))
          {
                    // Get the reals
          GetPassedArray(structureArray[0], realValues);
                    // Get the strings
          GetPassedArray(structureArray[1], stringValues);
                    // Get the logicals
          GetPassedArray(structureArray[2], integerValues);
                    // Use the array values
          if (stringValues[0] == "Octane rating")
               . . .
          }
     else
          {
                    // The structureArray did not arrive yet
                    //   or is not an array.
               . . .
          }
}
```

Working with global arrays

As discussed in “Global arrays” on page P83, global arrays provide a repository for model-specific data and are accessed and managed through a suite of functions (see “Global arrays” on page P151). Global arrays can be referenced either by name or index value. The following is an example of how to create, access, and dispose of a global array:

```
          // Declare the arrays at the top of the code.
integer    arrayIndex;      // index value for the global array

on initsim
{
arrayIndex = GAGetIndex("myGlobalArray");   // see if global array already exists

if (arrayIndex < 0)   // if global array does not exist...
   {
                 // Create a three column global array of real numbers named
                 // “myGlobalArray”. Assign array’s index to arrayIndex.
   arrayIndex = GACreate(“myGlobalArray”, GAReal, 3);
                 // Resize array to contain 10 rows of data
   GAResize(“myGlobalArray”, 10);
   }
}

on simulate
{
real realNumber;
. . .
                // Set second row and column of global array to realNumber
                // (row and columns start at zero)
GASetReal(realNumber, arrayIndex, 1, 1);
. . .
```

```
      // Read third row and column of global array and assign to realNumber
realNumber = GAGetReal(arrayIndex, 2, 2);
}

On Endsim
{                                       // We are done with myGlobalArray.
if (GAGetIndex("myGlobalArray") != -1)  // Only if myGlobalArray still exists,
    GADispose("myGlobalArray");         // dispose of it.
}
```

Copying arrays using "for" loops

Copying the elements of an array to another array is quite easy with the "for" loop construct. The following copies a two dimensional array that has 100 elements:

```
// Copy array a into array b.

integer     a[10][10], b[10][10];
integer     i, j;

for (i = 0; i <10 ; i++)
     for (j = 0; j<10; j++)
          b[i][j] = a[i][j];
```

Using arrays to import unknown rows of numbers

The Import function reads numbers into a real array. If you do not know how many lines are in the file and you use a fixed-size array, you will lose lines if the array is too small or waste memory space if the array is too long. Instead, use a dynamic array to be sure that you will get all the rows without having to specify the dimension of the rows. The function returns the number of rows read, so you can then reduce the size of the array after the call. For example:

```
integer numRowsRead;
real fileArray[];
string theFileName, thePrompt, theDelim;
...
MakeArray(fileArray, 10000);
numRowsRead = Import(theFileName, thePrompt, theDelim, fileArray);
MakeArray(fileArray, numRowsRead);
```

The second call to MakeArray reduces the size of the array without disturbing the contents in the rows left. Of course, if you import into a two-dimensional array, you have to specify the size of the second dimension.

General programming

The following sections show you how to change and read any block parameter from within one block, use text on the model worksheet to globally change block parameters, pass messages between blocks, and much more.

Changing and reading parameters globally from a block

The GetDialogVariable and SetDialogVariable functions can be used to read and set dialog items by name for any block in the model from within one block. When you combine this with the ability to read block names and labels, this feature can be used to build a block that can globally control the model, gather statistics on a class of blocks, or change parameters within all blocks of a certain type. For example, you can build a block that can set specified Activity blocks (i.e. blocks whose labels include a specific wording, such as ABC) to double their entered delay value. Here is sample code that does this:

```
on doAllButton     // user clicked the button to change the blocks
{
integer    nBlocks, i;
real       value;
string     name, label, paramStr;

nBlocks = NumBlocks();
for (i=0; i<nBlocks; i++)      // all the blocks in the model
    {
    name = BlockName(i);       // get the name of the block
    label = GetBlockLabel(i);  // get the block's label

         // look for Activity at beginning (with no case sensitivity)
         // AND look for ABC anywhere in label
    if (StrFind(name, "Activity", FALSE, FALSE) == 0 &&
            StrFind(label, "ABC", FALSE, FALSE) >= 0)
        {
        // Found them. now read the parameter for the delay
        paramStr = GetDialogVariable(i, "waitDelta", 0, 0);

        if (paramStr != "")    // not empty means it was found
            {
            value = StrToReal(paramStr);
                               // now set it to double value
            SetDialogVariable(i, "waitDelta", value*2.0, 0, 0);
            }
        }
    }
}
```

Reading text blocks as commands

Text on the model worksheet can be accessed in a block's code. The BlockName function returns the name of a block for the specific block number. However, blocks aren't the only items on the worksheet that have numbers. Every piece of text that you add to a model gets its own number. Thus, you can use BlockName to read the text. This can be useful if you want to see how you have changed some text on the model.

You can also use this feature to globally change parameters in block dialogs. For example, assume you want to give a command to the model to change a specific parameter in many blocks. Normally, you would have to open all of the blocks and type the new parameter value. Here is a strategy that allows you to type some text, such as “Speed=55”, on the model window that will cause all of the speed parameters in all of the blocks to change when the simulation is run. Note that the following function can be put in any block and can be called for each typed value that the code needs:

```
real GetTypedValue(string name)     // User defined function
{
integer     nBlocks;
integer     i, position, nameLength;
string      typedText, valuePart;
real        valueFound;

nameLength = StrLen(name);          // Number of characters
nBlocks = NumBlocks();              // Number of blocks

for (i=0; i<nBlocks; i++)           // Loop thru all blocks
     {
     typedText = BlockName(i);      // Get block name or text

     // find the "name=" string
     position = StrFind(typedText, name+"=", FALSE, FALSE);

     if (position == 0)             // Must not be part of a larger word
          {
                    // Get numeric part of string (skip over "name=")
          valuePart = StrPart(typedText, position+nameLength+1, 255);
          valueFound = StrToReal(valuePart);     // Convert to real
          if (noValue(valueFound))
               {
               UserError("The value for "+name+" must be numeric");
               abort;               // Stop the simulation
               }
          else
               return(valueFound);  // Return the found value
          }
     }

          // No "name=" was found
UserError("Your typed variable, "+name+", was not found");
abort;     // Stop the simulation
}

          // Get your values. If they're not found or are bad, the
          //  GetTypedValue function stops the simulation with a message
on Checkdata
{
          // SpeedValue and MassValue are the names of dialog parameters
SpeedValue = GetTypedValue("Speed");
MassValue = GetTypedValue("Mass");
}
```

Passing messages between blocks

Extend allows blocks to “call” other blocks by sending them a message. You can use this feature to cause an action in another block, such as having it perform a calculation or open a plot. Global variables are used to pass arguments to the called block as well as to get results from the called block’s actions.

The following code calls another block that calculates some functions when a dialog button is clicked. Since any block can call this block and use its functions, you can think of this as a global function block. See the Executive block in the Discrete Event library (*Macintosh*: Discrete Event Lib; *Windows*: de.lix) for an example of a block that has global functions. See the Discrete Event library for examples of blocks sending messages using connectors.

Calling block

```
on Button  // called when the “Button” is clicked
{
. . .
Global0 = myArgumentValue;      // Set up the argument to the global function
GlobalInt1 = 1;                 // Number of the Hochmeister function

// GlobalInt0 contains block number of the "global function" block
// UserMsg0 is the message handler that calculates it
SendMsgToBlock(GlobalInt0, UserMsg0Msg);     // Call it
myResult = Global1;             // Get the result of the call
. . .
}
```

Global function block

```
          // Before the simulation runs, set up the block number so that
          //   other blocks can call it
on CheckData
{
GlobalInt0 = MyBlockNumber();     // Use a global integer variable
}

          // A message from another block (a "global function" call)
on BlockReceive0
{
          // GlobalInt1 contains the number of the function
switch(GlobalInt1)    // Which function did they want?
     {
     case 1:          // The Hochmeister function
          // Global0 has the argument, the result goes into Global1
          Global1 = cos(Global0)^2.0+sin(Global0)^2.0;
          break;

     case 2:          // The Lemski-Pemski factor
          // Global0 has the argument, the result goes into Global1
          Global1 = Sin(Global0)/Cos(Global0)-Tan(Global0);
          break;
     }
}
```

Changing data while the simulation is running

Sometimes you need to change values in a block's dialog while the simulation is running. In most cases, Extend handles this by allowing the you change the value, which is then used immediately in the block's code (usually in the Simulate handler). However, there has to be a special initializing procedure if the block needs to calculate intermediate values based on that new data. Extend provides a mechanism to tell the block that the block's data has been changed. Then the block can do a recalculation of intermediate data before the simulation resumes.

When any data is changed in a dialog during a simulation, Extend sends a ResumeSim message to the block before the simulation can resume. This is an optional message handler because most blocks do not need to recalculate intermediate data, they just use the dialog values directly. But if the block has a ResumeSim message handler, it can take some action before the simulation resumes.

Here is an example of a block that needs to do a lengthy calculation before the simulation begins after a change in parameters. It recalculates the coefficient when the user changes the dialog parameter, but doesn't zero out the accumulated value so the simulation can continue properly:

```
real     coeffValue, accum;

real     CalcCoeffValue(real theValue)     // This function does the
                                           //    coeffValue calculation
{
real     coeff;
integer  i;

coeff = 0.0;                         // Initialize the sum
for (i=0; i<100; i++)                // Loop a hundred times
     coeff = coeff+cos(theValue*i);  // Use theValue and calculate coeff
return(coeff);                       // Done, return it
}

on InitSim       // Initialize all values for beginning of simulation
{
coeffValue = CalcCoeffValue(dialogParameter);  // Calc coeff and
accum = 0.0;                                   //   init to 0
}

                 // User changed dialogParameter during simulation.
on ResumeSim     // Just calculate new coeffValue,
                 //   don't initialize accumulated value.
{
coeffValue = CalcCoeffValue(dialogParameter);   // Just calc coeff,
                                                //   don't zero accum
}

on Simulate          // Calculate new values during the simulation
{
accum = accum+coeffValue*conIn;       // Fast. Multiply input by coeff
conOut = accum;                       // Output accumulated value
}
```

Scripting

In Extend, the process of building models typically relies heavily upon user interaction. The standard process of placing blocks on the worksheet, connecting them together, and filling in the appropriate dialogs (while graphical and intuitive), requires direct user participation. However, models can also be built, modified, and controlled indirectly using Extend's *scripting* features.

Scripting is an extremely powerful feature which allows you to:

- Build, run, and control models from within another application
- Create custom wizards to simplify tasks or interact with users
- Develop self-modifying models

By using the scripting functions (see "Scripting" on page P157), you can tell Extend what blocks to place on the worksheet and where, how to connect the blocks together, and what values to use for the block's dialog parameters. In addition, any menu command can be executed by a function call, for instance to run a model. When used in conjunction with the IPC functions (see "Interprocess Communication (IPC)" on page P102 or "OLE (Windows only)" on page P105), the scripting functions can be used to build entire models based on information from another application.

The scripting functions can also provide a means for developing "wizards" – blocks that help the user to perform a task by gathering information, then building or modifying a model based upon that information.

You can also develop blocks which help models achieve desired results. You can build artificial intelligence into your models by building blocks that query the model for specific metrics and simultaneously modify the model based on those metrics. For instance, you would use this self-modifying feature to automate the process of changing models in response to simulation results or to build goal-seeking models.

See the Scripting block in the ModL Tips library (*Macintosh*: ModL Tips Lib; *Windows*: modl.lix) for an example of using the scripting functions.

BLANK and NoValue

The constant BLANK is a special value that represents "no value". Technically, it is not a number and appears in a dialog as a blank item. To make a variable a NoValue, assign BLANK to it (a = BLANK).

If a number and a NoValue are added, multiplied, divided or subtracted, the answer is always a NoValue. Thus, if any of the values in any equation is a NoValue, the result will always be a NoValue.

If a real value is divided by 0, the answer is a NoValue; the square root of a negative number is also a NoValue. In fact, any operation on numbers that causes an undefined result or an infinity produces a NoValue answer which can be tested with the NoValue function described in "NoValue(real x)" on page P85.

Note To test for a NoValue, do not compare it to BLANK in an "if" statement. Instead, always use the NoValue function:

```
if (myValue == BLANK)       // THIS WILL NOT WORK!

if (NoValue(myValue))       // this will always work
```

NoValues can cause unexpected problems when converting reals to integers. It is a good idea to screen for NoValues before converting reals to integers, as discussed below.

Numeric conversion

When a value of one type is assigned to a variable of another type, conversion is automatic and conforms to the following rules.

Real to integer

Like all programming languages, converting from real numbers to integers in ModL is not always exact. If a real number is also an integer, it will be exactly converted. Otherwise, the conversion will cause the fractional value (mantissa) to be truncated. Thus, 0.001*1000.0 may not equal 1 (the integer value), but may equal 0.99999... When this number is converted to an integer, the answer is 0, not 1.

NoValue to integer

As described above, setting a real value into an integer variable has a consistent result, namely the truncation of the real value. This is true in all cases except where the real variable contains a BLANK, or NoValue value. In this case the NoValue is too large to fit into an integer, and the integer variable will then contain a meaningless value which could cause problems in calculations.

The best method for dealing with this potential problem is to screen the real values before assigning them to the integer variable. The following code is an example of how to do this.

```
Real       realV;
integer    intV;

if (NoValue(realV))
     intV = 0;        // Can’t convert NoValue to integer. Meaningless result.
else
     intV = realV; // Can convert real to integer.
```

Integer to real

All connectors are real. If you use them in an expression which uses integer constants or variables, ModL converts the integers to real during the calculation, slowing execution. To speed the execution of your models, use real variables with other real entities where possible.

OLE and ActiveX

A set of functions is available in ModL to access embedded objects. These are described in detail in “OLE (Windows only)” on page P105. They include functions to insert objects; (OLEInsertObject) Functions to invoke methods, and set properties of objects; (OLEInvoke, OLEPropertyPut, OLEPropertyGet) and many other miscellaneous functions that allow extensive control of an embedded object.

Controlling OLE Embedded Objects from ModL script

A set of functions is available in ModL to access embedded objects. These are described in detail on “OLE (Windows only)” on page P105. They include functions to insert objects; (OLEInsertObject) Functions to invoke methods, and set properties of objects; (OLEInvoke, OLEPropertyPut, OLEPropertyGet) and many other miscellaneous functions that allow extensive control of an embedded object.

Extend as an OLE Automation Server

Extend also supports a simplified version of OLE Automation as a server. This is the ability for another application to control the Extend application from outside via OLE. The automation supported in Extend is three methods on a simplified single object model. The folder "OLE Test" in the ModL tips folder contains an example of C++ and Visual Basic code that exercises all three of the methods supported.

For use with C++ or other related languages, the Object is the Extend application, referenced by the following GUID:

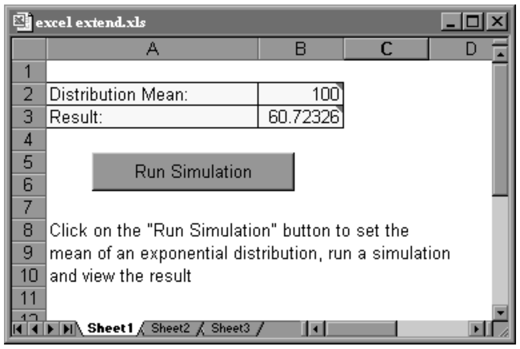
```
{E167B362-7044-11d2-99DE-00C0230406DF}
```

The three methods available are Poke, Request, and Execute.

NOTE: Because ModL scripts can be executed with the Execute method, complete control of Extend is available through these methods.

Visual Basic example

The Visual Basic code example is set up as a macro in a spreadsheet called *ExcelExtend.xls*:


excel extend.xls

	A	B
2	Distribution Mean:	100
3	Result:	60.72326

Run Simulation

Click on the "Run Simulation" button to set the
mean of an exponential distribution, run a simulation
and view the result

Excel spreadsheet using macro associated with button

When the button is clicked, the following Visual Basic code is executed, launching Extend, loading the model, running the model, and getting the results from a block in the model:

```
Sub RunSim()
   ' The Extend application is an object

    Dim ExtendApp As Object
    Dim GettingObject
    Dim ExtendPath As String

    'Create an instance of Extend

    On Error GoTo ErrorHandler:      ' If Extend is not loaded an error will
                     be returned and the CreateObject method will be called
    GettingObject = 1
    ' Get the active Extend object. If Extend is not open, this will cause an error
    '   and will go to the ErrorHandler label. Here, the CreateObject function
                  is called

    Set ExtendApp = GetObject(, "Extend.Application")

loadmodel:

    GettingObject = 0

    ' Find out where Extend is installed

         ExtendApp.Execute "GlobalStr0 = GetAppPath();"     ' get the path to extend

         ExtendPath = ExtendApp.Request("System", "GlobalStr0+:0:0:0")

         ' open the model

         ExtendApp.Execute "OpenExtendFile(" + """" + ExtendPath +
           "Examples\ModL Tips\Ole Automation\vb\VB Example.mox" + """" + ");"

    ' Poke the value into the dialog of the block

    ExtendApp.Poke "System", "MeanDist:#0:0:0",
                  Str(Worksheets("Sheet1").Range("b2").Value)

    ExtendApp.Execute "RunSimulation(FALSE);"
    'Request the dialog variable from Extend

    Worksheets("Sheet1").Range("b3").Value = ExtendApp.Request("System",
                  "Meanval:#1:0:0")
    ' Destroy the Extend object

    Set ExtendApp = Nothing

    GoTo done:


ErrorHandler:

    If GettingObject Then

       ' Extend is not running (otherwise GetObject would have worked) so we
       '   call create object to start Extend

       Set ExtendApp = CreateObject("Extend.Application")
       GoTo loadmodel:
    Else
        MsgBox Error$
        Set ExtendApp = Nothing
    End If

done:

End Sub
```

Retrieving the IDispatch interface

The following C++ sample code would be one possible way you could access an IDispatch interface on an Extend application. See the Visual Basic section later in this chapter for a description of how to do this in Visual Basic.

(The GetActiveObject code attempts to find a running copy of Extend in the ROT (running object table.) The CoCreateInstance code creates a new instance of Extend if the running one is not found.)

```
CLSIDFromString ("{E167B362-7044-11d2-99DE-00C0230406DF}", &clsid);
hr = GetActiveObject(clsid, NULL, (IUnknown **) &m_pUnknown);

if (hr == S_OK)
    {
    // JSL - found a existing instance
    hr = m_pUnknown->QueryInterface(IID_IDispatch, &m_pDisp);
    hr = m_pUnknown->Release();
    }
else
    {
    theErr = GetLastError();
    MessageBox (NULL, TEXT("GetActiveObject Failed, creating a new Object"),
                TEXT("OleTest"), MB_OK) ;

    hr = CoCreateInstance (
            clsid,      // class ID of object
            NULL,       // controlling IUnkown
            CLSCTX_LOCAL_SERVER,// context
            IID_IDispatch,// interface wanted
            (LPVOID *) &m_pDisp) ;// output variable


    if (hr != NOERROR)
        {
        theErr = GetLastError();
        MessageBox (NULL, TEXT("Create Instance Not Successful"),
                TEXT("OleTest"), MB_OK) ;
        FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL,
        theErr, MAKELANGID(LANG_NEUTRAL,
                SUBLANG_DEFAULT), buf, sizeof(buf), NULL);
         MessageBox (NULL, buf, "Error", MB_OK);
        return;
        }
    }
```

Execute

The Execute method takes just a single argument that is the code to be executed. The dispID of Execute is 1.

The Execute method is the most flexible method call, as the command that is passed to Extend via this method is just a section of ModL script, and can contain anything that can be put into ModL script, including scripting functions. This means that by using the execute method; you can build Models, run models, or any of a number of things.

The C++ code you would use to call the execute method would look something like this (This code will cause Extend to display a userError statement with the value in userGlobal0):

```
// Arguments are all passed as variants

bStr       = SysAllocString((WCHAR *) L"userError(userglobal0);");

VariantInit(&vString);
vString.vt     = VT_BSTR;
vString.bstrVal    = bStr;

//Set the DISPPARAMS structure that holds the variant.

dp3.rgvarg     = &vString;
dp3.cArgs     = 1;
dp3.rgdispidNamedArgs = NULL;
dp3.cNamedArgs     = 0;

//Call IDispatch::Invoke()

hr = m_pDisp->Invoke(executeID, IID_NULL, LOCALE_SYSTEM_DEFAULT,
        DISPATCH_METHOD, &dp3, NULL, &ei, &uiErr);
```

Request
The Request method uses Topic and Item in the same way as the Poke method, and returns a string value. The dispID of Request is 2.

The C++ code you would use to call the request method would look something like this (This code will return the value present in userGlobal0):

```
// Arguments are all passed as variants

requestVariant = malloc(sizeof(VARIANTARG) *2);

itemStr = SysAllocString((WCHAR *) L"userglobal0:#0:0:0:0:0");
VariantInit(&requestVariant[0]);
requestVariant[0].vt = VT_BSTR;
requestVariant[0].bstrVal = itemStr;

topicStr = SysAllocString((WCHAR *) L"system");
VariantInit(&requestVariant[1]);
requestVariant[1].vt = VT_BSTR;
requestVariant[1].bstrVal = topicStr;

//Set the DISPPARAMS structure that holds the variant.

dp2.rgvarg = requestVariant;
dp2.cArgs = 2;
dp2.rgdispidNamedArgs = NULL;
dp2.cNamedArgs = 0;

var.vt = VT_EMPTY;

//Call IDispatch::Invoke()

hr = m_pDisp->Invoke( requestID, IID_NULL, LOCALE_SYSTEM_DEFAULT,
                  DISPATCH_METHOD, &dp2, &var, &ei, &uiErr);
```

```
SysFreeString(topicStr);
SysFreeString(itemStr);
```

Poke

The Poke method takes the three arguments Value, Topic, and Item, and sets the value specified in the value argument into the item specified in the item argument. The dispID of poke is 3. Topic is the name of the worksheet or model, or "system" if you don't need to specify a model. Value, Topic, and Item are all strings. The Item is specified as follows:

```
"VariableName:#BlockNumber:RowStart:ColStart:RowEnd:ColEnd"
```

RowStart, ColStart, RowEnd, and ColEnd can all be set to zero if the item specified is not a data table object, or an array.

The C++ code you would use to call the poke method would look something like this (This code will set the value of userGlobal0 to 34.5):

```
pokeVariant = malloc(sizeof(VARIANTARG) *3);

valueStr = SysAllocString((WCHAR *) L"34.5");
VariantInit(&pokeVariant[0]);
pokeVariant[0].vt = VT_BSTR;
pokeVariant[0].bstrVal = valueStr;

itemStr = SysAllocString((WCHAR *) L"userglobal0:#0:0:0:0:0");
VariantInit(&pokeVariant[1]);
pokeVariant[1].vt = VT_BSTR;
pokeVariant[1].bstrVal = itemStr;

topicStr = SysAllocString((WCHAR *) L"system");
VariantInit(&pokeVariant[2]);
pokeVariant[2].vt = VT_BSTR;
pokeVariant[2].bstrVal = topicStr;

//Set the DISPPARAMS structure that holds the variant.

dp.rgvarg  = pokeVariant;
dp.cArgs   = 3;
dp.rgdispidNamedArgs = NULL;
dp.cNamedArgs = 0;

//Call IDispatch::Invoke()

hr = m_pDisp->Invoke( pokeID, IID_NULL, LOCALE_SYSTEM_DEFAULT,
                  DISPATCH_METHOD, &dp, NULL, &ei, &uiErr);

SysFreeString(topicStr);
SysFreeString(itemStr);
SysFreeString(valueStr);
```

Using OLE Automation to control Extend from Microsoft Visual Basic

Extend can be controlled from Microsoft Visual Basic as well as C. The basic procedure is the same, but because of syntax and structural differences between the languages there are some significant differences.

In Visual Basic the first action is to create the instance of the Extend object. This is done by declaring a variable as an Extend object and either using the CreateObject function (if Extend is not running) or the GetObject function (if Extend is currently running):

```
'The Extend application is an object
    Dim ExtendApp As Object
'Create an instance of Extend
    Set ExtendApp = CreateObject("Extend.Application")
```

One technique that can be used if you are not sure whether Extend will be running or not is to call GetObject and have an error handling routine which calls CreateObject (assuming GetObject failed because Extend was not loaded).

Once the Extend object instance has been created, Extend's three OLE methods, poke which sets a variable in Extend, request which gets a varaible from Extend, and execute which runs a ModL script methods can be used to control the Extend application.

```
' Poke the value of 2 into the MeanDist" Dialog variable of block number 0
    ExtendApp.Poke "System", "MeanDist:#0:0:0", "2"
' Execute the RunSimulation ModL command
    ExtendApp.Execute "RunSimulation(FALSE);"
'Request the MeanVal dialog variable from block number 1and set it to the text
property on the GetValue TextBox
    GetValue.Text = ExtendApp.Request("System", "MeanVal:#1:0:0")
```

The Extend OLE Project (ModLTips\OLE\VB\Extend OLE.vbp) illustrates the use of each of these three methods to set a dialog variable, run the simulation, and get a dialog variable from Extend.

At the end of the Visual Basic session, the Extend object should be disposed:

```
    ExtendApp = Nothing
```

Debugging

Extend and the ModL language provide many methods for debugging models and block code. If you are interested in debugging the ModL source code in blocks, please see "Chapter 5: Using the ModL Source Code Debugger" on page P245

Slowing the model

Turn on animation and click the "Slow" button when running the model. This will slow the model and make it easy to see problems during the run.

Putting breakpoints in ModL code

The current Extend ModL source code debugger has a lot of advantages in block debugging (see "Chapter 5: Using the ModL Source Code Debugger" on page P245), offering conditional breakpoints, stack crawl, and viewing variable values in specified blocks. However, you can still use the old method of using the DebugMsg function to insert a breakpoint and monitor variable values as the simulation runs. You specify a message and variable values as this function's string argument. When the function gets called, it displays the argument in an alert. The advantage of using DebugMsg or DebugWrite rather than UserError is that Extend warns you if those functions are present when you load the library. See also the functions in "Debugging" on page P167.

Viewing intermediate results

When developing a block's code, there is an easy way to view intermediate results of your calculations without any interruption of the model. Just add an assignment statement:

```
comments = myVar;   // myVar will be visible as it changes
```

or add some more information and variables:

```
comments = "myVar = " +myVar+ ", myVar2 = " +myVar2;  // more info
```

in your code after some calculations. Now open the block's dialog, tab to the *Comments* field, and run your model. Since "comments" is the comments box (editable text) item in the dialog, any number assigned to it will be immediately visible in the dialog. This is different than using the DebugMsg() function to display data, as it doesn't interrupt the model for each new number displayed.

Stepping through the model run

If Step Each Block, Step Next Animation, or Step Entire Worksheet (Debugging command in the Run menu) is selected, you can pause (using the Pause command or button) the simulation and step through it (using the Step command or button). This would be mostly useful if a simulation runs too fast to see animation or the display of data.

Showing block messages

You can choose the Show Block Messages command (Debugging hierarchical menu) with the Step Each Block command, showing block messages in the order they are executed in the block's code. This makes it easy to trace infinite message loops or situations where no messages are being sent. If Only Simulate Messages is also active, the block messages shown will be limited to messages that occur during the Simulate phase of the simulation run. Running the simulation with Show Block Messages active automatically pauses the simulation at the beginning, so you then can manually step through it (using the Step command or button). Alternatively, you can choose the Pause At Beginning option (Debugging hierarchical menu) to pause the simulation at the beginning of a run.

Model profiling

Profiling generates a text file showing the percentage of time blocks execute during a simulation. You can use model profiling to analyze the amount of time that each block in your model is used when you run a simulation. This is generally used by programmers who want to determine if they should optimize custom blocks, although non-programmers might also use it to find the areas of their models that are most heavily used. To generate a profile text file, choose the Profile Block Code command under the hierarchical Debugging command from the Run menu, then run the model. Be sure to run the model long enough (at least 5 seconds for each block in the model) to compensate for extraneous events and get a good sample. It is also important to not move blocks in the model between runs if you repeatedly run the profile.

For example, the profile of the Bank Line model looks like:

```
Block Name              Block Number  Time (seconds)   Percent

Generator                    0             2.42           9.3
Queue, FIFO                  2             7.65          29.5
Activity, Delay              3             2.27           8.8
Activity, Delay              5             1.15           4.4
Activity, Delay              6             1.82           7.0
Exit (4)                     7             4.15          16.0
Plotter, Discrete Event      8             5.83          22.5
Executive                   10             0.62           2.4
```

You can use the information in this profile to tune your model or to look for anomalous results. For example, if one of the three Activity Delay blocks used a much higher percentage of the time than the other two, but you had expected them to be about the same, you could use that information to see what it was about that block that was different.

Note that only blocks that use 1% or more of the simulation time are shown in the profile. The percentages are approximate, so the sum of the percentages might not equal 100%.

Profile text files are opened, closed, and edited just like any other text file in Extend.

Adding Trace and Report code

Extend's Trace and Report features can be useful in debugging your model. If you create your own blocks, those blocks require additional ModL code to take advantage of these features. All of the blocks in the Generic and Discrete Event libraries already contain all the necessary code to work with Tracing and Reporting.

When building a new block, you should include code similar to the following.

Trace code

For continuous blocks (such as those in the Generic library), put the following code in the Simulate message handler. For discrete event blocks (such as those in the Discrete Event library), put the following code in the departure procedure:

```
// sysGlobal2 is the file reference number for the Debug Trace
// template for trace:    BLOCK NAME   BLOCK NUMBER   CURRENTTIME
if( sysGlobal2 != 0.0 ) // check for open file for TRACE
    {
    fileWrite(sysGlobal2,"myBlockName   block number "+(myBlockNumber())+
                         ".  CurrentTime:"+currentTime+".","",True);
    if(getBlockLabel(myBlockNumber()) != "")
        fileWrite(sysGlobal1,"Block Label: "+
                           getBlockLabel(myBlockNumber()),"",True);
....
....
    }
```

Report code
As noted in “Model reporting” in the main Extend User’s Guide, there are two type of reports - dialogs and statistics. During the On BlockReport message, the *GetBlockReportType* can be used to determine the type of report chosen by the user. For both continuous and discrete event blocks, put the following code in the On BlockReport message handler:

```
// sysGlobal1 is the file reference number for the TEXT REPORT
// template for report:    BLOCK NAME   BLOCK NUMBER
if( sysGlobal1 != 0.0 ) // check for open file for REPORT
    {
    if (GetBlockReportType() ==0) // 0 is dialogs, 1 is statistics
        {
         fileWrite(sysGlobal1,"myBlockName   block number "+
                                        (myBlockNumber()),"",True);
         if(getBlockLabel(myBlockNumber()) != "")
               fileWrite(sysGlobal1,"Block Label: "+
                                getBlockLabel(myBlockNumber()),"",True);
....
....
         }

    if (GetBlockReportType() == 1) // statistics report
         {
....
....
         }
    }
```

The blocks in the Generic and Discrete Event libraries have report and trace code in them. In addition, their code is set up to cause the Report and Trace files to be formatted in a structured manner.

Miscellaneous

Dates and times

Extend stores dates and times in a single number that is the number of seconds since midnight, January 1, 1904. The Now function returns the current date and time. Thus, it is easy to determine the difference between two times that are returned from Extend. For instance, the following code will report the amount of time that has elapsed since the beginning of the simulation:

```
...
real TimeStart, TimeNow, ElapsedTime;
...
on InitSim
{
     TimeStart = Now();
}
on Simulate
{
...
     TimeNow = Now();
     ElapsedTime = TimeNow - TimeStart;
...
}
```

To see a text representation of a date/time number, use the DateToString and TimeToString functions. For instance:

```
UserError("It is now "+TimeToString(Now())+".");
```

The CalcDate function creates a single numeric date value from its arguments (year, month, etc.) For example:

```
...
integer MyTime;

MyTime = CalcDate(2003, 2, 1, 14, 0, 0); // Feb. 1, 2003 at 2PM
```

Extensions

When you installed Extend, you also installed an *Extensions* folder in the same folder or directory as the Extend program. This folder or directory contains various Extend *extensions* and *include files*.

Note Do not store anything except valid extensions or include files in the Extensions folder or directory. If you store other files there, it will cause Extend to operate more slowly and could cause functional problems.

Extensions on the Macintosh

On the Macintosh, most extensions are resource files. Resources are either segments of code or pictures which are automatically opened and become accessible when Extend starts up. You can also add your own resource files to the Extensions folder. For Macintosh systems, Extend supports the following types of resources: XCMDs and XFCNs, sounds, drivers, picture resources, and QuickTime movies. These are discussed individually later in this chapter.

You access resource files using the ModL functions listed in Chapter 3: The ModL Language. Except for QuickTime movies, the resources must be copied into stand alone files and the files copied into Extend's Extensions folder. Except for QuickTime movies, the name referenced by the ModL functions should be the resource name, which is not necessarily the same as the file name. For simplicity, you should name the resource and the file the same name and keep each resource as a separate file (for QuickTime movies the name used to reference it is the file name).

Extensions on Windows
On Windows, extensions are files of various types: resource files converted from the Macintosh using Extend's MacWin converter utility, WAV, WMF or BMP files, or DLLs. In addition to the files shipped with Extend, you can also add your own extension files to the *Extensions* directory. For Windows, Extend supports the following types of files: DLLs, WAV files, bitmap files, WMF files, and QuickTime movies. These are discussed individually later in this chapter.

On Windows, except for DLLs, the name referenced by the ModL functions will be the file name. For DLLs, as described below, the name referenced will be the function name of the particular function you are trying to call from the DLL.

XCMDs and XFCNs

As discussed in "Accessing code from other languages" on page P57, Extend provides sets of functions that allow you to call code segments written in a language other than ModL from within a block's code. This is handy if you want to access existing functions written in another language or solve problems that are difficult in ModL. On Macintosh, these functions are identified as XCMDs (external commands) and XFCNs (external functions).

Overview
XCMDs and XFCNs are segments of code written and compiled in any language. Their standardized interface provides a method for linking between Extend and languages other than ModL. In order to access these code segments, they must be stored as a resource file in Extend's Extensions folder, discussed above.

Extend's XCMD and XFCN functions allow you to call these code segment resources from within a block's code and perform operations. For example, you can use an XCMD to calculate some function, perform a task, or even access the Macintosh toolbox. XCMDs and XFCNs can also be used instead of drivers to access external devices, or to solve problems that might be difficult or impossible to solve in ModL.

Turning code into an XCMD
Taking an existing code segment and turning it into an XCMD involves the following:

⇨ You must write or edit the code segment using a Macintosh compiler which must be capable of compiling XCMD or XFCN code segments. The code segment may be compiled on a 68K Macintosh or a Power PC. When calling the code segment, you must specify the platform it has been compiled on and whether it is an XCMD or an XFCN (see "XCMDs (Macintosh only)" on page P117).

- ➪ You need to slightly modify the code by adding the XCMD calling interface. The ModL Tips folder that ships with Extend gives an example of this interface using C.
- ➪ After compiling the code, you will have a file with the XCMD/XFCN in its resource fork. Place this file in Extend's Extensions folder and restart Extend. This will allow you to use Extend's XCMD functions in your block code to call the external code. **Note:** You can also call the DLLLoadLibrary() function to load a DLL that is not in the Extensions folder.
- ➪ To call an XCMD or an XFCN you need three functions: XCMD, XCMDParam, and XCMDArray. You call the latter two functions to build the XCMD's parameter list. Each call to XCMDParam or XCMDArray adds another parameter to the parameter list for the XCMD function. Thus, you need to call those functions before the XCMD function.

Note that the three XCMD functions are the exact equivalent of the ModL DLLXCMD, DLLParam, DLLArray functions for the Windows system. If you use either the DLLXCMD, DLLParam, DLLArray or XCMD, XCMDPARAM, and XCMDARRAY functions in your blocks, the block code will not have to be changed when you transfer libraries between Macintosh and Windows systems. However, the extension will have to be rebuilt.

See the XCMD functions on page P117 for more details.

DLLs

As discussed in "Accessing code from other languages" on page P57, Extend provides sets of functions that allow you to call code segments written in a language other than ModL from within a block's code. This is handy if you want to access existing functions written in another language or solve problems that are difficult in ModL. On Windows, these functions are identified as DLLs or dynamic-link libraries.

Overview

DLLs are libraries of code written and compiled in any language. Their standardized interface provides a method for linking between Extend and languages other than ModL. In order to access these code libraries, they must be stored in Extend's Extensions directory, discussed above.

The Extend DLL functions allow you to call DLL code libraries from within a block's code and use that code to perform operations. For example, you can use a DLL to calculate some function, perform a task, or even access Windows API calls. DLLs can also be used to access external devices, or to solve problems that might be difficult or impossible to solve in ModL.

DLL interfaces

There are two interfaces for calling DLLs from within Extend:

- The first set of DLL functions allows the user to access existing Windows DLLs. Because they have variable argument lists, these functions allow you to call almost any existing DLL. This interface includes the following functions: DLLLongCFunction, DLLDoubleCFunction, DLL-

BoolCFunction, DLLVoidCFunction, DLLLongPascalFunction, DLLDoublePascalFunction, DLLBoolPascalFunction, DLLVoidPascalFunction, DLLMakeProcInstance.

- The second set of DLL functions is designed for cross platform compatibility with the Macintosh version of Extend. It is only useful for calling DLLs written specifically for Extend. Since it uses a similar calling format and argument list as the XCMD functions for the Macintosh version, it requires a minimal amount of change (changing any platform specific calls and rebuilding under the new operating system) for cross platform compatibility. This interface includes the following functions: DLLXCMD, DLLParam, DLLArray.

  Note that these three functions are the exact equivalent of ModL's XCMD, XCMDPARAM, and XCMDARRAY functions for the Macintosh system. If you use either the DLLXCMD, DLLParam, DLLArray or XCMD, XCMDPARAM, and XCMDARRAY functions in your blocks, the block code will not need to be rewritten when you transfer blocks between Macintosh and Windows systems. However, the extension will have to be rebuilt. See "Transferring files between operating systems" on page P274 for more information.

See the DLL functions on page P118 for more information about these two interfaces.

Turning code into a DLL
Taking existing code and turning it into an DLL involves the following:

▸ You must write or edit the DLL code using a Windows compiler which must be capable of compiling DLLs. The DLL must be built for 32-bit execution.

▸ You need to select one of the two Extend DLL interfaces discussed above. The cross platform interface is very similar on both Windows and Macintosh platforms, and makes moving code from one platform to the other easier. However, this interface is more limited and less flexible and therefore may not be the best for all applications. For example, it will not work with any existing Windows DLLs, and the argument list and return values are constrained by the Macintosh XCMD interface.

▸ You need to slightly modify the DLL code by adding the DLL calling interface. This will be different depending on whether you use the cross platform interface or the existing Windows DLL interface.

▸ After compiling the code, you will have a DLL file. Place this file in the Extend Extensions directory and *restart Extend*. This will allow you to call the external code using the DLL functions in your block code.

You should also make note of the following:

- If there is any possibility that your blocks will be used cross-platform (on both Windows and Macintosh systems), the code in your blocks should include checks to allow for the differences between platforms. For more information, see "Extensions and drivers" on page P277.

- Real and integer variables passed from the ModL code to a DLL are passed by value. Real variables are 8 byte double precision and integers are 4 byte long integers.
- Strings and arrays are passed to DLLs as pointers to data that has been allocated by Extend. Modifications to that data will affect the original information in Extend.
- As discussed above, arrays that are passed to a DLL come through as pointers to the original data in Extend. Accessing and modifying the data is fine, but you should not try and resize the pointer. If you do, Extend will not be able to access the data and will probably crash.
- Strings are passed to DLLs from ModL as Pascal strings, not C strings. This means that the string is preceded by a size byte and is not terminated by a zero. For example, if you pass a string to a DLL, the DLL will get a pointer to 256 bytes of data in which the first byte contains the number of characters in the string.

The section “Accessing code from other languages” on page P57 and the Extend ModLTips directory give examples of writing a DLL in C and calling it from within an Extend block.

Sounds

Depending on your operating system, Extend has access to several sounds. You can play sounds using the Sound block from the Generic library (*Macintosh*: Generic Lib; *Windows*: generic.lix) or by calling the PlaySound function in the code of a block.

Sounds on the Macintosh

There are three sources of sounds that Extend can access on the Macintosh system:

- Extend contains a “click” sound.
- Your System file contains default system sounds such as beep sounds. To see the system sound names, access the Sound device in the Control Panel under the Apple menu. The sounds listed in the Alert Sounds dialog are the system sounds.
- You can also use sound resources (SNDs) made by other Macintosh utilities or obtained from user groups.

In order to have access to the sound resources which are not Extend or System sounds, you must copy them into the Extensions folder, as discussed above.

Sounds on Windows

There are three sources of sounds that Extend can access on the Windows system:

- Extend contains a “click” sound extension.
- Your System contains default system sounds such as beep sounds. To see the system sound names, access the Sound device in the Control Panel in the Main program group in Window’s Program Manager. The sounds listed as .WAV files are the system sounds.

- You can also use .WAV files created by other Windows applications or obtained from user groups.

In order to have access to the sounds which are not System sounds, you must copy the .WAV file into the Extensions directory, as discussed above.

Picture and movie files

Pictures and movies are used for specialized animation. See the animation functions in "Animation" on page P123, for more detail. To see an example of using pictures and movies for animation, see "Showing a picture or movie on an icon" on page P190. "File conversion" on page P275 provides information about converting pictures between Windows and Macintosh operating systems.

Pictures and movies on the Macintosh

As mentioned in "Extensions on the Macintosh" on page P236, pictures and movies must be stored as a resource file in Extend's Extensions folder. PICT resources are not the same as PICT files. In order to use pictures for animation, they must be saved as a resource file with a PICT-type resource and not as a PICT file from a graphics or drawing program. Movies are QuickTime movies.

Pictures and movies on Windows

As mentioned in "Extensions on Windows" on page P237, pictures and movies must be stored as a file in Extend's Extensions directory. Movies are .MOV files created by a 32-bit compatible version of QuickTime for Windows. The AnimationPicture function will recognize either a .WMF file, a .BMP file, or a Macintosh picture resource that has been converted using Extend's MacWin Converter utility.

Drivers (*Macintosh only*)

Drivers are code segments which allow you to access external devices and functions. They are more difficult to use than XCMDs and the user must be familiar with the driver architecture of the Macintosh (see *Inside Macintosh*, Addison Wesley).

If you have an existing driver and want to access it from Extend, Extend's driver functions will be helpful. The documentation that came with your software driver should provide more information about exactly how your driver needs to be called. Unless otherwise instructed by your device driver documentation, drivers must be stored as a resource file in Extend's Extensions folder, as discussed above. See the driver functions in "Other drivers (Macintosh only)" on page P116, for more detail.

Protecting libraries

If you build your own blocks and you do not want others to have access to your block code, you can protect your libraries by removing the ModL code of the blocks. Extend normally keeps both the ModL code and compiled code in the block; when you remove the ModL code, the compiled code is still there. When a user attempts to edit a protected block, Extend displays a dialog message that the block is protected and cannot be opened. The structure and dialog windows for the protected block are never shown.

A protected library can be used the same as any other library except that you cannot view or alter the ModL code. This means that someone using the library has all the functionality of the blocks in that library but no ability to see how the blocks work. This is a convenient way to hide proprietary programming while still giving users full access to the power of the block. Protecting the ModL code has the added benefit of preventing someone from changing the icon or the block's help text.

Note After a library is protected, *you can never unprotect it*. Before Extend protects a library it makes a copy of the library. However, it is a good practice to make an additional copy of the original and store it away from your computer on a diskette.

The steps for protecting a library are:

- Select *Protect Library* in the Tools command of the Libraries menu.
- When you give the Protect Library command, Extend warns you that protecting a library's ModL code will permanently and irrevocably prevent access to block code.
- In the dialog that appears, select and open the library you want to protect.
- In the "Create new library" text field, give a name for the library. Note that you can only give the same name as the original library if you save the protected library to a new location. This insures that you are protecting a copy of the library, not the original. Since your models will be expecting the original library name, it is suggested that you save the library to a new location, rather than changing its name.
- Extend protects the library ModL code by cutting it from the block.

Any copies you make of this protected library will also be protected. You can easily show that a library is protected by opening it and attempting to edit a block in the library using the Open Block Structure command in the Define menu.

Please note that libraries require the Extend application to run. You may distribute the libraries you build to other Extend owners but, under the terms of your Extend license agreement, you are specifically prohibited from distributing copies of the Extend application or its libraries. The Extend RunTime and Player licenses allow you to distribute a specialized version of Extend so users can change parameters and run models that you build, but cannot build their own models. Contact Imagine That, Inc. for more information.

Cross-platform development

The ModL language can only be compiled or interpreted by Extend. Extend will execute ModL code in the same way regardless of the platform it is running on. However, Extend is capable of communicating with other applications through ModL code, and some of these applications may not have a consistent interface across platforms. The following is a discussion of some of the things which should be considered when writing ModL code that is intended to be cross-platform compatible.

IPC cross-platform and multi-server considerations

The interprocess communication (IPC) functions allow Extend to act as a client application and request data and services from server applications. When using the IPC functions, it is important to remember that the syntax of some of the function arguments are dependent both on the server application which Extend is communicating with and the platform that Extend is running on. For example, the *serverName* argument of the IPCConnect function (which defines the application you want to connect Extend to) must be in the format appropriate for that application and platform. The *serverName* for Microsoft Excel on the Windows operating system would be “Excel”, while on the Macintosh operating system it would be “XCEL”. This type of information can be found in the documentation of the server application.

When writing ModL code that utilizes IPC functions and is meant to be cross-platform compatible or capable of operating with multiple server applications, it is necessary to define the function arguments according to the platform application *prior* to calling the function. This can be accomplished by using the platform variables PLATFORMPOWERPC, and PLATFORMWINDOWS) and using dialog items to define the server application (see blocks in the IPC library for examples).

For instance, the following code establishes a conversation between Extend (the client) and Excel (the server) on either the Windows or Macintosh operating system:

```
if (PlatformPowerPC)
     serverName = "XCEL";
if (PlatformWindows)
     serverName = "Excel";
IPCConnect (serverName, theSpreadsheet);
```

The following is a list of functions that contain arguments where the syntax of the argument depends on the server application (the dependent argument is shown in italics):

- IPCConnect(*serverName*, topic)
- IPCExecute(conversation, *executeData*, item)
- IPCPoke(conversation, pokeData, *item*)
- IPCRequest(conversation, *item*)

Chapter 5: Using the ModL Source Code Debugger

A guide to debugging your blocks using the ModL debugger

"We have gone beyond the absurd...
our position is ridiculous."
— John Vacarro

Code, any code in any language, hardly ever works correctly the first time. This chapter will show you how the ModL source code debugger can save your sanity when building your own library blocks. You will learn how to step through lines of source code, examine values of variables, create breakpoints with conditions, and analyze problems with blocks.

First we will present an overview of debugging to define some terms, and then a short tutorial so that you will become comfortable with Extend's source code debugger.

Overview of debugging

The motivation behind the use of a source code debugger is that it makes it much easier to find out what causes a malfunction in a block. You can watch the execution of the ModL code and see its path and the effects it has on any of the variables used in the block.

The blocks in an Extend library contain resources which describe the dialog, on-line help, icon, and connectors, but what we will be concerned with is the ModL source code and its compiled machine code. Let's define some of the terms we will be using here:

Breakpoint: Where we want the debugger to initially stop execution of the block's code and open the debugger's window. We can then manually step through lines of code to trace its execution and examine the effects on the variables defined in that code and in the dialog and connectors of the block.

Breakpoint condition: The TRUE or FALSE boolean decision that causes the breakpoint to break execution only under certain circumstances. This is valuable if there could be too many breaks and an important break only occurs rarely, with certain values of variables.

Step over: We want the debugger to execute the line of code with a function call, but not to trace the code of the function.

Step into: We want the debugger to trace the actual function call, including the code of the function.

Quick steps to debugging

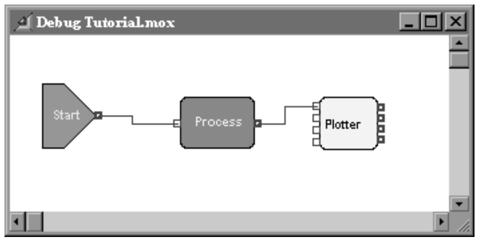
Briefly, here are the steps to debug a single block.

- Select the block to be debugged, on the model.
- Right-click the block or choose "Open Set Breakpoints Window" in the Define menu. This automatically recompiles the block with debugging code if it wasn't already set up for debugging previously.
- Add breakpoints to the debugging window code by clicking in the margins.
- Close the debugging window and run the model or click an item in the block's dialog to execute the code.

- When the breakpoint is reached, step through the code and examine the variables.

Debugger tutorial

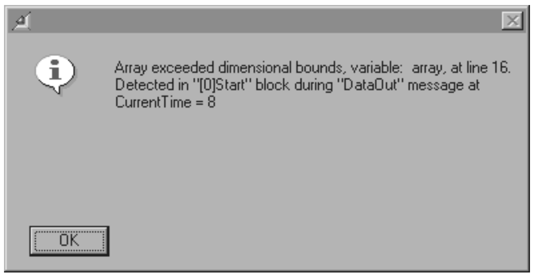
Open the DebugTutorial model in the Tutorials folder. You will see three connected blocks:



DebuggerTutorial model

This model uses custom blocks that generate data, processes the data, and then plots the result.

Run the model. You will see an Extend error message:

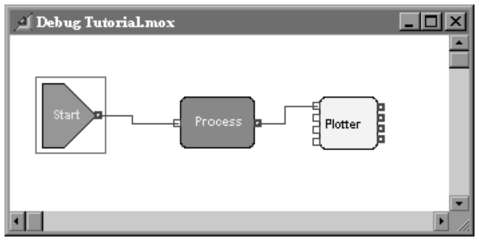


Array exceeded dimensional bounds, variable: array, at line 16.
Detected in "[0]Start" block during "DataOut" message at
CurrentTime = 8

Error message

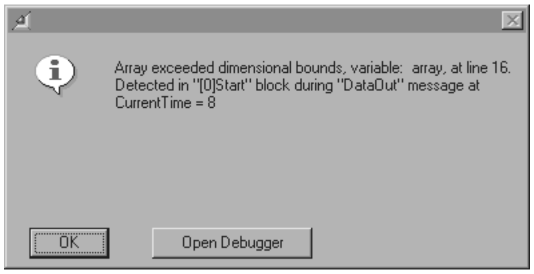
Setting up for debugging

Wouldn't it be nice if we could figure out how this error is being generated? Select the Start block and right-click the block and choose *Open Set Breakpoints Window* (also in the Define menu). This shortcut automatically recompiles the block, if necessary, in debugging mode and opens the block's *Set Breakpoints* window. We don't need to set a breakpoint yet, so just close the *Set Breakpoints* window. Note the new appearance of the model:



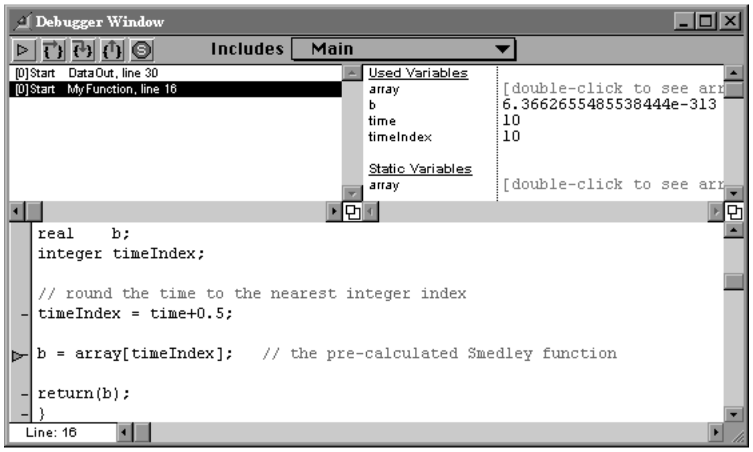
Model with Start block debugging code

Extend shows you which blocks have been compiled in debugging mode. This reminder is important because those blocks will execute more slowly in a model and you will want to remember to remove their debugging code when you are done. Now run the model again. Notice how the error message now has an Open Debugger button?



```
Array exceeded dimensional bounds, variable:  array, at line 16.
Detected in "[0]Start" block during "DataOut" message at
CurrentTime = 8
```

Click the Open Debugger button. This will open the Debugger Window.



```
[0]Start   DataOut, line 30
[0]Start   MyFunction, line 16
```

```
Used Variables
array          [double-click to see arr
b              6.3662655485538444e-313
time           10
timeIndex      10

Static Variables
array          [double-click to see arr
```

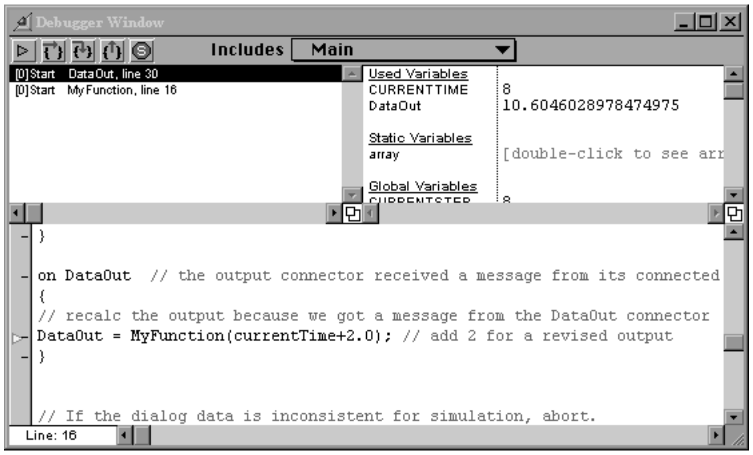
```
real    b;
integer timeIndex;

// round the time to the nearest integer index
timeIndex = time+0.5;

b = array[timeIndex];   // the pre-calculated Smedley function

return(b);
}
```

Debugger window after error message

Look at the line of code where the error occurred. Notice that the index variable *timeIndex* has a value of 10, but the variable *Array* is declared at the top of the script with a number of members equal to 10, indexed as 0 to 9.

But how did MyFunction get called? What was the chain of events that led to this error message?

Click the top entry (above the selected entry) in the upper left pane, known as the *Call Chain* pane.



```
[0]Start   DataOut, line 30
[0]Start   MyFunction, line 16
```

```
Used Variables
CURRENTTIME        8
DataOut            10.6046028978474975

Static Variables
array              [double-click to see arr

Global Variables
```

```
}

on DataOut  // the output connector received a message from its connected
{
// recalc the output because we got a message from the DataOut connector
DataOut = MyFunction(currentTime+2.0); // add 2 for a revised output
}


// If the dialog data is inconsistent for simulation, abort.
```

Line: 16

Debugger window after clicking top entry in call chain

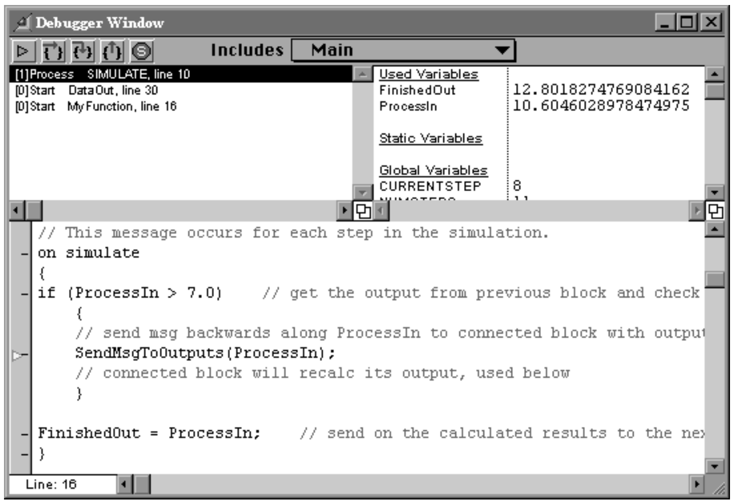
Notice that the location arrow in the left column is yellow. That means that we are looking at a previous entry in the *Call Chain*. Notice that the message handler is called *DataOut*, the name of one of the connectors on the block. That means we received a connector message from a connected block. It would be nice if the block that sent the message could be seen on the *Call Chain* so we can click on its entry and see the actual code that caused this chain of events.

To see a block in the *Call Chain*, it has to be compiled in debugging mode. The fastest way to insure that all the blocks in a model are compiled in debugging code is to open their library windows and compile them all in debugging mode.

To do this, close the Debugger Window, go to the Library menu, select Debug Tutorial, select *Open Library Window* (Note that you do not want any other library windows open when you do this operation, so close any other open library windows), then select *Add Debug Code to Open Library Windows*.

After Extend finishes compiling all of the blocks in the *Debug Tutorial* library, all the blocks on the model will now be outlined in red, showing that they are currently in debugging mode.

Run the model again. When the error occurs, click the Open Debugger button and look at the debugger window. Now click on the top *Call Chain* entry; the one that started this whole chain of events.



```
[1]Process   SIMULATE, line 10
[0]Start   DataOut, line 30
[0]Start   MyFunction, line 16
```

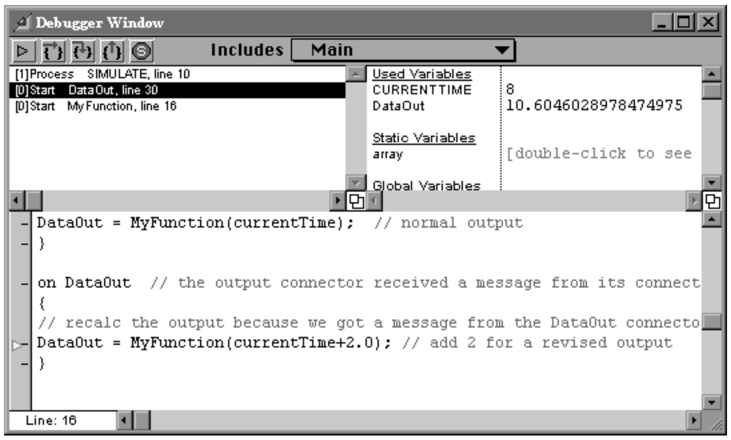
```
Used Variables
FinishedOut        12.8018274769084162
ProcessIn          10.6046028978474975

Static Variables

Global Variables
CURRENTSTEP        8
```

```
// This message occurs for each step in the simulation.
on simulate
{
if (ProcessIn > 7.0)    // get the output from previous block and check
    {
    // send msg backwards along ProcessIn to connected block with output
    SendMsgToOutputs(ProcessIn);
    // connected block will recalc its output, used below
    }

FinishedOut = ProcessIn;    // send on the calculated results to the ne
}
```

Line: 16

Debugger window with top Call Chain entry selected

What sent that message in the Start block? There was a SendMsgToOutputs function call from the Process block because its input was greater than 7.0. Its actual value can be seen in the variables pane as 10.6046. Now click on the second *Call Chain* entry:



```
[1]Process   SIMULATE, line 10
[0]Start   DataOut, line 30
[0]Start   MyFunction, line 16
```

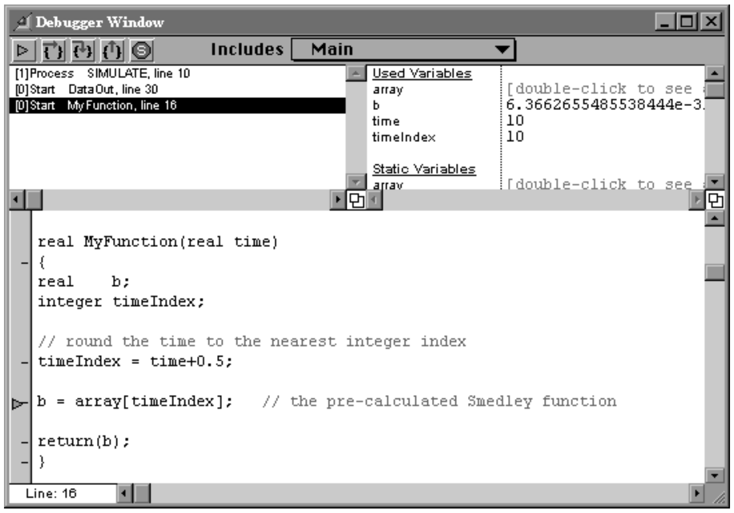
```
Used Variables
CURRENTTIME        8
DataOut            10.6046028978474975

Static Variables
array              [double-click to see

Global Variables
```

```
DataOut = MyFunction(currentTime);  // normal output
}

on DataOut  // the output connector received a message from its connect
{
// recalc the output because we got a message from the DataOut connecto
DataOut = MyFunction(currentTime+2.0); // add 2 for a revised output
}
```

Line: 16

Debugger window with message handler selected

It looks like we are calling MyFunction with a valid argument equal to 8.0+2.0 or 10.0... or are we? Click on the bottom *Call Chain* item where the error occurred:



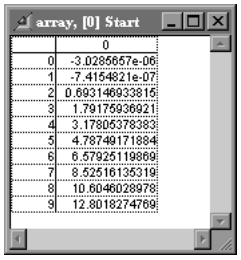
```
[1]Process   SIMULATE, line 10
[0]Start   DataOut, line 30
[0]Start   MyFunction, line 16
```

```
Used Variables
array        [double-click to see
b            6.3662655485538444e-3
time         10
timeIndex    10

Static Variables
array        [double-click to see
```

```
real MyFunction(real time)
{
real    b;
integer timeIndex;

// round the time to the nearest integer index
timeIndex = time+0.5;

b = array[timeIndex];   // the pre-calculated Smedley function

return(b);
}
```

Debug window with error point selected

Since we know that *Array* was indexed beyond its bounds, let's check *Array* by double-clicking it in the Variables pane:



	0
0	-3.0285657e-06
1	-7.4154821e-07
2	0.693146933815
3	1.79175936921
4	3.17805378383
5	4.78749171884
6	6.57925119869
7	8.52516135319
8	10.6046028978
9	12.8018274769

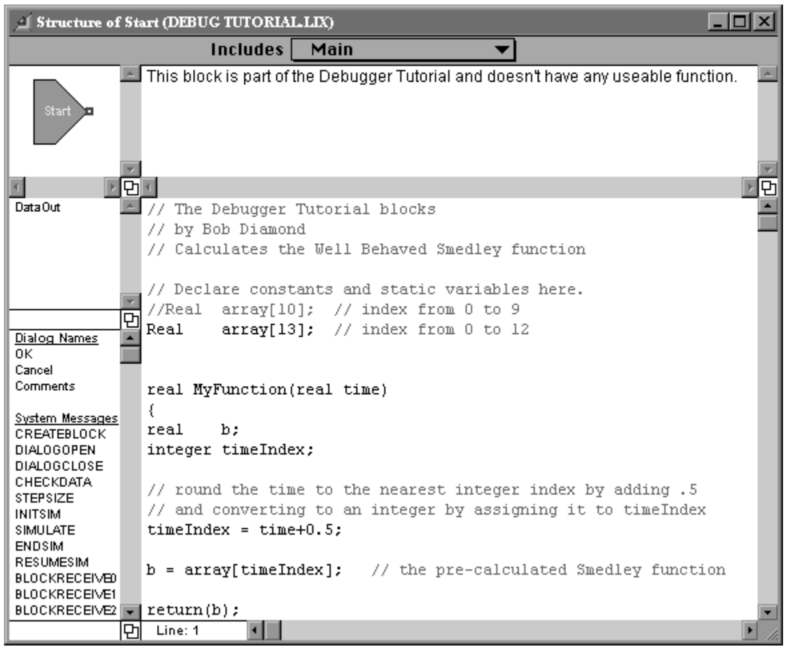
Array window

Notice how the array is indexed from 0 to 9. Since we tried to index it with 10, we got an error message because an index of 10 is outside its bounds!

How can we fix this model? We know that it is being run from a currentTime of 0 to a currentTime of 10, and we could be adding 2 to the currentTime in some cases, so we need to increase the size of Array so that it can be indexed from 0 to at least 12. Let's do that so that the model will work.

Close the Debugging window.

Then Alt-Double-click the Start block so that we can edit the code:



Structure of Start (DEBUG TUTORIAL.LIX)

Includes: Main

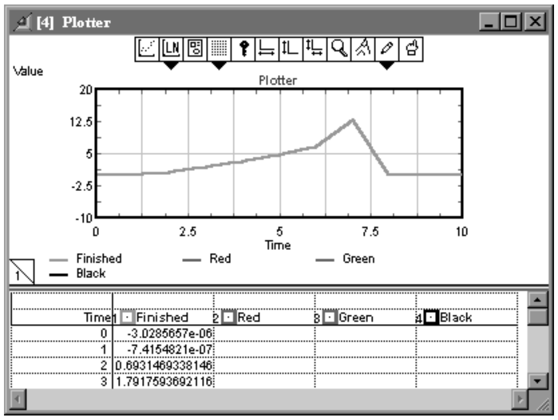
This block is part of the Debugger Tutorial and doesn't have any useable function.

```
// The Debugger Tutorial blocks
// by Bob Diamond
// Calculates the Well Behaved Smedley function

// Declare constants and static variables here.
//Real  array[10];  // index from 0 to 9
Real    array[13];  // index from 0 to 12


real MyFunction(real time)
{
real    b;
integer timeIndex;

// round the time to the nearest integer index by adding .5
// and converting to an integer by assigning it to timeIndex
timeIndex = time+0.5;

b = array[timeIndex];   // the pre-calculated Smedley function

return(b);
```

Structure of the Start block

Change the declaration of Array[10] to Array[13] as shown. Then close the Structure window and compile the block.

Try runing the model again and observe the plotter:

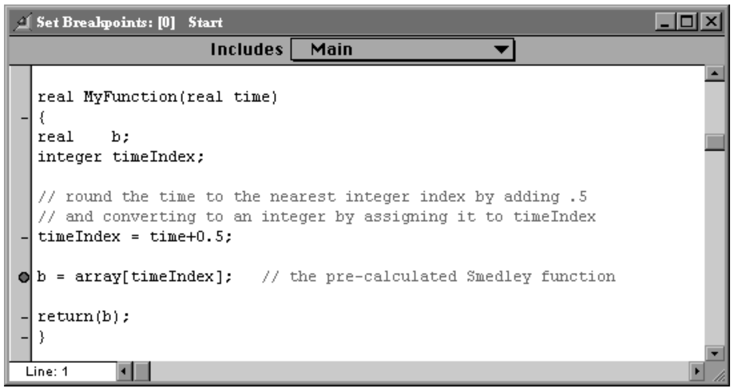


Time	1 Finished	2 Red	3 Green	4 Black
0	-3.0285657e-06			
1	-7.4154821e-07			
2	0.6931469338146			
3	1.7917593692116			

Plotter showing strange drop off of Smedley function output

The plotter shows a drop off in values where everyone (everyone?) knows the Smedley function should be increasing. We need to set a breakpoint and see what is going on here!

Right-click on the Start block in the model and select *Set Breakpoints Window*. This shows the source with a left margin. Click to the left of the *B = array[timeIndex];* statement in the MyFunction declaration:

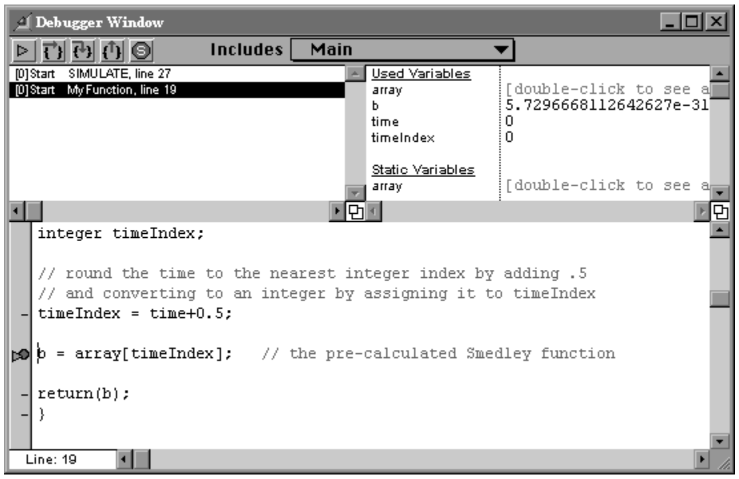


```
real MyFunction(real time)
{
real    b;
integer timeIndex;

// round the time to the nearest integer index by adding .5
// and converting to an integer by assigning it to timeIndex
timeIndex = time+0.5;

b = array[timeIndex];   // the pre-calculated Smedley function

return(b);
}
```

Start block debugging window

Notice the red dot that appears at that breakpoint position. If we run the model and it breaks at that point, we should be able to see some of the values that cause the problems.

Run the model:



```
[0]Start   SIMULATE, line 27
[0]Start   MyFunction, line 19
```

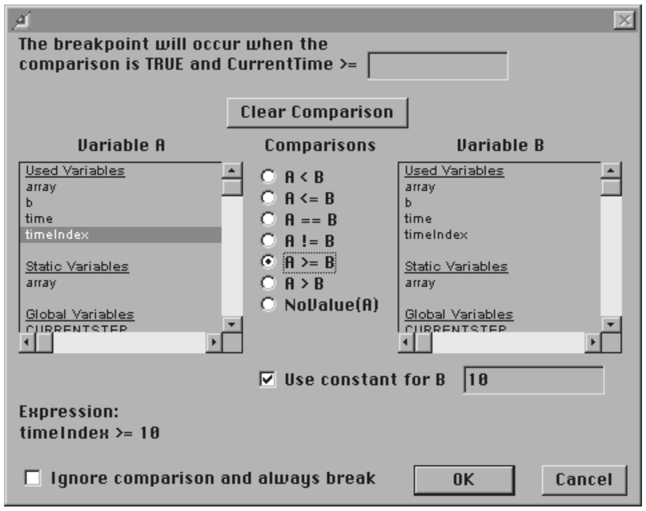
```
Used Variables
array        [double-click to see a
b            5.7296668112642627e-31
time         0
timeIndex    0

Static Variables
array        [double-click to see a
```

```
integer timeIndex;

// round the time to the nearest integer index by adding .5
// and converting to an integer by assigning it to timeIndex
timeIndex = time+0.5;

b = array[timeIndex];   // the pre-calculated Smedley function

return(b);
}
```

Breakpoint reached

Notice how the Debugger window opens and displays the breakpoint. Looking at the variables, we can see that the timeIndex is 0, so we need to click the green triangle (Continue) until the breakpoint occurs again. This can get tedious... clicking until a *timeIndex* of 10 is finally reached. There is a better way. Why don't we set a condition on the breakpoint so we only break when *timeIndex* is greater or equal to 10!

First stop the run. Then go to the Windows menu and select the Breakpoints window (or click on the Breakpoints window) to bring it forward. Now double-click on the right column of our breakpoint. A new window should appear, letting us set a condition for this breakpoint:



The breakpoint will occur when the comparison is TRUE and CurrentTime >=

Clear Comparison

Variable A	Comparisons	Variable B
Used Variables array b time timeIndex Static Variables array Global Variables CURRENTSTEP	○ A < B ○ A <= B ○ A == B ○ A != B ● A >= B ○ A > B ○ NoValue(A)	Used Variables array b time timeIndex Static Variables array Global Variables CURRENTSTEP

☑ Use constant for B 10

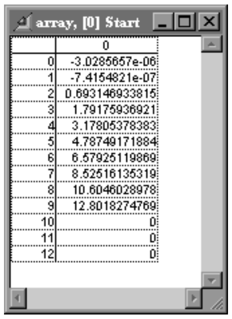
Expression:
timeIndex >= 10

☐ Ignore comparison and always break

Condition window

Here we click on *timeIndex*, the >= radio button, the *Use constant* checkbox, and enter a 10 for the constant. Now run this model again. When the breakpoint occurs, click on the *Step over* button so we can see the value of *b* calculated (in this case, the *Step over* and *Step into* buttons do the same thing, because there is no function call to step into).

Notice that the value of b that was calculated was zero! Double click the Array variable so we can look at its values:

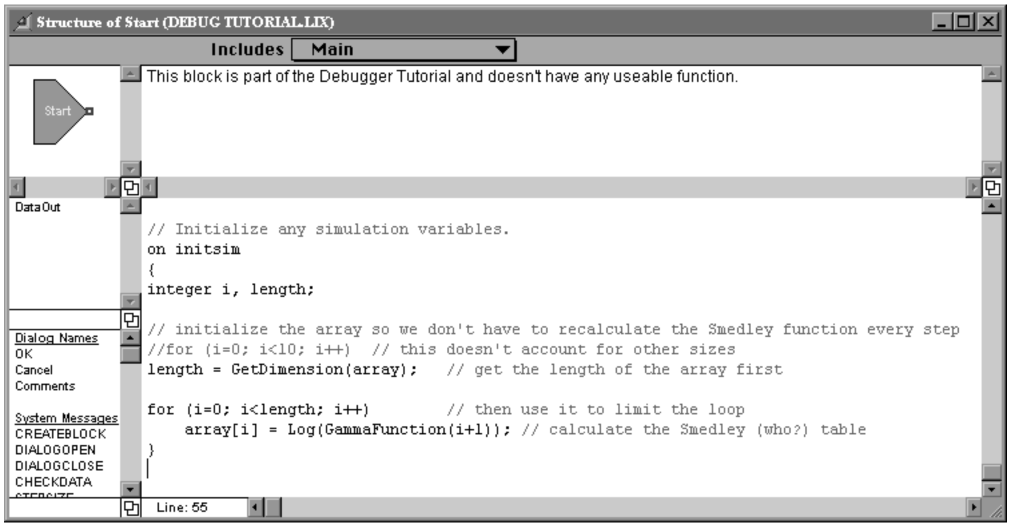


array, [0] Start

	0
0	-3.0285657e-06
1	-7.4154821e-07
2	0.693146933815
3	1.79175936921
4	3.17805378383
5	4.78749171884
6	6.57925119869
7	8.52516135319
8	10.6046028978
9	12.8018274769
10	0
11	0
12	0

Array showing zeroes in top three members

It looks like we forgot to calculate the Smedley values for the new larger array that we declared. Let's change the code so we will always fill in the entire array:



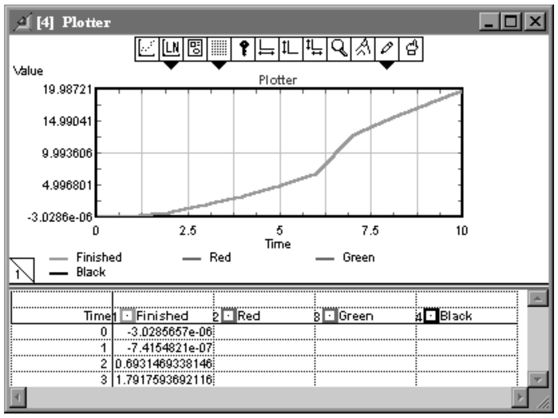
This block is part of the Debugger Tutorial and doesn't have any useable function.

```
// Initialize any simulation variables.
on initsim
{
integer i, length;

// initialize the array so we don't have to recalculate the Smedley function every step
//for (i=0; i<10; i++)  // this doesn't account for other sizes
length = GetDimension(array);   // get the length of the array first

for (i=0; i<length; i++)        // then use it to limit the loop
    array[i] = Log(GammaFunction(i+1)); // calculate the Smedley (who?) table
}
```

Fixed code in Start block

Instead of hard coding a <10 to limit the loop, we have called a function to return the array length into a new variable called *length*, and we use that to limit the calculation loop. That way, we can always change the size of the array and we will always fill in the entire array.

Remove the breakpoint in the Breakpoints window by selecting it and hitting the delete key. Now, let's run the model again. It's nice to see the Smedley function so well behaved:



Time	1 Finished	2 Red	3 Green	4 Black
0	-3.0285657e-06			
1	-7.4154821e-07			
2	0.6931469338146			
3	1.7917593692116			

The well behaved Smedley function

We could go a step further, making *Array* a dynamic arrray and then resizing it in *InitSim*. Then the model could then be run for any *EndTime* value without overrunning *Array*.

This should have given you a taste of how valuable a debugger is. The next section talks about the details of the debugging system in Extend.

Debugger details

Preparations

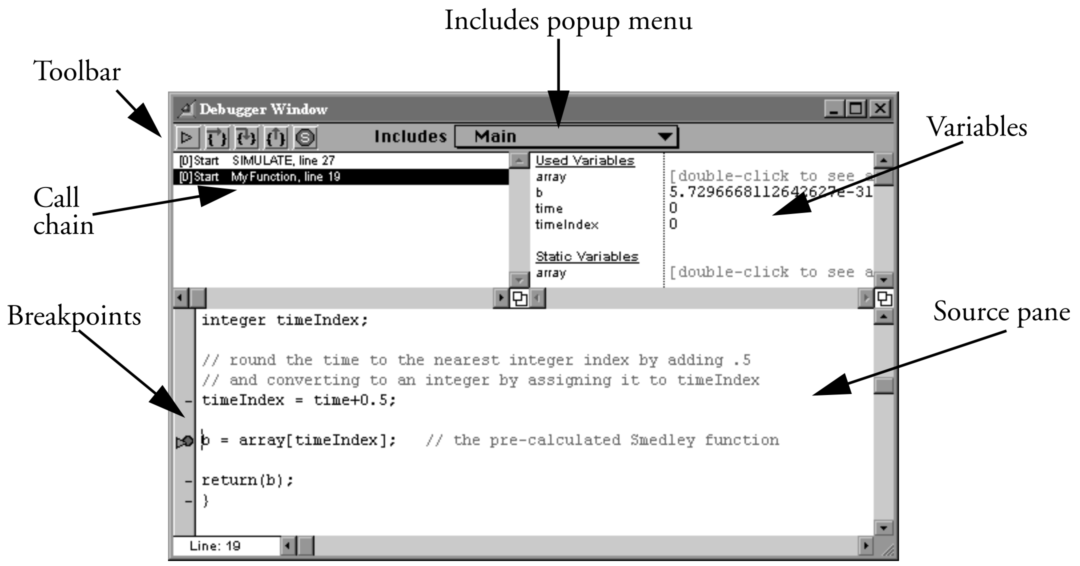
In order to debug a block, Extend has to generate debugging code for that block in the library. You can either:

- Select a block or blocks on the model and choose the Define:Open Set Breakpoints Window command. This automatically generates debugging code in the library for the selected blocks and opens a debugging window for each block selected.
- Open the library window for some or all libraries in your model and choose the Library:Tools:*Add Debugging Code to Open Library Windows* menu item. This takes longer, but is very useful if you are trying to debug a DE model, as the messages will be traceable back to their originators via the Call Chain in the Debugger Window. Then, select the blocks you wish to add breakpoints to, and right-click or choose the Define:Open Set Breakpoints Window command.

You can set a breakpoint in a block's debugging window by clicking one of the dashes in the left margin. Clicking again toggles the breakpoint.

When the model is run and execution reaches the breakpoint, the Debugger window will open and show the ModL code, values of variables, and the Call Chain.

Debugger window



```
[0]Start   SIMULATE, line 27
[0]Start   MyFunction, line 19
```

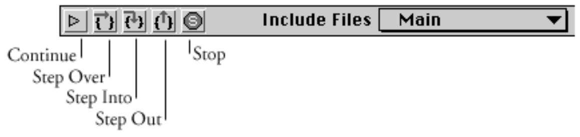
```
Used Variables
array        [double-click to see a
b            5.7296668112642627e-31
time         0
timeIndex    0

Static Variables
array        [double-click to see a
```

```
integer timeIndex;

// round the time to the nearest integer index by adding .5
// and converting to an integer by assigning it to timeIndex
timeIndex = time+0.5;

b = array[timeIndex];   // the pre-calculated Smedley function

return(b);
}
```

```
Line: 19
```

Debugger window

Tools



Debugger tool bar

Continue: This just continues execution until the next breakpoint is reached.

Step Over: This executes a function call without stepping into the function code.

Step Into: This steps into the function call, tracing execution of the function.

Step Out: This continues execution until it gets to the caller of the function.

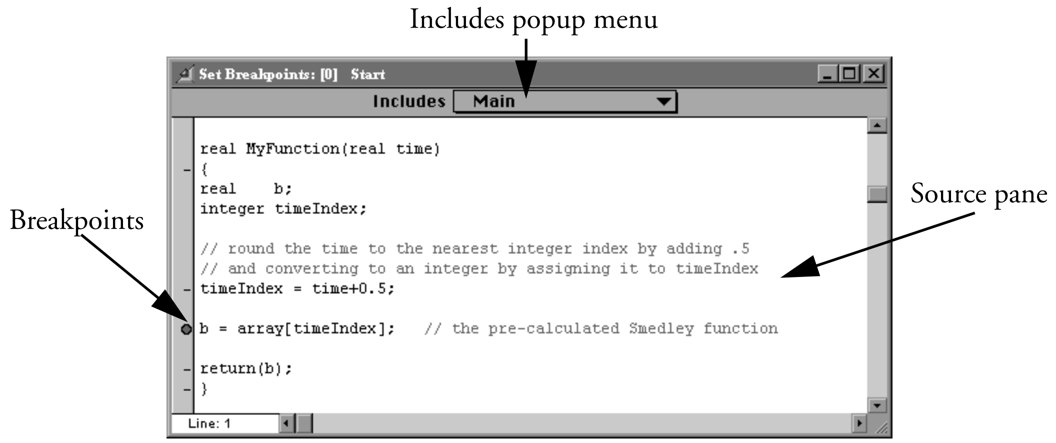
Stop: This aborts the debugging session.

The *Includes* popup menu shows which include files are used in the block, and allows you to set breakpoints in an include by displaying its contents in the source pane.

The Breakpoints margin shows any breakpoints as a red circle. A red half-circle indicates a conditional breakpoint.

To add a condition to a breakpoint, use the Breakpoints window.

Set breakpoints window



```
real MyFunction(real time)
{
real    b;
integer timeIndex;

// round the time to the nearest integer index by adding .5
// and converting to an integer by assigning it to timeIndex
timeIndex = time+0.5;

b = array[timeIndex];   // the pre-calculated Smedley function

return(b);
}
```

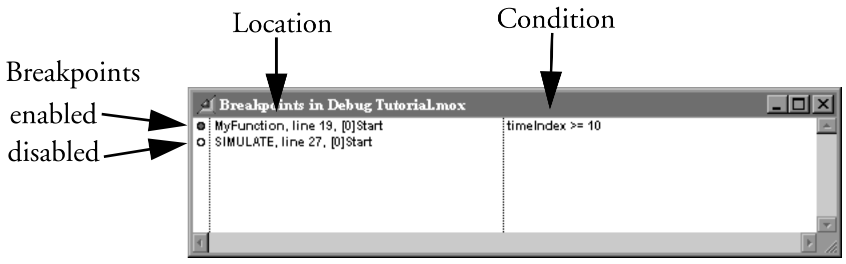
Set breakpoints window

The *Includes* popup menu shows which include files are used in the block, and allows you to set breakpoints in an include by displaying its contents in the source pane.

The Breakpoints margin shows any breakpoints as a red circle. A red half-circle indicates a conditional breakpoint.

To add a condition to a breakpoint, use the Breakpoints window.

Breakpoints window



MyFunction, line 19, [0]Start	timeIndex >= 10
SIMULATE, line 27, [0]Start	

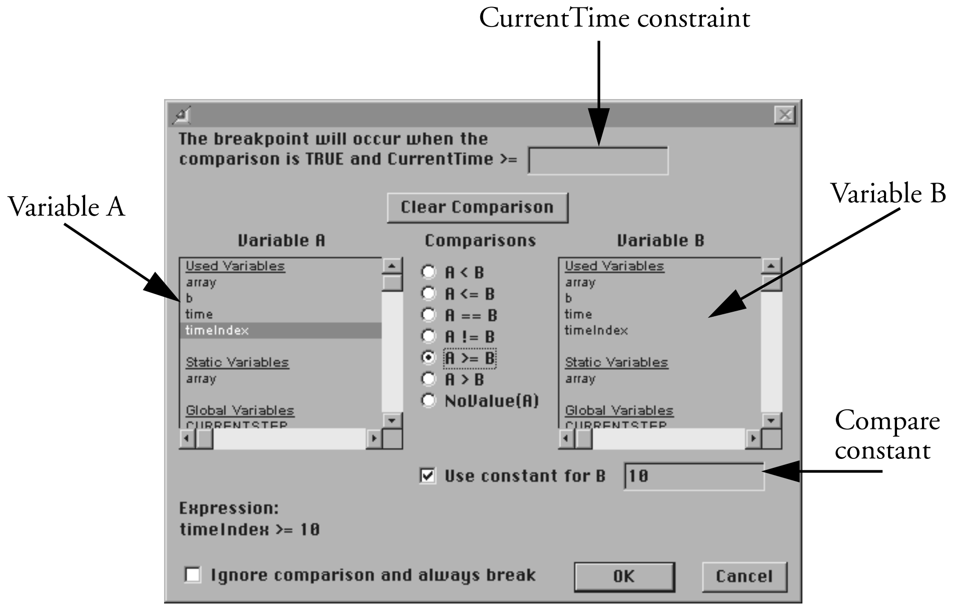
The Breakpoint window, in the Windows menu, allows the user to both enable and disable, delete, and add conditions to breakpoints. Conditions allow the breakpoint to be ignored unless the condition is TRUE.

To Enable/disable a breakpoint in the Breakpoint window, click on the red circle. When it is red, the breakpoint is enabled. When it is empty, it is disabled. This is powerful because you don't have to delete a breakpoint to temporarily disable it.

To delete a breakpoint, select it's location and hit the Delete key.

To enable a breakpoint condition, double-click its condition column, then click on the left variable, compare condition, and right variable, or click the checkbox under the right variable window and use a constant to compare. You can change or disable the condition, in that window, if you decide you don't want it anymore.

Conditions dialog



The breakpoint will occur when the comparison is TRUE and CurrentTime >=

[✓ A >= B]

[✓] Use constant for B: 10

Expression:
timeIndex >= 10

[ ] Ignore comparison and always break

Conditions dialog

Conditional breakpoints are useful when you might get too many breakpoints and you are only interested in a breakpoint that occurs at a specific time or when a variable reaches a specific value

The Conditions dialog makes it easy to construct a conditional breakpoint with no coding. Just enter a *currentTime* value, and enter any other comparison that might be helpful in narrowing down the problem.

To enter a comparison, first select a variable from column A, then choose a variable from column B or enter a constant and check the *Use constant for B* checkbox. Click the desired comparison operator radio button and click on *OK* to save the condition or *Cancel* to reject it.

You can also click the *Ignore comparison...* checkbox so that your entered condition will be saved but ignored for the present.

Debugging tips

- To quickly debug a block, select the block and choose Set Breakpoints Window (right-click the block) command. This checks the block structure, recompiles the block for debugging, and opens the set breakpoint window.
- To debug the sending of interblock messages, compile the entire library for debugging using the Library Tools menu. That way, the call chain will contain the entire chain of messages and you can see who sent what to whom.
- If you are finished with debugging your blocks, you might want to remove all of the debugging code, as it slows execution considerably. To do this, open all of the library windows, choose Library:Tools:Remove debugging code.

Appendix A: Upper Limits

A list of the maximum numbers of things that you can do at one time

"The thing I am most aware of is my limits."
— André Gide

Like every program, Extend has its limits. It is unlikely that you will find them in your normal work, but it is good to know what they are in case you come across them.

Blocks in a model	2 billion
Output connectors in a model (nodes)	2 billion
Connectors per block	255
Length of a block's ModL code(characters)	4 megabytes
Dialog items in a block	1024
Dynamic arrays (each array)	2 gigabytes
Number of array dimensions	5
Maximum index for array dimensions	2 billion
Dynamic arrays per block	255
Text files open at one time	200
Blocks in a library	200
Libraries open at one time	30
Columns in a table	255 (data table); 255 (text table)
Total table size (cells) per block for static data tables	3260 (data table); 2030 (text table)
Total table size (cells) each, for dynamic data tables	2 billion
Steps in a simulation run	2 billion
Block name length, characters	31
Variable name length, characters	63
Number of attributes for discrete event item	300
User defined function arguments	127
Nested loops	32
Number of simulations in a multiple run	32K
Maximum, minimum of real numbers	±1E±308
Maximum, minimum of integer numbers	±2,147,483,647
Significant figures in real calculations	Macintosh: 20 (IEEE extended) PowerMacintosh: 16 (double) Windows: 16 (double)

Appendix B: ASCII Table

A table to help you determine the values of the ASCII characters

"I would sooner read a timetable or catalogue than nothing at all. They are much more entertaining than half the novels that are written."
— W. Somerset Maugham

This table shows the ASCII values from 00 to 127, which are the same for Macintosh and Windows. Values above 127 are not part of the standard ASCII set and vary depending on the font.

00	NUL	32	space	64	@	96	`
01	SOH	33	!	65	A	97	a
02	STX	34	"	66	B	98	b
03	ETX	35	#	67	C	99	c
04	EOT	36	$	68	D	100	d
05	ENQ	37	%	69	E	101	e
06	ACK	38	&	70	F	102	f
07	BEL	39	'	71	G	103	g
08	BS	40	(	72	H	104	h
09	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Appendix C: Menu Command Numbers

A list of the menu command numbers to be used with the ExecuteMenuCommand() function

"Obedience alone gives the right to command."
— Ralph Waldo Emerson

The ExecuteMenuCommand(commandNumber) function executes a specified menu command (see “Scripting” on page P157). This is functionally the same as selecting the command from the menus. The following is a list of the command numbers that are used with the ExecuteMenu-Command() function.

File Menu	**Command Number**
New	2
New Text File	1601
Open	3
Open Text File	1602
Append Model	2015
Close	4
Revert Model	7
Save Model	5
Save Model As	6
Save Selected	2016
Import Data	2005
Export Data	2006
Show Page Breaks	2000
Page Setup	8
Print	9
Print Structure	2050
Get Info	2001
Quit / Exit	1

Edit menu	**Command Number**
Undo	16
Cut	18
Copy	19
Paste	20
Clear	21
Select All	23
Duplicate	2013

Edit menu	**Command Number**
Create Publisher	9000
Subscribe To	9001
Publisher Options	9002
Open Publishers/Subscribers	9003
Paste Link ⊞	9005
Delete Link ⊞	9006
Sensitize Parameter	2046
Show Clipboard	22
Preferences	9011

Library Menu	**Command Number**
Open Library	4500
Close Library	4501
New Library	4503

Library\Tools Menu	**Command Number**
Protect Library	2063
Set Library Version	2064
Convert to RunTime Library	2061
RunTime Startup Screen Editor	2062
Compile Library	1530
Compile Selected	1531

Model Menu	**Command Number**
Show Notebook	1401
Make Selection Hierarchical	1501
New Hierarchical Block	1502
Show Named Connections	2011
Hide All Connections	2012

Model Menu	**Command Number**
Reduce	2007
Expand	2008
Reduce to Fit	2009
Normal Size	2010
Lock Model	2028
Find Block	2017
Show Block Labels	2049
Show Block Numbers	2025
Show Simulation Order	2018
Open Sensitized Blocks	2019
Spoken Messages	2002

Model\Connection Lines Menu	**Command Number**
Non Right-Angle Connections	5002
Right-Angle Connections	5001
Right Pointing Arrows	1208
Left Pointing Arrows	1209
View Using Defaults	1250
Thin Line	1201
Medium Line	1202
Thick Line	1203
Hollow Line	1204
Solid Line	1205
Dashed Line	1206
New Connections Are Black	1210
New Connections Use Color	1211

Model\Controls Menu	**Command Number**
Slider	1301
Switch	1302
Meter	1303

Text Menu	**Command Number**
Size \ Other	3000
Plain Text	30
Bold	31
Italic	32
Underline	33
Outline	34
Shadow	35
Condensed	36
Extended	37
Align Left	41
Align Center	42
Align Right	40
Border	5000
Transparent	5100

Define Menu	**Command Number**
Build New Block	1503
Rename Block	1509
Open Block Structure	2047
Compile	1508
Set Block Type	1535
New Dialog Item	1510
Define New Tab	1551
Edit Tab	1552
Move Selected Items to Tab	1550

Define Menu	Command Number
Find	1512
Enter Selection	1513
Find Again	1514
Replace	1515
Replace, Find Again	1516
Replace All	1517
Shift Selection Left	1518
Shift Selection Right	1519
Goto Line	1523
New Include File	1520
Open Include File	1521
Delete Include File	1522

Run Menu	Command Number
Run Simulation	6000
Simulation Setup	6001
Use Sensitivity Analysis	2027
Show Animation	2020
Show Movies	2026
Generate Report	2050
Dialogs Report	2057
Statistics Report	2058
Add Selected To Report	2051
Add All To Report	2055
Remove Selected From Report	2052
Remove All From Report	2053
Show Reporting Blocks	2054
Stop	30000
Pause	30001
Step	30003
Resume	30002

Run\Debugging Menu	**Command Number**
Pause At Beginning	2033
Step Each Block	2024
Step Next Animation	2032
Step Entire Model	2023
Show Block Messages	2021
Only Simulate Messages	2022
Scroll To Messages	2030
Generate Trace	2040
Add Selected To Trace	2041
Add All To Trace	2045
Remove Selected From Trace	2042
Remove All From Trace	2043
Show Tracing Blocks	2044
Profile Block Code	2029

Help Menu (Apple Menu)	**Command Number**
About	256
Help	257

Block Dialog	**Command Number**
OK	100
Cancel	101

Simulation Status Bar	**Command Number**
Stop	30000

Simulation Status Bar	Command Number
Pause	30001
Resume	30002
Step	30003
Slower	30004
Faster	30005

Appendix D: Cross-Platform Considerations and Equivalents

File conversion, file name comparisons, and keyboard shortcuts for the Macintosh and Windows systems

"There never were, since the creation of the world, two cases exactly parallel."
— Lord Chesterfield

Libraries

 Macintosh: library names can be up to 31 characters long and usually end in "Lib", although this is not required. Libraries are stored in the Libraries subfolder within the Extend folder.

 Windows: Extend supports file names up to 64 characters for Windows 95, 98, ME, 2000, and Windows NT. Library names must end in the ".LIX" extension. Libraries are located in the *Libraries* subdirectory within the Extend directory.

Transferring files between operating systems

The Windows and Macintosh versions of Extend are cross-platform compatible. For example, if you build a model or a library on your Windows computer, you can move it to a Macintosh computer and Extend will read it, and vice versa.

There are three considerations when transferring Extend files between Windows and Macintosh systems: file name adjustments, physical transfer, and file conversion.

Note Models and libraries developed in an older version and different platform may not transfer successfully. It is strongly recommended that you upgrade to the latest version on the source platform, resave models, recompile libraries, and resave the structure of hierarchical blocks in libraries (see "Saving hierarchical blocks in a library" on page E259) before transferring your files.

File name adjustments

If you transfer files from a Macintosh to a Windows computer, you may need to change the name of your file before you transfer it. Extend supports file names of up to 64 characters for Windows 95, 98, ME, 2000, and Windows NT. File names must end in a three (3) character extension (the extensions are ".MOX" for Extend model names, ".LIX" for library names, and ".TXT" for text file names). To make your file names be in Windows format, change the name of the file *on the Macintosh computer* using Extend's Save As command (if the file is open) or using the Finder (if the file is not open).

If you are transferring files from a Windows computer to a Macintosh, you do not need to change the file name or delete the extension; the Macintosh system can read names up to 31 characters long.

Note It is important that you do not delete the .MOX extension from a Windows model file name before transferring the model to your Macintosh system. The .MOX extension is required so Extend can identify the file as a Windows model. After Extend has converted the Windows model to Macintosh format, you can save the model under the same name or a new name.

Physically transferring files

Once you have made any necessary file name adjustments, you need to physically transfer the files between Macintosh and Windows computers. This process depends on your system resources and is independent of Extend. For example, you might copy the file onto a floppy disk to be read by a program that is designed to read disks from other operating systems (such as PC Access on the Macintosh or Conversion Plus for Windows). If you have the two computers networked, you could send the files directly from one to the other. If you are using Macintosh System 8.6+, it can read and write disks formatted for Windows systems.

Note Libraries and extensions that you build on a Macintosh computer need to be converted *before* being physically transferred to the Windows system, as discussed below.

File conversion

Depending on the type of file, file conversion may be handled automatically by Extend or may involve using a conversion application. As discussed below, Extend automatically converts model files when they are opened on a different platform. If you program your own blocks, the Extend for Macintosh package includes a file conversion utility which you can use to convert libraries and picture resource extensions from one operating system format to another. Other extensions that you build, such as QuickTime movies, DLLs, and XCMDs require more extensive conversion. Include files are, of course, already cross-platform compatible.

Model files

The Windows version of Extend can read Extend model files created on the Macintosh as long as the name format is correct, as discussed above. The Macintosh version of Extend can read Extend model files created under Windows without file name modification (in fact, as discussed in the Note above, the .MOX extension should not be removed.) When you open a model file that was created on another operating system, Extend will notify you that it is converting the file from that system to the current one. Once you save the model file, it will be in the format of your current operating system.

If your models (including hierarchical blocks) use libraries that you have created yourself, and you have changed the name of those libraries, Extend will not be able to locate the library. In this case, Extend will ask you to find and select the correct library as described in "Searching for libraries and blocks" on page E142 in the Extend User's Guide. Keeping all your libraries in the Libraries folder will make this search process easier. Saving the model will cause the new libraries to be used from then on.

For model files that have blocks that access text files (such as the File Input block from the Generic library) you may need to change the name of the text file that is being read to conform to platform requirements, as discussed above. Be sure to also change the name of the file in the File block's dialog to correspond to the new file name.

Note The first time you run a model that has been transferred from one operating system to another, any Equation blocks in the model will recompile to the format of the new system at the beginning of the simulation run.

Hierarchical blocks in libraries

If you have a hierarchical block saved in a library and you have renamed any of the libraries of the blocks *inside* the hierarchical block (for example, to comply with Windows format), you need to update the hierarchical block's information so that it can locate the renamed libraries. The easiest way to do this is to drag hierarchical blocks from their libraries, place them on a worksheet, and update their structure, as discussed below.

Note This is only required for hierarchical blocks saved in libraries; hierarchical blocks saved only in a model get updated with the model.

When you add a hierarchical block from a library to a model worksheet, the hierarchical block causes Extend to open the libraries of the blocks inside it. Since you have renamed those libraries, Extend will not be able to locate them. In this case, Extend will ask you to find and open the correct libraries. Note: keeping all your libraries in the *Libraries* folder will make this search process easier.

If you save the model worksheet that contains the hierarchical block, the location of the renamed libraries is saved for the model only. Before you close the model worksheet, you also need to update the hierarchical block's library information. To do this, open the hierarchical block's *structure window* and then close it, causing the hierarchical block Save Dialog to appear. In the dialog, choose "Also Save to Library". This process is described in "Results of modifying hierarchical blocks" on page E263 in the Extend User's Guide.

Libraries

The libraries that come with your Extend package are already formatted correctly for your operating system. However, if you build your own libraries, and want to transfer them to a computer running a different operating system, you must convert them to the appropriate operating system format. You do this **on the Macintosh computer** using the Extend conversion utility, MacWin Converter. This utility converts libraries to the specified operating system format, ensures that the file names are properly formatted, and so forth.

After the libraries have been converted to Windows or Macintosh format, physically transfer them to the target computer (that is, keep them on the Macintosh or transfer them to a Windows computer). Then recompile the libraries under the target computer's operating system using Extend's Compile Open Libraries command.

The MacWin Converter converts libraries to either Macintosh or Windows format. After conversion, you must recompile the library on the target computer. When you do this, the library will compile to native code for the target system. For example, if you compile on a Windows computer, the library will be in native Windows mode. If you compile on a Power Macintosh computer, the library will be in native Power Macintosh mode.

Blocks that use the equation functions

If you build blocks that use the equation functions, your code needs to detect if the model is being opened on a different platform. See "Equations" on page P96 for more information.

Extensions and drivers

Extensions are files (such as pictures and DLLs) that can be accessed by Extend to fulfill specialized tasks. Drivers (*Macintosh only)* are code segments that allow you to access external devices and functions. Like libraries, the extensions that come with your Extend package are formatted correctly for your operating system. However, if you build your own extensions or use Extend's driver calls, and you want to transfer your extensions or blocks to a computer running a different operating system, you will need to do some conversion:

- *Pictures:* As discussed in "Picture and movie files" on page P241, Extend for Windows accepts three kinds of pictures: .WMF (Windows MetaFiles), BMP (Bitmap), and a Macintosh picture resource file that has been converted to Windows format. Extend for the Macintosh accepts only picture resources. To convert Macintosh picture resource files to Windows format, use Extend's MacWin Converter utility on the Macintosh. To convert Windows pictures to Macintosh format, use a graphics conversion application.
- *Movies:* When you create a QuickTime movie file, save it in multi-platform format. You can then move the movie file between operating systems without having to do any conversion. Note that Extend for Windows requires the 32-bit compatible version of QuickTime for Windows.
- *Sounds:* Shareware utilities are available to convert Macintosh sound resources (SNDs) to Windows sound files (.WAV) and vice versa.
- *DLLs, XCMDs, and XFCNs:* If you use either the DLLXCMD, DLLParam, DLLArray (*Windows*) or XCMD, XCMDPARAM, and XCMDARRAY (*Macintosh)* functions in your blocks, the block script will not need to be rewritten when you transfer blocks between Macintosh and Windows systems. However, the extensions will have to be rebuilt on the platform to which you are transferring them. These extensions are discussed fully in "XCMDs and XFCNs" on page P237. If your blocks call DLLs, XCMDs, or XFCNs, their code needs to detect if they are being used on a different platform, as discussed in the Note, below.
- If you build blocks that use the driver functions, your code needs to detect if the blocks are being used on a different platform, as discussed in the Note, below.

For more information, see "Extensions" on page P236 or "Other drivers (Macintosh only)" on page P116.

Note The following ModL constants return TRUE or FALSE depending on the platform: PLATFORMWINDOWS; PLATFORMPOWERPC. If your code calls XCMDs, DLLs, or other drivers, use these constants in an "if" statement to make the code be cross-platform capable. For example:

```
if (PLATFORMWINDOWS)
     use DLL TypeLanguage calls;
else if (PLATFORMPOWERPC)
     use Driver calls;
```

Appendix E: The SDI Database API

A detailed description of the SDI Database API, including all of the methods that can be used

SDI Database API Reference

This is a reference for users of SDI Industry Developer, which is an add-on to Extend. Developer includes SDI Industry and the SDI Database API (Application Programming Interface). The SDI Database API licenses users to develop SDI Database-aware blocks in Extend using ModL.

What is the SDI Database?

The SDI Database is the key element that SDI Industry adds to Extend. The SDI Database is a complete system for managing simulation model data. It is designed from the ground up as a flexible tool for simulation modelers, and it is completely integrated with Extend.

The system includes features in both Extend and Excel. Databases can be built in Excel and imported into Extend, or vice versa. The Excel interface is optional since complete databases can be built and managed within Extend.

Industry includes database-aware blocks, such as DB Lookup, DB Matrix Lookup, DB 3D Lookup, DB Write, DB Matrix Write, DB Extract, DB Inject, and DB Specs. These blocks allow the database to be used effectively without the need for programming. For general information on the SDI Database, refer to the Extend User's Guide section on SDI Industry or the SDI Industry on-line Help.

The Extend Embedded Database

A database is created in Extend by placing a Database Manager block onto an Extend worksheet. The Database Manager houses an embedded Active-X control, which stores multiple tables of data. These tables are accessed throughout the model by any database-aware block. The database is "embedded" because the data is saved inside the model. When a model containing an embedded SDI Database is opened, the data is loaded into memory.

Design Criteria for the Embedded Database

The embedded database is not meant to be a general-purpose database transaction and reporting tool. Instead, it is intended as a tool for simulation modelers. The following criteria have guided the development of the SDI Database:

- Access speed and write speed
- Low processing overhead (as compared to a transaction tool such as Oracle)
- Simplicity
- Granular Data Support (built-in randoms)
- Portability (easy import/export)

What is the SDI Database API?

The SDI Database API provides a standard set of methods (procedures and functions) for making an Extend block database-aware. The SDI Database API is comprised of ModL include files (header files) that can be referenced by any ModL block code. The API simplifies the creation and maintenance of database-aware blocks. All of the database-aware blocks within Industry are written using the methods from the SDI Database API.

The SDI Database API provides over 40 methods for accessing and restructuring the SDI Database. The API methods provide fast random access to the SDI Database and enable dynamic creation or deletion of records, fields and tables. In addition, methods are provided for user selection of table, field, or record via pop-up menus. The SDI Database API makes use of dynamically linked libraries (DLLs) in order to reduce the compilation size of database aware blocks.

How the SDI Database API is Used

The following are some ways that the API is used:

- General Data Access: Model-wide access to equipment and product data.
- Experimentation: Read/write multiple scenario data.
- Reporting: Create specialized report tables.
- Schedule Generation: Generate SDI Industry-format schedules before or during model execution.
- Schedule Coordination: Housing and coordination of schedules.

Tutorial 1: Getting Started

This section assumes that you want to create a database-aware block to access data from existing tables in a model database. While it is possible to create your tables using the API, we will leave that as an exercise for the user. To start this tutorial, do the following.

1. Have Some Tables to Work With

You can create some tables using wizards, or you can import tables from a DB Text file. Tofollow along with the subsequent examples, drop a Database Manager onto a new worksheet, and import the DB Text File associated with the packaging line demo. This DB Text file is located in: Extend5\SDI Industry\Packaging Line Demo\Packaging Line.txt

2. Ensure that Required Files are Installed

Ensure that SDI Industry and the SDI Database API Includes are properly installed. The include files are located in Extend5\Extensions\SDI\Includes. The three files required for this example are SdiDlls.h, UtilityAPI.h, and DBAPI.h

Tutorial 2: How To Create a Database-Aware Block

This section explains the required minimum steps for creating a database-aware block. Create a new block, and in the structure of the block do the following:

1. Make Dialog Changes
Create a hidden checkbox with ModL name of "DBBlock".

2. Enter Include Statements
In the code for this block, after the initial variables, insert the following statements:

```
#include "SDI\Includes\SdiDLLs.h";     ** DLL Loading routines.
#include "SDI\Includes\UtilityAPI.h"; ** Commonly-used non-DB functions
#include "SDI\Includes\dbAPI.h";      ** Database functions.
```

3. Connect to the Database
In the block's CreateBlock event handler, insert the following statements:

```
DBBlock= 1;
GetDatabaseFunctions();
```

In the block's PasteBlock event handler, insert the following statement:

```
GetDatabaseFunctions();
```

In the block's OpenModel event handler, insert the following statement:

```
GetDatabaseFunctions();
```

These statements ensure that your block obtains access to database methods that will be called subsequent to block creation, or opening of a model containing the block.

Accessing Table Data

In the examples below, we will access the tables from the packaging line demo database. The variables thisTable, thisField, and thisRecord are all integers used to store the table number, field number, and record number, respectively.

1. Get Table Number By Name

```
thisTable= LookupTableNumber("Packaging Line Rates",TRUE); ** True= give
                            ** error and abort if not found.
```

If you choose not to abort if the table is not found, the value of thisTable is -1.

2. Get Field Number by Name

```
thisField= LookupFieldNumber(thisTable,"Rate",TRUE);** True= give error
                                                ** and abort if not found.
```

A field number corresponds exactly to the number of the field. In the Database Viewer, the left-most field shown is field number 1.

3. Get the Data

Use the functions GetDataR, GetDataS, and GetDataI for field types of real, string, and integer, respectively. For example:

```
ThisReal= GetDataR(thisTable,thisField,thisRecord); ** Returns a random if
                                         ** cell has a random definition.

ThisString= GetDataS(thisTable,thisField,thisRecord);
ThisInteger= GetDataI(thisTable,thisField,thisRecord);
```

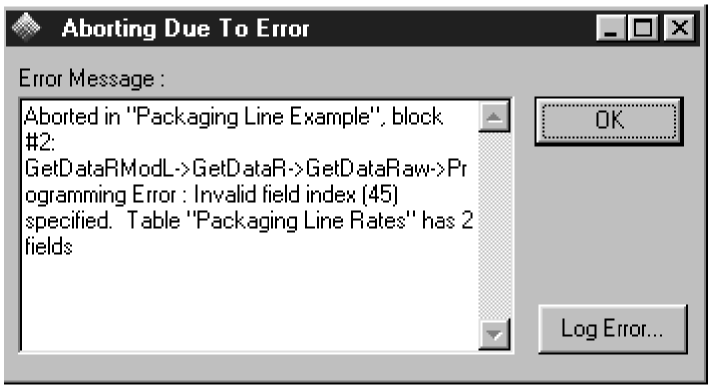
To write to these cells, refer to the PutData procedures.

Database API Error System

The SDI DB error-reporting system is integrated with Extend. When invalid conditions or other errors are detected in the SDI Database API, an error message is stored in the database. This error message is displayed by Extend, and Extend aborts naturally. Given that the Packaging Line database is in use, the following is an example of code that will cause an error:

```
ThisField = 45;
GetDataR(ThisTable,ThisField,ThisRecord); ** Packaging Line Rates has only
                                        ** 2 fields!
```

On execution, the following dialog is displayed:



```
Aborted in "Packaging Line Example", block
#2:
GetDataRModL->GetDataR->GetDataRaw->Pr
ogramming Error : Invalid field index (45)
specified.  Table "Packaging Line Rates" has 2
fields
```

Extend will subsequently abort normally.

Useful Techniques:

Iterating Through All Tables in a Database.

```
ThisTable= GetFirstTable(); ** This is the table number of the first table.

ThisNumberOfTables= GetNumberOfTables();

for ThisCounter = 1 to ThisNumberOfTables
      {
      ** Use ThisTable to do what you want; e.g.,
      ThisTableRecords= GetTableRecords(ThisTable);
      ** Now increment to next table
      ThisTable = GetTableNextTable(ThisTable);
      ** Gets the next table number in alphabetical order.
      }
```

Iterating Through All Fields in a Table

Once you have the table number of a given table you can traverse its fields. To traverse the fields of a given table, do the following:

```
integer NumFields, ThisField;
string ThisFieldName;

ThisTable= LookupTableNumber("My Table",TRUE);** Start with some table.
NumFields=  GetTableFields(ThisTable);
for ThisField= 1 to NumFields
      {
      ** Use ThisTable and ThisField to do what you want; e.g.,
      ThisFieldName= GetFieldName(ThisTable, ThisField)
      }
```

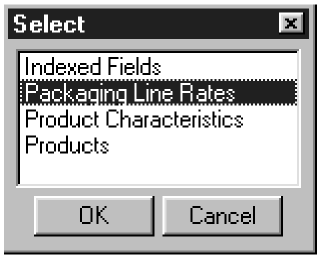
Selecting Tables, Fields, and Records

It is common in database-aware block dialogs to have popup selections for tables, fields and records. In your dialog, you should create dialog items of editable text for the table and field names; e.g., TableName and FieldName. You should also create buttons to fire the selection menus; e.g., SelectTableButton and SelectFieldButton.

Selecting a table:

```
on SelectTableButton

      {
      integer NewTable;
      newTable = SelectTableMenu(0,FALSE);  ** Select the table from a menu.
      if (newTable > 0)
            TableName= GetTableName(NewTable);
      }
```

The following shows the dialog which results from the *SelectTableMenu* call above:



The following shows how to select the field for a given table.

We assume that we have *Table* set to the selected table name now.

```
on SelectFieldButton
{
     integer NewField;

     newField = SelectFieldMenu(Table,1, FALSE,** Select the field
                                                ** from a menu.
     if (newField > 0)
          FieldName= GetFieldName(Table, NewField);
}
```

Additional Database API Functions

There are many other methods which are provided in the Database API. The additional methods are characterized as follows:

- Excel Date-Time Handling and Selection Windows
- Database Import/Export Methods
- Wizards for Table Manipulation

Please check the web page and on-line help for up-to-date help on these methods at:

http://www.simulationdynamics.com/IndustryDeveloper/Updates.htm

Parameters to the API Methods

In many cases a Database API method requires a parameter for the table, field or record number of the data to be accessed. If the table number parameter is not valid, an exception will occur stating that the table was not found in the directory. If the field number is out of bounds for the table, an exception will occur stating that the field index is out of bounds. If the record number is out of bounds for the table, an exception will occur stating that the record index is out of bounds.

Reference for All Methods

SDI Database API Method	Description	Returns
integer AddField (integer Table, string FieldName, integer FieldType, integer Access, integer Format)	Appends a field with the attributes specified by the parameters to *Table*. Note that field names must be unique. See *SetFieldAccess* for access codes. See *SetFieldFormat* for format codes. This method is used for existing tables. For new tables make one or more *DefineField* statements followed by a *NewFields* statement.	I
string CellLocationName(integer Table, integer Field, integer Record)	Returns the cell at the specified *Table, Field,* and *Record*.	S
procedure ChangeNumberofRecords (integer Table, integer numberofRecords)	Adds or removes records to or from the end of the specified *Table*, such that it will have the specified number of *Records*.	N/A
procedure DefineField(integer Field, string FieldName, integer FieldType, integer Access, integer Format)	Caches a field definition for later addition to a table. When creating a new table, one or more *DefineField* statements are followed by a *NewFields* statement. Note that field names on a table must be unique. See *GetFieldType* for valid *FieldType* codes. See *SetFieldAccess* for valid *Access* codes. See *SetFieldFormat* for valid *Format* codes.	N/A
procedure DeleteDatabase()	Removes all the tables from the database.	N/A
procedure DeleteField(integer Table, integer Field)	Removes the indicated *Field* from the *Table*.	N/A
procedure DeleteRecords(integer Table, integer NumberOfRecords, integer DeleteFrom)	Removes a number of records specified by *Records* from *Table*. The first record to delete is indicated by *DeleteFrom*. If *NumberOfRecords* is greater than or equal to the number of records subsequent to *DeleteFrom*, all the records following *DeleteFrom* are removed from the table.	N/A

SDI Database API Method	Description	Returns
procedure DeleteTable(integer Table)	Deletes *Table* from the database.	N/A
procedure DisplayTable(string TableName)	Displays the SDI Database Viewer PlugIn, with focus on the table named *TableName*.	N/A
integer FindIndex (integer Table, integer Field, string StringToFind, integer ExactMatch)	Returns the record number of the first occurrence of the *StringToFind* parameter in the specified *Table* and *Field*. If the *ExactMatch* parameter is FALSE, partial substring matches will also be returned.	I
integer FindIndexRecursive(integer Table, integer Field, string SearchText, integer ExactMatch)	Similar to *FindIndex*. If a “Child” field is specified, then the result is the first record number in the child field whose value refers to the *SearchText* value in the parent lookup field.	I
string GetData (integer Table, integer Field, integer Record)	Returns a string that contains automatically converted data from the cell referenced by the specified *Table, Field,* and *Record*. For example using *GetData* to access a Real type field will return a string that contains the real value. See also *GetDataFormatted*.	S
string GetDataFormatted(integer Table, integer Field,integer Record)	Returns a string containing the value from the cell referenced by the specified *Table, Field,* and *Record*. The data is automatically converted to a formatted string using the field’s format code.	S
integer GetDataI (integer Table, integer Field, integer Record)	Returns data from the cell referenced by the specified *Table, Field,* and *Record*. The data type of the field must be integer.	I
real GetDataR (integer Table, integer Field, integer Record)	Returns data from the cell referenced by the specified *Table, Field,* and *Record*. The data type of the field must be real. If the cell contains a random definition, a random value is returned.	R
string GetDataRecursive(integer Table, integer Field, integer Record)	Returns a string that contains automatically converted data from the cell referenced by the specified *Table, Field,* and *Record*. For “Child” type fields, this method performs the lookup and returns the parent field value referred to by the index value.	S

SDI Database API Method	Description	Returns
string GetDataS (integer Table, integer Field, integer Record)	Returns data from the cell referenced by the specified *Table, Field,* and *Record*. The data type of the field must be string.	S
integer GetDBMgr-BlockNumber()	Returns the block number of the Database Manager block in the model. Returns -1 if the block is not found.	I
integer GetFieldAc-cess(integer Table, integer Field)	Returns the field access code stored as a property of the specified *Field* of the *Table*. See *SetFieldAccess* for access codes.	I
integer GetFieldFor-mat(integer Table, integer Field)	Returns the format code stored as a property of the specified *Field* of the *Table*. See *SetFieldFormat* for format codes.	I
string GetField-Name(integerTable, integer Field)	Returns the name of the specified *Field* of the *Table*.	S
integer GetFieldPar-entField(integer Table, integer Field)	Returns the parent field number (lookup) for the specified *Field* of the *Table*. If the field is not a “Child” field, this method returns 0.	I
integer GetFieldPar-entTable (integer Table, integer Field)	Returns the parent table number (lookup) for the specified *Field* of the *Table*. If the field is not a “Child” field, this method returns 0.	I
integer GetFieldPar-entTypeIndex (inte-ger Table, integer Field)	Returns the field type code for the specified *Field* of the *Table*. If the field is not indexed by type, then this method returns 0.	I
integer GetField-Type(integer Table, integer Field)	Returns the field type code for the specified *Field* of the *Table*. Valid field types for database fields are: cTypeReal = a field that contains floating point values. cTypeString = a field that contains string values. cTypeInteger = a field that contains integer values. cTypeChild = a field that contains integer indexes to a lookup field in another table.	I
integer GetFirstTa-ble()	Returns the table number of the first table in alphabetical order of the tables available in the database. This function is primarily used in conjunction with the *GetTableNextTable*() function to cycle through all the tables in the database.	I

SDI Database API Method	Description	Returns
integer GetNumberOfTables()	Returns the total number of tables currently managed by the database.	I
string GetRandomDescription (integer Table, integer Field, integer Record)	Returns the “name” of the random distribution defined in the cell specified by *Table*, *Field*, and *Record*. If there is no random defined in the cell an error will occur.	S
integer GetTableAccess(integer Table)	Returns the access code for the specified *Table.* See *SetTableAccess* for valid table access codes	I
integer GetTableFields(integer Table)	Returns the number of fields for the specified *Table*.	I
procedure GetTableHelp(integer Table, string Help Array [2])	Returns the WinHelp parameters stored as properties of the specified *Table*. Return values are in the array *HelpArray.* Declare *HelpArray* as follows: **string HelpArray[2];** On return, HelpArray[0] will contain the WinHelp file name, and HelpArray[1] will contain the WinHelp topic string. All help files are located under the Extend5 Help directory.	N/A
string GetTableName(integer Table)	Returns the name of the specified *Table*.	S
integer GetTableNextTable(integer Table)	Returns the next table in alphabetical order for the specified *Table*. If the specified *Table* is the last table in alphabetical order, or if the table parameter is not in the directory, this function will return -1.	I
string GetTableNote(integer Table)	Returns the table note of the specified *Table*.	S
integer GetTableRecords(integer Table)	Returns the number of records in the specified *Table*.	I
procedure InsertField(integer Table, string FieldName, integer FieldType, integer Access, integer Format, integer Field)	Inserts a field into the specified *Table*. Note that field names must be unique. The field will be inserted before the field number specified by the specified *Field*. See *GetFieldType* for type codes. See *SetFieldAccess* for access codes. See *SetFieldFormat* for format codes.	N/A

SDI Database API Method	**Description**	**Returns**
integer IsData-baseAvailable()	Returns TRUE if the Database Manager for this model has been registered with the DBAPI library. This function is primarily used by Database-Aware blocks to verify that a valid Database Manager exists in the model.	I
integer Look-upFieldNumber (integer Table, string FieldName, integer AbortOn-NoField)	Returns the *FieldNumber* for the *FieldName* in the specified *Table*. If the field is not found in the table, and *AbortOnNoField* is TRUE, an exception will occur stating that the field with the given name was not found on the table. If the table is not found, and *AbortOn-NoField* is FALSE, the function returns -1.	I
integer LookupT-ableNumber (string TableName, integer AbortOnNoTable)	Returns the *TableNumber* for the specified *TableName.* If the table is not found in the directory, and the *AbortOnNoTable* is TRUE, an exception will occur stating that the table was not found in the directory. If the table is not found, and *AbortOnNoField* is FALSE, the function returns -1.	I
procedure New-Fields(integer Table, integer Number-OfFields)	Adds the specified *NumberOfFields* from the cache to the specified *Table*. Any and all remaining cached fields will no longer be avail-able after this method is called. This is used in conjunction with *DefineFields*.	N/A
procedure NewRecords (inte-ger Table, integer Number OfRecords, integer InsertAt)	Adds the specified number of records to the specified *Table*. The records are empty and will be inserted before the record specified by *InsertAt*.	N/A
integer NewTable (string TableName)	Creates a new table with the name specified, and returns the table number which is automatically assigned by the Database Manager. Note that table names must be unique.	I
procedure PutData (integer Table, integer Field, integer Record, string Value)	Generically assigns a value to the cell referenced by the specified *Table, Field,* and *Record*. The value parameter is automatically con-verted to the type of the field specified. If the data cannot be suc-cessfully converted to the target type, an exception will occur.	N/A

SDI Database API Method	Description	Returns
procedure Put-DataI(integer Table, integer Field, integer Record, integer Value)	Assigns integer data to the cell referenced by the specified *Table, Field,* and *Record*. If the field's data type is not integer, an exception will occur.	N/A
procedure PutDa-taR (integer Table, integer Field, inte-ger Record, real Value)	Assigns floating point data to the cell referenced by the specified *Table, Field,* and *Record*. If the Value parameter is above a threshold used for sentinel random values, an exception will occur. If the field's data type is not real, an exception will occur.	N/A
procedure PutDa-taS (integer Table, integer Field, integer Record, string Value)	Assigns string data to the cell referenced by the specified *Table, Field,* and *Record*. If the field's data type is not string, an exception will occur.	N/A
procedure PutRan-domR (integer Table, integer Field, integer Record, integer DistType, real Parm1, real Parm2, real Parm3)	Assigns a random distribution to the cell referenced by the spec-ified *Table, Field,* and *Record*. Valid *DistType* (distribution type) codes are defined as constants in the include file. They are as follows: dtNone; dtBeta; dtBinomial; dtEmpirical ; dtErlang ; dtExponential; dtGamma; dtGeometric; dtHyperExponential; dtUniformInteger; dtLogLogistic; dtLogNormal; dtNegBino-mial; dtNormal; dtPearsonV ; dtPearsonVI; dtPoisson; dtPrime; dtUniformReal; dtTriangular; dtWeibull; dtUserDefined;	N/A
integer RealStrin-gOrInteger(integer Table, integer Field)	This method differs from *GetFieldType* in that it will automatically convert a result of "Child" field type into an "Integer" type result. This method is used primarily to determine which method of data access to use: *GetDataR*, *GetDataS*, or *GetDataI*. Returns *cTypeReal*, *cTypeString*, or *cTypeInteger*, respectively. See *GetFieldType* for details.	I
integer SelectField-Menu(integer Table, integer DefaultSelected-Field, integer Show-BlankItem)	Displays a selection dialog containing the field names from the spec-ified *Table*, and returns the number of the field selected. If *Default-SelectedField* is greater than -1, then that field will be highlighted. Zero is only valid if *ShowBlankItem* is TRUE. If true, a blank will be selectable at the top of the menu and its return value is zero. *Show-BlankItem* is true/false.	I

SDI Database API Method	Description	Returns
integer SelectRecordMenu (integer Table , integer Field, integer DefaultSelected Record, integer ShowBlankItem)	Displays a selection dialog containing the values from the table and field specified by the parameters. If *DefaultSelectedRecord* is greater than -1, then that record will be highlighted. Arguments are similar to those for *SelectFieldMenu*.	I
integer SelectTableFromDirectory (integer Directory Table, integer Field, integer DefaultSelectedTable, integer Show BlankItem)	Displays a selection dialog containing the names of all the tables in the directory table. A directory table is a table where the first field contains a list of table names. The parameters determine the appearance and properties of the dialog. If *SelectedTable* is greater than -1, then that Table will be pre-selected. Zero is only valid if *ShowBlankItem* is TRUE. *ShowBlankItem* is true/false. If true, a blank will be selectable at the top of the menu and zero will be returned.	I
integer SelectTableMenu(integer DefaultSelectedTable, integer ShowBlankItem)	Displays a selection dialog containing the table names from all the tables in the database.*ShowBlankItem* is true/false.If *SelectedTable* is greater than -1, then that Table will be pre-selected. Zero is only valid if *ShowBlankItem* is TRUE. *ShowBlankItem* is true/false. If true, a blank will be selectable at the top of the menu and zero will be returned.	I

procedure SetFieldAccess(integer Table, integer Field, integer AccessCode)	Assigns the Field access code for the specified *Field* of the specified *Table*. Valid Field Access codes are defined as constants in the include file. They are as follows:
	cReadWrite = Read/Write
	cReadOnly = Read Only
		N/A

procedure SetFieldAsChild(integer Table, integer Field, integer ParentTable, integer ParentField)	Causes the specified *Field* of the specified *Table* to be a "lookup" or "child" field. *ParentTable* specifies the *Table* containing a list of possible values in the field specified by *ParentField.* The integer value of a record's "child" serves as an index to the *ParentTable.* The index is the record number of the *ParentTable* record which contains the desired value. If either table number parameter refers to a Table not present in the database, an exception will occur. If either field number is out of bounds for their respective tables, an exception will occur.	N/A

SDI Database API Method	Description	Returns

procedure SetField-Format(integer Table, integer Field, integer FormatCode)	Assigns the format code for the specified *Field* of the specified *Table*. Valid format codes are defined as constants in the include file. The meaning of "x' is: Add 0.1. 0.2, 0.3, ... to the specifier to define up to 9 decimals, e.g. (cCurrency + 0.2) gives $5.75. The format codes are as follows:
	
	cGeneralFmt = General - no special formatting
	cDecimalFmt = 0.x - decimal precision up to 9 places.
	cThousandsFmt = 0,000.x includes the thousands separator
	cCurrencyFmt = $0.x - currency
	cCurrencyThousandsFmt = $0,000.x -currency and thousands seperator.
	cPercentFmt = 0.x% - percent
	cDateFmt = Date
	cTimeFmt = Time
	cDateTimeFmt = Date and Time
		N/A

procedure SetField-Name(integerTable, integer Field, string lsNewName)	Assigns the field name for the specified *Field* of the specified *Table*. Note that a table's field names must be unique within a given table.	N/A
procedure SetField-Note(integer Table, integer Field, string Note)	Assigns the field note for the specified *Field* of the specified *Table*.	N/A

procedure SetTable-Access (integer Table, integer NewValue)	Assigns the access code to the specified *Table*. If the new access code is out of bounds, no changes will be made to the table. Valid Table Access codes are defined as constants in the include file. They are as follows:
	cUnlockedTableType = unlocked Table
	clockedTableType = locked table
		N/A

procedure SetTable-Name (integer Table, string NewValue)	Assigns a new name for the specified *Table*. Note that each table name must be unique.	N/A
procedure SetTable-Note (integer Table, string NewValue)	Assigns the note for the specified *Table*.	N/A

SDI Database API Method	**Description**	**Returns**
procedure TrimTable (integer Table, integer Field)	Automatically detects the last populated record in the table and field specified. Subsequent records are deleted from the specified table. A record is determined to be empty if it contains zero, or a blank string.	N/A
procedure SetTable-Help(integer Table, string WinHelpFileName, string Topic)	Creates an association between the table and a WinHelp file and topic. Table Help is accessed from the Help menu. All WinHelp files are assumed to be in the Extend5\Help subdirectory.	

Index