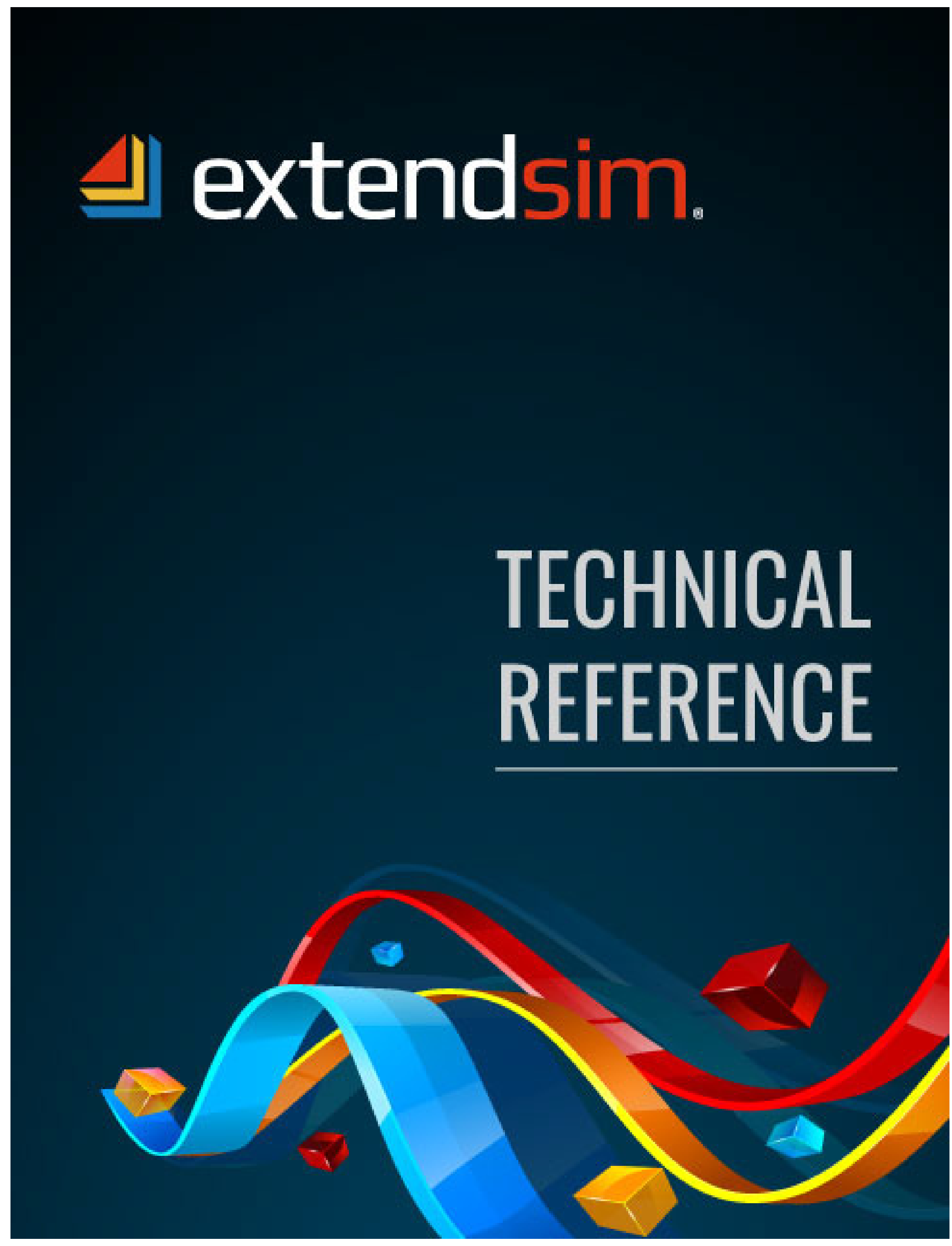

extendsim
TECHNICAL
REFERENCE

© 2024 ANDRITZ Inc. This program is protected by US and international copyright laws.

You may not copy, transmit, or translate all or any part of this document in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than your personal use without the prior and express written permission of ANDRITZ Inc.

License, Software Copyright, Trademark, and Other Information

The software described in this manual is furnished under a separate license and warranty agreement. The software may be used or copied only in accordance with the terms of that agreement. Please note the following:

> ExtendSim blocks and components (including but not limited to icons, dialogs, and block code) are copyright © by ANDRITZ Inc. and/or its Licensors. ExtendSim blocks and components contain proprietary and/or trademark information. If you build blocks, and you use all or any portion of the blocks from the ExtendSim libraries in your blocks, or you include those ExtendSim blocks (or any of the code from those blocks) in your libraries, your right to sell, give away, or otherwise distribute your blocks and libraries is limited. In that case, you may only sell, give, or distribute such a block or library if the recipient has a valid license for the ExtendSim product from which you have derived your block(s) or block code. For more information, contact ANDRITZ at Info.ExtendSim@Andritz.com or Support.ExtendSim@Andritz.com.

© 2024 ANDRITZ Inc. This program is protected by US and international copyright laws. Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation. The copyright for Stat::Fit® is owned by Geer Mountain Software. All other product names used in this manual are the trademarks of their respective owners. All other ExtendSim products and portions of products are copyright by ANDRITZ Inc. All right, title and interest, including, without limitation, all copyrights in the Software shall at all times remain the property of ANDRITZ Inc. or its Licensors.

Acknowledgments

Extend was created in 1987 by Bob Diamond; it was re-branded as ExtendSim in 2007.

The contents of this document are the result of years of work by software architects, simulation engineers, and technical writers and editors of ExtendSim products.

ANDRITZ Inc • 13560 Morris Road, Suite 1250 • Alpharetta, GA 30004 USA
770.640.2500 • Info.ExtendSim@Andritz.com
www.ExtendSim.com

Table of Contents

TABLE OF CONTENTS

TECHNICAL OVERVIEW

Introduction 1
- About the Technical Reference 2
- Additional resources 3

Parts of a Block 5
- The Example block 6
- The block's user interface—its dialog 6
- The block's structure 6
- Overview of block parts, by tab and pane 7
- Icon tab 9
- Script tab 11
- Dialog tab 13
- Help tab 14
- Dialog items 14
- Connectors 21

ModL Overview 25
- ModL feature overview 26
- ModL compared to other programming languages 27
- ModL language terminology 30
- Differences between equation blocks and programmed blocks 31
- Structure of a block's ModL code 32
- Accessing connectors from a block's code 34
- Accessing dialog items from a block's code 35
- Accessing code from other languages 42
- External source code 42

TUTORIAL

Creating a Block 43
- Building a simple block that converts miles to feet 44
- Adding user interaction and display features 48
- Adding an intermediate results feature 51
- Adding 2D animation 53
- Other features you might have used 54
- Defining functions 56

INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

The ModL Language 59
- Names 60
- Data types: definitions and declarations 60
- Scope of global, local, and static variables 62
- Constant definitions 62
- Constants that are pre-defined 63

BLANK and NoValue 63
Numeric type conversion 64
Arrays 65
Array-like structures 67
Operators 68
Control statements and loops 70
User-defined functions 72
Message handlers 75
System variables 76
Global variables 76
Conditional compilation 76

Programming Tools 77
Script Editor 78
Debugging and profiling 81
Include files 81
Conditional compilation 82
External source code control 83
Extensions 85
DLLs 86
Sounds 89
Picture and movie files 90
Protecting libraries 90

Programming Techniques 93
Data source indexing and organization 94
Equation block programs 94
Working with dialogs 95
Remote access to dialog variables 99
Working with connectors 102
Working with arrays 103
Working with linked lists 111
Using message handlers 111
Working with databases 113
Reading text blocks as commands 115
Changing data while the simulation is running 117
Scripting 118
OLE and ActiveX Automation 119

Animation Using ModL 127
2D animation 128

Simulation Architecture 137
Running a simulation 138
How discrete event blocks and models work 146
Globals in discrete event blocks 160
Creating blocks for discrete event models 165
How discrete rate blocks and models work 165
Globals in discrete rate blocks 165
Globals for ARM (Advanced Resource Management) 167
Other reserved global variables 167

Debugging ... 169
Debugging models ... 170
Profiling ... 170
Debugging block code without the Source Code Debugger ... 171
Source Code Debugger ... 172
Debugger tutorial ... 172
Source code debugger reference ... 183

Variables, Messages, & Functions

ModL Variables ... 189
System variables ... 190
Global variables ... 191

Messages and Message Handlers ... 193
Summary of messages ... 194
Simulation messages ... 194
Model Status messages ... 196
Block Status messages ... 197
Dialog messages ... 199
Connector messages ... 201
Block to block messages ... 202
Dynamic Link messages ... 203
OLE messages ... 203

ModL Functions ... 205
ModL function overview ... 206
Math functions ... 207
I/O functions ... 222
Animation ... 250
Blocks and inter-block communications ... 256
Models, notebooks, and libraries ... 293
Scripting ... 300
Reporting ... 308
Plotting/Charts ... 308
Database functions ... 318
Arrays, pointers, queues, delay, linked list, and string lookup table functions ... 340
Miscellaneous functions ... 358
User-defined functions for ADO ... 371

Appendix

Menu Command Numbers ... 375

Upper Limits ... 379

ASCII Table ... 383

Index

Technical Overview

Introduction

Where you learn where you need to begin

"'Begin at the beginning,' the King said, gravely,
'and go 'til you come to the end; then stop.'"
— Lewis Carroll

About the Technical Reference

The purpose of the Technical Reference is to reveal the ExtendSim integrated development environment (IDE) so that you can:

- Use equations, logic statements, and function calls in equation-based blocks
- Create new blocks or modify copies of ExtendSim blocks
- Interface with ExtendSim using external code such as DLL's or VBA
- Build and control models using scripting

This manual is a companion to the ExtendSim User Reference which is a guide for those building models.

ExtendSim IDE

The Technical Reference reveals the ExtendSim IDE, which has several components:

- The ModL language, a variation of C++ that has been customized for simulation
- An equation editor for creating logic statements, formulas, and programming code in equation blocks
- A script editor for modifying or creating custom-programmed blocks that are fully integrated with ExtendSim
- A built-in compiler so the custom blocks you create are compiled into machine language and saved in libraries. The blocks that ship with ExtendSim, and the blocks that you created, are pre-compiled for use in models.
- A graphical user interface for building custom blocks, including an icon builder and a dialog editing environment so you can modify or create custom dialogs
- Programming tools, such as include files and external source code capability, that support and simplify your programming efforts
- A built-in source-code debugger that helps to locate errors in the blocks or equations you create

☞ The Technical Reference presumes that you know how to create models in ExtendSim. See one of the ExtendSim Quick Start Guides for model-building information.

How the Technical Reference is organized

The Technical Reference is organized into several sections:

- Overview
    - “Parts of a Block” (which starts on page 5) describes the internal parts (structure) of blocks. This chapter is also helpful for non-developers, to understand how the blocks in their models work.
    - “ModL Overview” (starting on page 27) gives an overview of the “action” part of a block – the ModL code. If you use equation-based blocks — Equation and Optimizer blocks (Value library), Equation(I) and Queue Equation blocks (Item library), or the Buttons block (Utilities library) — you will also find this chapter helpful.
- Tutorial
    - The “Creating a Block” chapter provides a tutorial on how to build a new block; the chapter starts on page 44.

- Integrated Development Environment
  - “The ModL Language” on page 59 describes ModL’s structure and constructs in detail. It presumes you have read the chapters in the Overview module.
  - “Programming Tools” describes all the tools, such as include files and external source code control, that ExtendSim provides to help you program efficiently; it starts on page 81.
  - “Programming Techniques” gives procedures and suggestions for programming using ModL and for interfacing ExtendSim with other languages, such as VBA. That chapter starts on page 93.
  - “Animation Using ModL”, starting onpage 127, describes the ExtendSim 2D capability and features.
  - “Simulation Architecture” gives important information about how ExtendSim runs simulations and how discrete event and discrete rate models and blocks work. The chapter starts on page 137.
  - “Debugging”, which begins on page 170, discusses model and code debugging and shows how to use the ExtendSim source code debugger.
- Reference

  Starting on page 190 are three chapters that list and discuss all of the ExtendSim variables, messages, and functions.

- Appendices
  - Appendix A: upper limit values for ExtendSim
  - Appendix B: an ASCII table
  - Appendix C: menu commands and numbers for the ExecuteMenuCommand function

As you will see, programming blocks in ExtendSim is quite easy, especially if you have ever programmed in any language.

Additional resources

ANDRITZ provides several resources to support your simulation experience.

1) Getting Started. The Getting Started model open when you launch ExtendSim. Use this interface to explore sample models and to view tutorials on building and running models.
2) Online help:
   - Access the electronic User Reference and Technical Reference by giving the command Help > ExtendSim Help or press F1 on your keyboard.
   - Use tooltips to identify interface elements.
   - Get complete definition of how a block works, including descriptions of its dialog items and connectors, by clicking a block’s Help button.
3) Web advice. FAQs are available at www.ExtendSim.com/Support.
4) Networking. Links for ExtendSim user forums, networks, and blogs are at www.ExtendSim.com. For example, the ExtendSim Exchange is a user forum for sharing ideas, insights, and modeling techniques with other ExtendSim users. Use this forum to post

issues and solutions, share blocks and models, and to talk directly to other people developing simulations. You must register to join, but access is free and available to all ExtendSim modelers.

5) Complimentary support. Get technical assistance for installation issues, basic usage questions, and troubleshooting for the first year after purchasing a new product or upgrade.

Contacting Technical Support

You must be registered to receive technical support. Support is complimentary for the first year after purchase. After that, you must either subscribe to the ExtendSim Maintenance Plan or purchase per-incident support.

To contact our support representatives, go to our website at www.ExtendSim.com, click the Contact Us link, and fill out a support ticket.

Be sure you are using the current version of ExtendSim. Software updates are available at our web site (www.extendsim.com). Upgrades to newer versions may be purchased separately if your license is not covered by a support plan.

Overview

Parts of a Block

An introduction to the internals of a block

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

This chapter discusses a typical ExtendSim block's internal structure—its icon, dialog, code, and more.

☞ The Technical Reference assumes you know how to build models since many of the concepts used when building blocks assume an understanding of how blocks are used in models.

The Example block

The Example block is a good source for investigating the dialog and internals of a block and will be used throughout this chapter. It is a copy of the Simulation Variable block (Value library) that ships with ExtendSim.

Any changes you make to a block in an ExtendSim library will affect that block in all existing models and will get discarded whenever the library is updated. Thus, it is important that you don't make any changes to the blocks that are shipped with your ExtendSim product. **Instead, make a copy of the block and save it with a new name and in a new library, as was done for this example.**

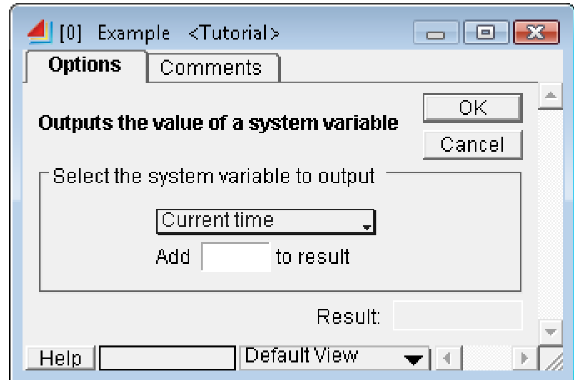
- Open a new model worksheet.
- Open the **Tutorial library** located at ExtendSim/Documents/Libraries/**Example Libraries**.
- From the Tutorial library's library window, select the **Example** block and place it on the model worksheet.

The block's user interface—its dialog

Every block has a dialog, which may be as simple as just the OK and Cancel buttons or it may be quite complex. A block's dialog is its user interface—that is what modelers see when they double-click the block's icon.

Dialog of the Example block

- Double-click the icon of the Example block. This opens its dialog, revealing two tabs:
  - An Options tab dealing with the system variables
  - A Comments tab for user comments



The frame, text, buttons, and entry boxes that are seen in the tabs are collectively known as *dialog items*.

In each block's dialog, a *Help* button, *block label* field, and *View* popup are located at the bottom of the dialog and to the left of the scroll bar, if there is one.

The block's structure

A block's internal structure is where the user-interface and block behavior is created and where it's icon, ModL code, dialog items, and other block components are defined. ExtendSim has built-in editing tools for creating or modifying blocks.

How to open a block's internal structure

There are three ways to access an existing block's internal structure:

1) Select the block by clicking once on its icon in a model worksheet. Then choose Develop> Open Block Structure.
2) Right-click the block's icon in a model worksheet or in the library window. Then select Open Block Structure.
3) Double-click the block's icon on a model worksheet or the library window while holding down the Alt (Windows) or Option (Mac OS) key.

Examining the internal structure of the Example block

▸ Open the internal structure for the Example block using one of the methods from above.

The block's Script tab opens in front by default.

- On the left side are tabs for *Icon*, *Script*, *Dialog*, and *Help*.
- For each of these tabs:
  - The title bar displays the name of the block and the name of the library the block resides in
  - There are panes for *Connectors* and *Dialog Item Names* on the right

Overview of block parts, by tab and pane

A block is composed of many parts, which are created in the tabs of the block's structure window. These parts comprise a block's internal structure and are interconnected. For example, the code reads information from the connectors, the help text is displayed through the dialog, and so on. All of the block's parts can be controlled by ModL code.

The parts are listed below, by tab, and discussed in more detail later in this chapter.

Script tab

This tab is where the block's code is entered. The code is written in ModL, the ExtendSim programming language. ModL is what makes the block work and determines how it behaves.

Part	Description	Pages
ModL code	ModL code can read and write information via the connectors, dialog, ExtendSim database, the model environment, and other blocks, as well as control all the parts of a block. This pane is also where you enter headings about the block and comments about the code.	11
Functions	This pane lists the message handlers (in all caps) and functions used in the block.	12
Includes	This pane lists and opens the includes that are used in the code.	12

Icon tab

The Icon tab is for creating the icon and everything else in its environment. The icon is the visual representation of the block as seen in a model. Icons can display 2D animation and most often have connectors that facilitate the exchange of data between blocks.

Overview

Part	Description	Pages
Icon	Icons are created using the ExtendSim drawing environment or a painting program, or by copying/pasting clip art.	9
2D Animation	Objects for 2D animation are part of the icon environment but can display anywhere in the model, even outside the icon's footprint. 2D animation can show motion, levels, and values, and can alter an icon's static look.	10
Connec-tors	Connectors appear in the model as part of the icon. They are used to transmit information to and from each block's ModL code. Connectors can be normal (single) or variable (a row of single connectors). ☞ Blocks can also transmit information without using connec-tors by sending messages to other blocks and by using global variables, global arrays, or an ExtendSim database.	21
Icon views	Icons can have one or more *views* such as a Forward and a Reverse.	9
Show icon positioner	Used if connection lines are out of alignment when a model built in ExtendSim 9 or earlier is opened in ExtendSim 10+.	10

Dialog tab

The Dialog tab is where the block's dialog is created and, if wanted, tabs for grouping dialog items. The dialog is the window that appears when you double-click a block's icon.

Part	Description	Pages
Dialog items	When creating a block, you specify the frames, text, buttons, tables, and entry boxes that go into the dialog (collectively known as *dialog items*).	14
Tabs	You can place all the dialog items in one dialog, or separate them into functional groupings using *tabs*.	13
Dialog Resizer	Allows you to set the default bottom and right-side edges for each tab separately.	13

Help tab

Part	Description	Pages
Help text	The text that appears when you click the Help button to the left of the dialog's bottom scroll bar. When the Help dialog is open, buttons at the bottom can be used to find a block, or display information about blocks, in all open libraries.	14

Connectors pane

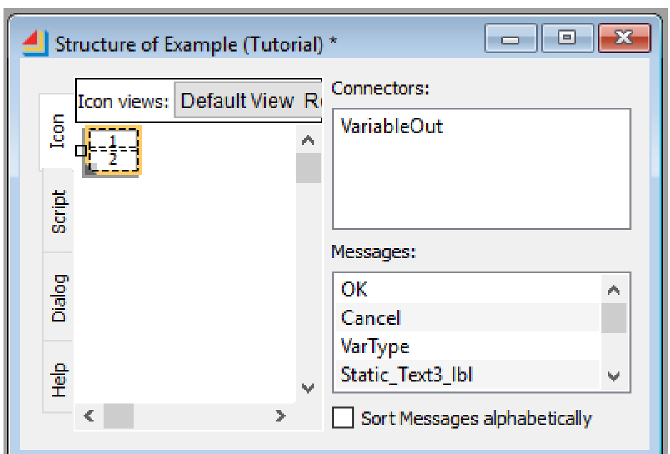
This pane lists the names of the block's connectors, if any. It is also used to edit the default connector names so they can more closely represent their function. See "Editing connector names" on page 22.

Dialog Item Names pane

This pane lists the names defined for the block's dialog items. The name of a dialog item is sent to the block code as a message when the dialog item is activated. By default, these names are listed in order created; they can also be sorted alphabetically.

Icon tab

The Icon tab of the Example block's structure is shown at right.



The area at the left of the Icon tab is used to enter and edit the icon(s) for the block and to add animation objects, block connectors, and icon views.

The Example block's icon has one input connector on the left of its icon and two animation objects, numbered 1 and 2.

☞ The panes on the right of the Icon tab are for Connectors and Dialog Item Names as discussed on page 8.

Icon

A block's icon is its most obvious aspect since it appears on the model worksheet. An icon consists of a drawing or group of drawn objects, the block's connectors, and possibly one or more animation objects.

A block can have more than one icon; which one is shown is determined by which view is selected, as discussed in "Icon views" on page 9.

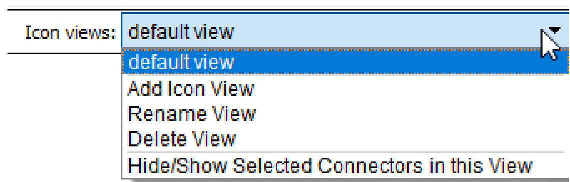
There are two ways to create an icon:

1) With the ExtendSim drawing tools, as discussed in the User Reference. The Shapes and Alignment tools provide a drawing environment for quickly creating icons.
2) By pasting drawings from the Clipboard. You can use a painting or drawing program to create an icon. Then copy and paste it into the Icon tab.

Use the same tools for selecting objects in the icon pane as you would use to select graphic objects in the model worksheet or notebook.

Icon views

The *Icon views* popup is located at the top of the Icon tab. Each block has at least one icon, which is its default. It is also possible for blocks to display different icons. For example, a block could have a Forward view icon and a different icon with a Reverse view. Or



the block could show a different icon depending on the type of machine it represents or based on a selection in the block's dialog. ExtendSim accomplishes this by facilitating the creation of different views of a block's icon.

The modeler selects icon views by right-clicking on a block or using the Views menu to the right of the block's label in its dialog, as seen on page 6.

Developers give a block additional icons using the *Icon views* popup in the Icon tab. The first icon that is created becomes the default view. As each view is added, ExtendSim copies the current icon. This gives the developer something to start with instead of having to redraw all of the icon. The Rotate and Flip buttons in the Alignment toolbar facilitate creating a new view that represents a flow in a new direction.

The IconViewChange message is sent when the modeler changes the icon view or when a ModL function call from a block changes the icon view; see page 199. There are also several functions for managing icon views; see "Icon views" on page 293.

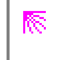
Views can be deleted or renamed at any time by using the commands in the Views popup menu. An example of a block with multiple views is the Select Value Out block (Value library).

☞ All views have the same number of connectors, but connectors can be selectively hidden in a view when it is created, or hidden and shown dynamically using ModL functions. ExtendSim automatically adds and deletes connectors from all views when the number of connectors on a block is changed.

Icon positioner

When models that were built prior to ExtendSim 10 are converted to ExtendSim 10 or later, the position of the model's blocks could be slightly different, causing connection lines to be unaligned. In those cases, the icon positioner can be used to adjust the relative location of the icon within the Icon tab so that the connection lines to the block are correctly aligned.

This adjustment has already been made in the current ExtendSim libraries, but you may want to use the positioner for any custom blocks that have been converted from earlier versions.



The *Show icon positioner* checkbox, located in the Icon tab of the block's structure, hides and shows the icon positioner, the pink icon shown here. By default, the icon positioner is located at the upper leftmost position of the icon's graphic items. The *Reset Icon Positioner* button resets the location of the icon positioner to the default.

To use the icon positioner, first determine approximately how many pixels and in what direction the icon needs to move so that the connection lines are aligned. Then in the Icon tab select the icon positioner and move it with the cursor or the keyboard arrow keys.

There is also a ModL function, blockAdjustPosition, that uses the location of the item positioner to shift the location of the block by the offset of the positioner location.

Connectors

See "Connectors" on page 21.

2D animation

In the Example block's icon shown above, the white rectangles at the center of the icon (labeled with the numbers 1 and 2) are *animation objects*. When the block is created, one or more animation objects can be added to the icon using the Animation Object button in the Icon toolbar.

The block's ModL code interacts with animation objects, causing them to show various behaviors. Depending on how the objects are coded, the display could change due to user interaction with the block (its dialog or connectors) and/or due to information received during the simulation run. For the Example block, the animation object for the Example displays as text which system variable has been selected in the block's dialog.

There are several ways to animate a block using 2D animation objects:

- Typically, the block's icon would be animated but it is also possible to animate outside the icon's footprint.
- Blocks can: show, hide, and change the 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.

The Item and Rate libraries use 2D animation extensively.

For more information about 2D animation

- The ExtendSim User Reference gives an overview of 2D animation, including block-to-block animation that flows along the connections in discrete event models.
- In this Technical Reference, the tutorial "Adding 2D animation" on page 53 shows how to add 2D animation objects to a custom block.
- The section "2D animation" on page 128 has a lot more information about programming 2D animation effects.
- ExtendSim has many functions that facilitate customizing animation depending on which icon views are being selected by the modeler. See the functions for "Icon views" on page 293.

Script tab

he Script tab of the Example block looks like the screenshot above. This tab is used to enter and edit the block's ModL code and comments. ModL is the internal programming language for ExtendSim.

Unlike the other block parts, which can be seen by the modeler, ModL code can only be accessed through a block's structure window or in an equation-based block's editor window.

☞ As shown on page 78 there is a Script dialog in the Edit > Options menu that is used to specify characteristics for a block structure's Script tab. For example, you can change the colors of user functions or keywords. When the block's Script tab is the active window, use Alt + O to open the Script dialog.

ModL code

If you know a programming language, you will probably recognize the structure of the ModL code. ModL is essentially C++ with some enhancements and extensions to make it more robust for simulation modeling.

☞ When you build blocks using ModL, the block's program is compiled to native machine code.

Layout

After the copyright and modification history information, the first lines are the declaration of the types of variables used in the code. Following is the code which is grouped into sections, where each section is either a *message handler* or a *function*. Lines that begin with "//", "**", or "/*" are comments.

For more information about the ModL language

- "ModL Overview" on page 25
- A tutorial starting on page 44
- "The ModL Language" on page 59

Functions

This popup lists the message handlers (in all caps) and functions used in the block.

Selecting something from the list causes the code to scroll to the section that uses the function or message handler. It also opens the include file if the function or message handler is used in an include.

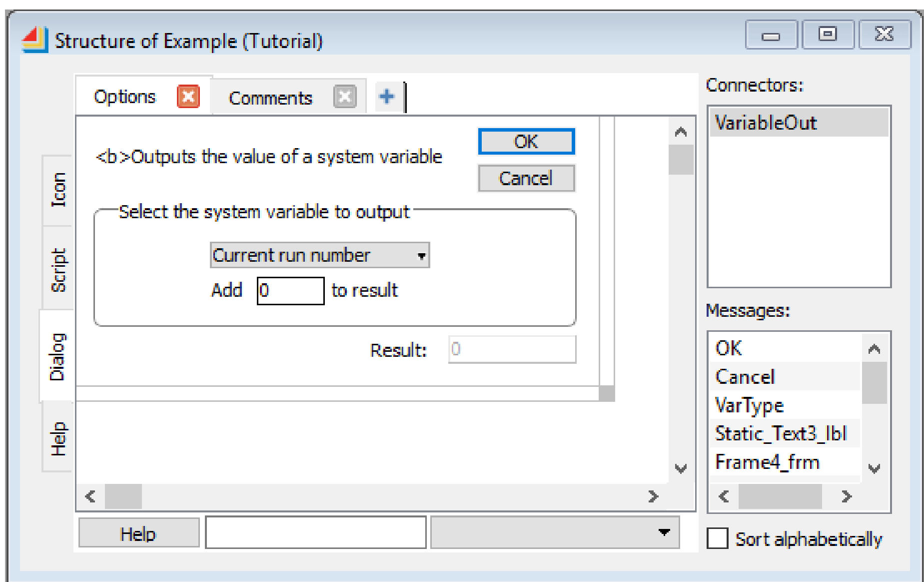
For more information

- "Messages and Message Handlers" on page 193
- "ModL Functions" on page 205.

Includes

Lists and opens the include files that are used in the code. Include files are standard header files that are put in ModL code and can contain ModL commands such as definitions, assignments, and functions. The purpose of an include file is to simplify maintenance when several blocks use similar variable definitions and functions. *Include files* are discussed on page 81.

Dialog tab



The Dialog tab for the Example block's structure looks like the screenshot shown here.

Dialog items

The buttons, parameter fields, check boxes, and so forth that comprise a block's dialog are known as dialog items.

☞ For a complete list and description of dialog items, see page 14.

Tabs

The block's Dialog tab has two tabs at the top—Options and Comments.

Tabs in a dialog are used to group dialog items by function. When you create or modify a block's dialog you can:

- Add tabs to the dialog by clicking the + sign
- Add dialog items to a tab after bringing the tab to the front
- Rename a tab by double-clicking its name and choosing a new name in the window that appears
- Delete a tab by clicking its close button, the X next to its name
- Move dialog items from one tab to another using the Cut, Copy, and Paste commands

Be careful when deleting a tab with dialog items on it. See the caution on page 19 about deleting dialog items from blocks that are used in models.

Dialog Resizer

Since they usually have different numbers and types of dialog items, a block's tabs are often different sizes. The Dialog Resizer allows you to position the right and bottom edges of each tab separately. This determines what is visible for each tab when it becomes the active window.

Use the keyboard's arrow keys, or drag the bottom right corner of the resizer, to set the default size for each tab. ModL functions can also be used to set or reset the size of each tab in the dialog.

☞ The arrow keys can be used to change the position of the resizer's bottom right corner.

Help tab

This tab is used to enter and edit the online Help for the block. This text appears when you click the Help button in the lower left corner of a block's dialog and is about the block and how it can be used in a model. It is not available through the ExtendSim Help menu command, which provides help for using the ExtendSim application.

You can add formatting to the text by selecting it and using the buttons in the Text toolbar.

☞ By default the block's name and the first sentence of the text in the Help tab is displayed as a tooltip when the cursor is hovered over the block. You can choose that just the block name, but not the Help text, be displayed. To do this, turn off the "Include additional block information" option in the Edit > Options > Model tab.

Dialog items

The frames, text, buttons, tables, and entry boxes that go into a dialog are collectively known as *dialog items*. Dialog items are created on the Dialog tab using buttons from the Dialog Items toolbar, shown below. Each dialog item is then configured or defined using options in its Properties window.

☞ See "Accessing dialog items from a block's code" on page 35 for how dialog items are called in ModL.

Types of dialog items

The buttons in the Dialog Items toolbar are shown above. The types of dialog items are summarized in the this table in the same sequence as the buttons.

Type	Description	Page
Parameter (Number)	Entry box that takes and/or displays a number. Users can dynamically link parameters to a cell in an ExtendSim database or global array instead of just entering data.	36
Popup Menu	A shadowed rectangle containing a menu of items. Each item in a popup menu has a label which can be changed using Modl code. Note: popup menu indexes start at 1, not 0. Each item in the list can have its text label formatted.	39
Checkbox	Square buttons that have a check mark in them when they are selected; the boxes are empty when not selected. Checkboxes can have labels that appear as text to their right; the labels can be changed using ModL code.	37
Button	A button that can be clicked, such as an OK or Cancel button. A button's label appears as text inside the button; the labels can be changed using ModL code.	38

Type	Description	Page
Radio Button	Round buttons that appear in groups where each group has a unique group number. Only one button in the group can be selected at a time; this causes all the other buttons in the group to become deselected. Radio buttons can have labels that appear as text to their right; the labels can be changed using ModL code.	37
Frame	Used to group dialog items. Has an optional label that will appear at the top of the frame. Does not require a dialog item name. This dialog item can have its text label formatted.	40
Static Text (Label)	Text that appears as a label in the dialog. Can be changed through ModL code but not by the modeler. Can be formatted.	39
Dynamic Text	Entry box that takes or displays text. Limited to 32,000 characters. This item uses an automatically resized dynamic array to store the text. It is useful for larger amounts of text such as for comments or for equations in the Equation and Optimizer blocks.	36
Editable Text	Entry box that takes or displays text. Limited to 255 characters. Users can dynamically link editable text to a cell in an ExtendSim database or global array instead of just entering data.	36
Editable Text 31	Same as Editable Text, above, but limited to 31 characters to save memory.	36
Data Table	A two-dimensional table with scrollbars and adjustable column widths (similar to a spreadsheet) for holding *numbers*. If block code provides for it, modelers can change the number of rows and columns and can dynamically link tables to an ExtendSim database or global array. Can have its text label formatted.	38
Text Table	A two-dimensional table with scrollbars and adjustable column widths (similar to a spreadsheet) for holding *text*. If block code provides for it, modelers can change the number of rows and columns and can dynamically link tables to an ExtendSim database or global array. The text can be formatted. ⚠ The Text Table uses a significant amount of memory and searching strings is slower than searching data. If possible, use the Data Table dialog item instead, especially if the table is large.	38
Slider	A control that allows you to select from a range of values by moving a value indicator. You can manually drag the knob to change a value or use ModL code to move the knob to show a value.	41
Meter	An output-only item that shows a needle in a meter.	41
Switch	A switch that resembles a light switch. It has two values, 0 (off) and 1 (on).	41
Calendar	Takes an ExtendSim date value and displays it on a calendar.	42

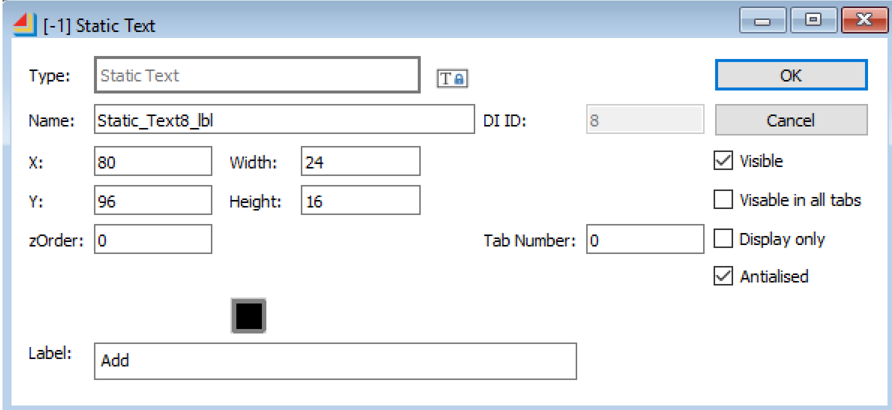
Type	Description	Page
Clock	Displays the time component of an ExtendDate value on a digital clock.	42

The Embedded Object dialog item is no longer used because embedded objects are not supported as of ExtendSim 10.

Properties of dialog items

To see the definition of a dialog item:

- Open the block's internal structure using one of the methods on page 7.
  - For example, open the structure of the Examples block (the block is located at ExtendSim/Documents/Libraries/Example Libraries/ Tutorial library)
- In the structure's Dialog tab, double-click the dialog item or right-click and select Properties
  - For example, double-click the "Add" dialog item of the Example block.
- This opens the definition window for the designated dialog item. In this example, it opens the properties for the dialog item labeled *Add*, as shown here.



Each properties window displays the properties of the selected dialog item. For the Add dialog item, the information is:

- The dialog item is of the type Static Text; its button in the Dialog Items toolbar is shown to the right of its type
- The variable name, for use in the script, is Static_Text8_lbl
- The dialog item's dimensions and position in the dialog are given
- Its zOrder is 0
- The text label that appears in the dialog is *Add*
- The label is visible on this dialog tab, but not visible on all the block's dialog tabs

Options in the dialog item's properties window

Each dialog item's properties window has multiple options, as discussed below. Not all options are available for all dialog items.

☞ In addition to the options in the properties window, some dialog items can have their text or labels stylized or aligned as discussed in "Stylizing and aligning dialog items" on page 20.

Option	Description
Type	A dialog item's type (static text, frame, etc) is determined by which dialog item you choose from the Dialog Items toolbar and cannot be changed through the properties window. Dialog item types are listed on page 14.
Name	A dialog item's Name is the variable name or message name used by the block's code to interact with that item in the dialog, as discussed in "Accessing dialog items from a block's code" on page 35. As of ExtendSim 10 names are required for all dialog items; default names are entered by the system for frames and static text. The name: 1) Must start with a letter 2) Cannot contain any non-alphanumeric characters 3) Can be up to 31 characters in length
X,Y,W,H	By default, the fields (X, Y, W, and H) reflect the size and position of dialog items as they are created. Their size and position can also be adjusted after the dialog item created.
zOrder	zOrder is the forward/backward position of dialog items. It's what is effected by the Bring to Front and Send to Back buttons in the Alignment toolbar. The zOrder numeric value gives programmers complete control over the dialog item.
Format	**Parameters only**. The number formats are: General, 2 decimal places, Integer, Scientific, and Percent.
Fill Color	Colors the text label. Works the same as the Fill Color tool in the Shapes toolbar. For dialog items with multiple labels (such as the Popup Menu) the selected color will be applied to all the labels. ☞ ModL code allows more control over dialog item appearance. For example, use the SetDialogItemColor function to set a color, and other functions to change the color or border depending on model circumstances. See "Dialog items" on page 267.
Border Color	Works the same as the Border Color tool in the Shapes toolbar. See above note.

Option	Description
Label	Dialog items have optional text labels. These can be up to 255 characters in length. If defined, labels are displayed as part of or along with the dialog item in the block's dialog. For example, the text that appears to the right of a checkbox is a label. For tables, labels are used to name the columns. ☞ If a dialog item has a label that will be displayed in the block's dialog, that label can sometimes be formatted using standard characters. See "Stylizing and aligning dialog items" on page 20. ⚠ If you use the ampersand character (&) in the label of a Radio Button, Checkbox, or Frame dialog object you will need to enter it twice (&&). Otherwise, the character will not show on the label.
DI ID	Upon creation, each dialog item is assigned a unique ID. Some functions use this identification number in addition to the Name, to control the item.
Tab Number	Determines which dialog tab the dialog item is on. For dialog items that are set to be visible on all tabs, the Tab Number is -1. For other items, the numbering starts with 0, indicating the first tab.
Tab Order	Many dialog items have a tab order number. This determines the order dialog items will be selected when modelers tab between entry boxes on the dialog. When the tab order is changed for one dialog item, the tab order for the other dialog items is automatically adjusted. ⚠ Be careful when changing the tab order of dialog items. If the block is used in a model, changing a dialog item's tab order can cause the block's cloned dialog items to become confused. In this case the clone will present itself with multiple question marks; it must be deleted and replaced.
Rows/Columns/Row Height	**Text Tables and Data Tables only**. These numbers are for the body rows and columns only; headers will be added automatically.

Option	**Description**
Visible	By default dialog items are visible in the tab in which they are created. (See below for making a dialog item visible in all tabs.) Unchecking this option provides a safe alternative to deleting a dialog item when you no longer want it in an existing block. ⚠ Deleting a dialog item from a block that is used in a model could disrupt the order of the data in the dialog. The data will have to be reentered for each instance of that block in all models that use it. Instead of deleting the dialog item, hide it by unchecking the Visible option. When the *Visible* option is unselected, the item will not appear in the dialog of the block in the model. It will, however, show in the Dialog tab as a red rectangle without text. Hidden dialog items can be moved in the Dialog tab. You can also revert a hidden dialog item by opening its properties and selecting the Visible option. ☞ You can also temporarily hide and show dialog items using the HideDialogItem function; it is described in the dialog item function list that starts on page 267. Also consider using the DIMoveTo and DIMoveBy functions to move dialog items out of the way depending on dialog settings.
Visible in all tabs	If a dialog has tabs, this option will cause the dialog item to appear on every tab. This is common for static text that describes the purpose of the block as well as the OK and Cancel buttons. For dialog items that are visible in all tabs, their Tab Number (discussed above) is -1.
Display only	Under normal circumstances, many dialog items are editable or respond to clicks in the dialog. Checking the *Display only* option means that a modeler can't change the dialog item, or that a click will be ignored, in the block's dialog. Instead, it can only be set through ModL code. This option is typically used to display results or to temporarily or permanently disable a dialog item depending on settings in the block. In a block structure's Dialog tab, display-only items have a gray outer border instead of the standard black border; this is shown at right. They also appear as dimmed in the block's dialog. Display-only dialog items can still be copied and cloned Regular 0 Display only 0 ☞ Using ModL functions, the state of any dialog item can be dynamically changed between editable and not editable.
Add to right click menu	**Buttons only.** Cause the block's Button dialog items to appear in a menu when a block on the model worksheet is right-clicked. These commands can then be executed without having to open the block's dialog to click the button.

☞ There are also functions that can control many dialog item properties. See "Dialog items" on page 267.

Stylizing and aligning dialog items

In addition to the above property options, numbers, and colors, some dialog items can have the style (and sometimes the alignment) of their text labels formatted in their Properties windows.

Dialog items that support stylizing and/or alignment

- Data tables
- Frames
- Popup menus
- Static text
- Text tables

Available formats

As indicated below, formatting is not case sensitive.

STYLE	ALIGNMENT
Bold: <B> or <b>	Left: <L> or <l>
Italicized: <I> or <i>	Right: <R> or <r>
Underlined: <U> or <u>	Centered: <C> or <c>

☞ For dialog items with multiple labels (such as popup menus or tables) set styles separately for each label.

How to format the text label

To format a label's style or alignment, in the dialog item's Properties window precede its text label with the initial(s) of the desired format within angle brackets as shown here.

Label: <b>Outputs the value of a system variable

Combined formats.

To combine formats, put multiple initials within one set of angle brackets. For example, a label that is bold, italicized, and right adjusted would be preceded by *<bir>*. The order of the initials for combined formatting does not matter.

☞ Text label formatting will only appear in the block's dialog, not in the Dialog tab. To see the format implemented, close and compile the block, then open its dialog.

Dialog item tooltips on block dialogs and in the Dialog tab

As indicated in the table above, each dialog item can have a name. This is how the items are referenced in equation-based blocks and in block code. If a dialog item has a name, tooltips will provide that information without you having to access the dialog item's properties window.

The command Edit > Options > Misc tab allows you to have tooltips show in the block's dialog and/or on the block structure's Dialog tab. With tooltips turned on, resting the cursor above a dialog item displays the name of the dialog item; the window will be blank if there is no dialog item name.

For more information about dialog items, see:

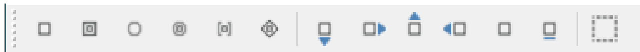
- “Accessing dialog items from a block’s code” on page 35
- “Working with dialogs” on page 95
- “Block connectors and connection information” on page 260

Connectors

Most blocks have connectors that transmit information to and from ModL code. Connectors and animation objects are added to a block in its structure window’s Icon tab using the Icon toolbar shown below.

The Icon toolbar

The Icon toolbar is used to add connectors and animation objects to block icons, both for hierarchical blocks and for the custom blocks you create. In both cases this is done in the Icon tab of the block’s structure window.



Connector types

The first six buttons in the Icon toolbar are for selecting the type of connector.

- The ExtendSim User Reference describes the use of the first five connectors: *value*, *item*, *flow*, *universal*, and *array*.
- The *user-defined* (or “diamond”) connector does not have any “special” properties, and is provided for your convenience for custom applications. For example, if you design a new set of blocks and want to be sure that modelers only connect those blocks to each other, you would use this connector. If the modeler tries to connect a diamond connector to a value or item connector, ExtendSim will not let them.

☞ By default new connectors are added to the icon as a ***normal*** (single/non-variable) connectors. They can be changed to variable connectors (arrays of single connectors) using the connector option tools, below.

Connector options

After the connectors, the next six buttons in the Icon toolbar are options for causing a connector to be *variable* rather than normal. While normal connectors represent one input or output, variable connectors act like a row of single connectors, where the row can be expanded or contracted to provide a required number of inputs or outputs.

The first five choices allow you to select, respectively, a variable connector that the modeler can expand downward, to the right, upward, or to the left, or which cannot be manually expanded or contracted (*variable connector with no resize bar*).

The sixth choice converts a variable connector into a normal connector. Use this if you selected a variable connector by mistake.

In the block’s code, you can specify a maximum and minimum number of variable connectors and change the number of connectors dynamically. An example of using normal and variable connectors is the Math block (Value library). To work with variable connectors, see the writeup on page 35 and the functions on page 264.

☞ Regardless of connector type (Value, Item, etc.), each connector can be normal or variable depending on the option selected in the Icon toolbar. However, the entire row of a variable connector must be of only one type of connector.

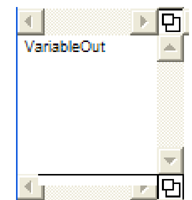
Animation object

The last button in the Icon toolbar is the Animation Object button. This is used to add 2D animation objects to a block's icon environment as discussed in "2D animation" on page 10 and "Adding 2D animation" on page 53.

Connector names

By default each connector is assigned a unique name which can be changed. The last part of a connector name defines whether it is an input or output connector.

Connector names are shown in the Connectors pane of a block's structure. For example, the name of the connector for the Example block is as shown at right. This block has one normal (non variable) value output connector named VariableOut.



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.

Rules for connector names

- Connector names are not case sensitive
- The name must start with a letter
- The name cannot contain any non-alphanumeric characters or spaces
- The name must be fewer than 32 characters in length
- Input connectors must end in some form of the word "In" (IN, in, In, iN)
- Output connectors must end in some form of the word "Out" (OUT, out, Out, ouT, etc)

Editing connector names

To change the name of a connector:

- Double-click the connector name in the connector pane. ExtendSim 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.

☞ You can name a connector anything you want as long as it follows the rules above.

Adding connectors to the icon

To add a connector to an icon:

- In the Icon toolbar (shown on page 21), click on one of the connector type buttons (Value, Item, Flow, Universal, Array, or User Defined)
- Then click in the block structure's Icon tab at the desired position

This will place a connector in the icon pane. By default, the connector is a "normal" connector.

To change the default normal connector to a variable connector:

- Select the connector in the Icon pane
- Select one of the variable connector options (down, right, up, left, or no resize) in the Icon toolbar

Normal
✔ Variable Down
Variable Right
Variable Up
Variable Left
No Resize Bar

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:

- Click on the connector in the block structure's Icon tab
- Click on the correct connector type in the toolbar

When you build blocks it is important that you use the above method to change connector types, rather than deleting a connector. If you delete a connector for a block that is used in any model, the order of the connections will be disrupted and existing connections to the block might become incorrect.

Connector labels

A connector label is associated with a particular connector and makes a block's inputs and outputs more understandable. Connector labels can be positioned, colored, and formatted, and are especially helpful to distinguish one input or output from the others when using variable connectors. Most ExtendSim blocks have one or more connector labels.

Connector label functions start on page 266. Connector labels are defined using the ConnectorLabelsSet function. The function's arguments allow setting the position and color of the label.

To format a connector label, precede the label text with angle brackets ("< "and ">") that enclose the desired format, using the same techniques shown in "Stylizing and aligning dialog items" on page 20. For instance, a bold, right-adjusted label named "want" would be entered as "<rb>want" in the Script tab.

Connector tooltips

Tooltips can display custom text when the mouse hovers over a connector. This is helpful to show additional information about the connector (including its value during the simulation) and what it can be used for. See "Connector tool tips" on page 267 and the Equation block (Value library).

For more information about connectors see:

- "Accessing connectors from a block's code" on page 34
- "Working with connectors" on page 102
- "Block connectors and connection information" on page 260

Overview

ModL Overview

Creating ModL code and dialogs for custom blocks

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

This chapter and the others that follow will teach you how to create new blocks, modify existing blocks, or call functions from equation-type blocks such as the Equation and Optimizer blocks (Value library) and the Equation(I) and Queue Equation blocks (Item library).

☞ These chapters only describe creating standard blocks, not hierarchical blocks. Hierarchical blocks contain ExtendSim blocks and are created through the user interface, without programming, as described in the User Reference.

As you saw in the preceding chapter, there are many parts of a block, the most complex of which is the ModL code. This chapter shows how a block's code is laid out and the basic ways that ModL code lets blocks interact with other blocks, with their own dialogs, and with the general parameters of a simulation.

☞ This overview of the ModL language is a prelude to, and should be read before, the "The ModL Language" reference chapter that starts on page 59.

ModL feature overview

As you will see, ModL is very much like C, although it is not as complicated and not case sensitive. ModL also has some enhancements that will help you create block code:

- Block code is organized as message handlers and function definitions, rather than just function definitions. Message handlers and functions can be overridden (e.g. if they are from include files) and 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 functions. The dialog and connector changes take effect immediately.
- Blocks can query, control, and send messages to other blocks, even if they are not connected.
- Blocks can have icon views facilitating the direction of model flow.
- The ModL language has several string data types. Strings can be up to 255 characters and can be concatenated (strung together) 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 over 1,200 ModL functions for performing general and simulation-specific tasks. In addition, there are many global variables that can be accessed from ModL code as well as some predefined constants.
- Including 2D animation in a block is easy. You do not need to do any graphics in your block code. Instead, you can position animation objects on the icon and use animation functions to show pictures or text within the objects.
- 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.
- You can create ExtendSim databases, linked lists, and global arrays that are useful as in-memory data repositories. You can also use ExtendSim blocks or code to access external spreadsheets and ADO and ODBC compatible databases.
- ModL allows for efficient connectivity with other languages, so you can make use of other features and technologies.

- ExtendSim has an internal source code editor (with code completion, syntax colorization, and more) as well as a source code debugger.
- An external source code feature allows multiple people to efficiently and effectively work on the code of blocks at the same time.

ModL compared to other programming languages

An ExtendSim block's code is written in ModL, a programming language that is much like C or C++ that has been enhanced for simulation purposes.

☞ DLLs and Shared Libraries provide a method for incorporating other technologies, such as Visual Basic, Java, or Visual C++, into ExtendSim. For more information, see "DLLs" on page 86.

If you code, and if you don't

- If you have some familiarity with C or C++: Although ModL is specialized for simulation, it uses many concepts from C++ and you can certainly program in ModL. Note that you do not need to know much C/C++ in order to be completely comfortable in ModL. The table that starts on page 27 lists major differences between ModL and C++.

- If you program in a language other than C/C++: ModL should not present much of a challenge. You can use your programming knowledge to program using ModL or write DLLs in other languages and call them from within ExtendSim code. The table that starts on page 29 compares common constructs for ModL and for some other common languages.

- If you do not program: You can still use the equation-based blocks, make some modifications to existing blocks, and build simple blocks without programming experience.

☞ Whether you know programming or not, the equation-based blocks provide access to ModL functions and variables. This is useful for accomplishing specialized tasks without having to program a block. The equation-based blocks are listed in the User Reference.

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 (Mac OS and Windows: 16 significant digits)	double
integer or long (32 bit)	long
string or Str255 (255 characters maximum)	typedef struct Str255 { unsigned char length; unsigned char str[255]; } Str255;

ModL	C++
Str15 (15 characters maximum) Str31 (31 characters maximum) Str63 (63 characters maximum) Str127 (127 characters maximum)	typedef struct StrN { unsigned char length; unsigned char str[N]; } StrN;
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
Functions and message handlers can be overridden.	Not available in C, but available in 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. Instead they must be of the format "i++;", or "i=i+1;" and "a[i] = 5;", or "a[i+1] = 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 (pointertypes can hold the address of a dynamic array)	Pointers
Linked lists support complex data structures	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	C++
#define, #include, #ifdef, #ifndef, #else, #endif can be used as preprocessor directives to conditionally compile code if a symbol of any type is defined (e.g. constant, variable, connector, function, etc.). There is no macro definition.	Macro definition is allowed in addition to preprocessor directives.

ModL compared to languages other than C++

The following table compares some of the common constructs in ModL to other languages.

☞ DLLs (Windows) and Shared Libraries (Macintosh) provide a method for linking languages other than ModL to ExtendSim. For more information, see “Accessing code from other languages” on page 42 and “DLLs” on page 86.

ModL	Java	Visual 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;
}
	double[] a = new double[10];
	double x;
	int i;
	
	for (i = 0; i < 10; i++){
	if (i == 5) {
	x = 3;
	}else{
	x = 5+i;
	}
	a[i] = x;
	}
		dim a(9) as single
		dim i as integer
		
		for i = 0 to 9
		if i = 5 then
		x = 3
		else
		x = 5+i
		a(i) = x
		end if
		next i
			real a(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];	int[][] b = new int[10][5]	dim b(9,4) as integer	integer b(9,4)

switch (i)
{
case 0:
x = 3;
break;
case 1:
case 2:
x = 5+i;
break;
default:
x = 0;
break;
}
	switch (i) {
	case 0:
	x = 3;
	break;
	case 1:
	case 2:
	x = 5+i;
	break;
	default:
	x = 0;
	break;
	}
		select case i
		case 0
		x = 3
		case 1
		case 2
		x = 5+i
		case else
		x = 0
		end select
			select case i
			case (0)
			x = 3
			case (1,2)
			x = 5+i
			case default
			x = 0
			end select

while (i < 10)
{
x = x+i;
i = i+1;
}
	while (i < 10) {
	x = x+i;
	i = i+1;
	}
		while i < 10
		x = x+i;
		i = i+1;
		wend
			do while (i .LT. 10)
			x = x+i;
			i = i+1;
			end do

ModL	Java	Visual Basic	FORTRAN

do
{
x = x+i;
i = i+1;
}
while (i < 10);
	do {
	x = x+i;
	i = i+1;
	}while (i < 10);
		do
		x = x+i
		i = i+1
		loop while i < 10
			10 x = x+i;
			i = i+1;
			if (i .LT. 10) goto
			10

//comments ** comments /* enclose multi-line comments like this */	//comments /* enclose multi-line comments like this */	rem comments ' comments (NOTE: only to end of current line)	! comments (NOTE: only to end of current line)
strng = strng+"abc";	strng = strng+"abc";	strng = strng&"abc"	strng = strng // 'abc'
strng = x;	strng = double.toString(x);	strng = Str(x)	depends on implementation

if (strng < "abc")
...
	if (strng.compareTo("abc") < 0)
	{
	...
		if strng <"abc"
		then
		...
			if (strng .LT. 'abc')
			then
			...

ModL language terminology

The following table describes the terms used in ModL coding.

Term	Description
array	An indexed list of numbers or strings, with indices starting at 0 (zero). Arrays can be fixed or dynamic. Dynamic arrays are static and cannot be locally declared. Fixed arrays can be declared as static or local.
constant	Value that does not change.
data type	The type of storage used for the data: real, integer, string, pointertype
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	Predefined named group of code instructions with specified arguments that may return a value (void functions don’t return a value) and can be called in a block’s code. Can be overridden by defining the function in an include file which can then be overridden in the block’s code.
global variables	Variables that are used to pass information between blocks. They are predefined by ExtendSim and can be viewed or modified by any block or equation. These variables’ values are preserved between simulation runs and are saved with the model. See “Scope of global, local, and static variables” on page 62 for more information regarding the scope of those variables.
identifier	Name that is entered.

Term	Description
literal	Number or string that is entered as a constant.
local variable	Variable that is locally declared and 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. See "Scope of global, local, and static variables" on page 62 for more information. ⚠ Do not give local and static variables the same name; local variables with the same names as static variables override the static variables.
message handler	Grouping of code that tells ExtendSim what to do in a particular circumstance that is defined by the message. Can be overridden.
statement	Section of code ending with ";".
static variables	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 it is saved. See "Scope of global, local, and static variables" on page 62 for more information. ⚠ Use caution when deleting static variables for blocks already used in models. It has the same harmful effect as deleting dialog items (discussed in the section on "Hiding/showing dialog items" on page 97).
system variable	Provide information about the state of the simulation. Like global variables, system variables are valid in any block in a model, are declared by ExtendSim, and can be viewed or modified by any block or equation.
type declaration	Defining a variable as a certain data type: real, integer, string, or pointertype.

Differences between equation blocks and programmed blocks

The equation-based blocks in the Value and Item libraries provide access to over 1,200 ModL functions; you can also use operators to enter logical statements, write compound conditions, and specify loops. The equation is automatically compiled when you click OK in the block's dialog.

Using an equation-based block you can accomplish much of what can be done with a custom-built block. However, there are some differences and limitations.

Feature	Equation-Based Blocks	Custom Blocks
Custom dialog	No	Yes
Pre-defined input and output variables	Yes	No
Data type declarations (real, integer, string, pointertype)	Yes	Yes
Constant definitions (e.g. "Constant N is 5")	Not directly; use an include	Yes
Pre-defined constants (Pi, Blank, True, False)	Yes	Yes
Dynamic arrays	No	Yes
Static fixed arrays	Not directly; use an include	Yes

Feature	Equation-Based Blocks	Custom Blocks
Locally declared fixed arrays	Yes	Yes
Static variables	Not directly; use an include or declare as an input variable	Yes
Global variables	Yes	Yes
Locally declared variables	Yes	Yes
ModL functions	Yes	Yes
User-defined functions	Not directly; use an include	Yes
Message handlers	Not directly; instead call SendMsgToBlock()	Yes
Syntax coloring	Yes	Yes
Code completion	Yes	Yes
Conditional compilation	Yes	Yes
Debugger	Yes	Yes
Call include files	Yes (see note)	Yes (see note)

☞ Include files used with equations are normally saved in the same location as the model using them; this makes it easy to move both the model and the includes it uses to a different location. Include files used in block code should be saved in the Extensions/Includes folder.

Structure of a block's ModL code

ModL is essentially C++ with enhancements and extensions to make it more robust for simulation modeling.

Layout of the code

Like C programs, a block's code starts with *data type declarations* and *constant definitions* (see page 33). Because you declare these at the beginning of the code, before any message handlers or user-defined functions, they are considered *static* or permanent variables. Unless overridden by a *local* variable declaration, static variables are valid throughout the block's code. However, their scope does not extend outside of that block's code. *Global variables* are predefined and have a global scope, making them valid in every block.

☞ For more information about the scope of variables, see "Scope of global, local, and static variables" on page 62 for more information.

After the type declarations, there are *function* (and *void function) definitions* and many *message handlers* (see page 33). This is where you write code and define the behavior of the block. The functions and message handlers are just definitions; they need to be called in order to be executed. They can also be overridden by re-declaring any number of times below the first declaration.

- Message handlers begin with a line "on *messageName*" and tell ExtendSim what to do in various circumstances; they are usually executed by the application. For example, a message handler in every block begins with "on Simulate" and the code within the message handler starts and ends with curly braces ("{" and "}").

- Functions are of the form "*type functionName(type argument, ...)*" or "*void functionName(type argument, ...)*" depending on whether they return a value. Functions are called within message handlers or from other parts of the block's code.

When you type the first letters of ModL functions in the Script tab, *code completion* pops up a window so you can get the correct spelling and arguments. As code is written, *syntax coloring* gives visual cues about its structure.

Like C, ModL ignores blank lines and indentation. Of course, it is a good idea to indent code with Tab characters and use comments. Single line *comments* are preceded by "//" or "**". Multi-line comments start with a "/*" and end with a "*/".

Data types

There are four main data types in ModL:

Data Type	Page
Real or double	60
Integer or long	60
String (Str15, Str31, Str63, Str127, Str255 or String)	61
Pointertype	61

Constants

Constant declarations can be of data type real, integer, or string, but not pointertype. The general form for a constant definition is:

```
CONSTANT id IS literal;
```

ModL includes four general-purpose predefined constants: Pi, Blank, True, and False. For more information, see page 62.

Constants are not directly supported in equation-based blocks; use an include or set the value from a Constant block (Value library).

Functions, message handlers, and local variables

ModL code has functions and message handlers that group the code into sections.

- *Functions* are procedures that do calculations and can be called from different points in the code. Functions can return a value; void functions do not return a value.
- *Message handlers* interpret messages that come from the simulation, from another block, or from user interaction with a block's dialog. Message handlers begin with a line "on *messageName*", where messageName is the name of the message. ExtendSim runs the message handler whenever one of the messages is passed to the block.
- *Local variables* are variables that are declared in message handlers and user-defined functions. Their scope is just the message handler or function in which they are declared.

Message handlers cannot be declared in equation-based blocks; instead call SendMsgToBlock(). See the table onpage 31 for additional differences when using equation blocks.

Message handler structure

Message handlers are denoted by:

```
on messagename
{
      zero or more declarations and/or statements;
}
```

MessageName must be the name of one of the messages listed in the chapter "Messages and Message Handlers". The code of the message handler is contained between the curly braces ("{" and "}") and tells ExtendSim what to do in the specific circumstance. To exit from a message handler before the ending brace, use a Return statement or an Abort statement.

For example, in a continuous model, the code in the "on Simulate" message handler is executed for every step in the simulation. However, the code in the "on InitSim" message handler is only executed once, at the beginning of the simulation.

You can declare local variables at the beginning of a message handler. However, you should not have a global and a local variable with the same name. The 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 variable will not be modified.)

Overriding user-defined functions and message handlers

Message handlers and user-defined functions can be overridden by being re-declared any number of times below the first declaration. This is useful in that include files can have basic forms of functions and message handlers which can then be re-declared and overridden in the main block code. See "Include files" on page 81.

☞ Message handlers are discussed more in "Message handlers" on page 75 and "Using message handlers" on page 111. For a list of messages, see the chapter "Messages and Message Handlers" that starts on page 193.

Other ModL features

- *Syntax coloring* gives visual cues about the structure and state of a block's code, making it easier to follow the logic. See "Syntax styling" on page 78.
- *Code completion* speeds up the coding process by reducing typos and other common mistakes. See "Code completion and call tips" on page 79.
- *Conditional compilation* allows segments of code to be compiled only if certain conditions are met. For more information, see "Conditional compilation" on page 82.

Accessing connectors from a block's code

As discussed in "Editing connector names" on page 22, when you create a block each connector's name must end in either "In" or "Out". Connector names are used as variables in ModL code.

☞ All connector names are real type variables. If you set a connector name to an integer, ExtendSim automatically converts the integer to a real.

☞ Typically, connectors pass values one at a time. As discussed in "Passing arrays" on page 104, connectors can also pass arrays of multiple values. Passing arrays is an easy way to pass more than one piece of information at a time through blocks.

There are several types of connectors as listed on page 21. No matter what their type, connectors can be single ("normal") or multiple ("variable").

Normal (single) connectors

By default a new connector is added as a normal (single or non-variable) input connector. That connector can be changed to a normal output connector by adding "Out" to the end of its name.

- To read from an input connector, use its name in the right side of a statement. For instance, read from an input connector called "firstConIn" with:

```
myNumber = firstConIn;
```

- To set the value of an output connector, assign it a value. For example, to set the value of an output connector called "totalOut", use:

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

Variable connectors

Each variable connector is actually a row of single connectors where the row expands and contracts. This allows the developer and modeler to control how many connectors are displayed for a particular purpose. They are described in the User Reference.

☞ See "Adding connectors to the icon" on page 22 for the steps required to change the default (normal) connector into a variable connector.

Accessing a variable connector from code

- Use the function ConArrayGetValue to read from a variable input connector. Note that input connector indexes start at 0.
- To set the value of an variable output connector, use the function ConArraySetValue. Note that output connector indexes start at 0.

Setting the number of connectors

The number of variable connectors can be managed directly within the code with the functions ConArraySetNumCons and ConArrayGetNumCons.

If the number of variable connectors changes, the application sends two messages to the concerned block. The first message is ConArrayChanged followed with the ConArrayChanged-Complete message.

For an example of how to access variable connectors from a block's code and change the number of connectors based on what is needed, see the code of the Math block (Value library).

☞ The entire row in a variable connector must be of only one type (Value, Item, etc.) and must be either an input or an output. Also, connector indexes start at 0 (zero).

Accessing dialog items from a block's code

The section "Dialog items" on page 14 introduced dialog items, including their definitions, options, and use.

Overview

The default dialog item names (OK and Cancel), as well as the names for any dialog items added to a block, are listed in the Dialog Item Names pane of the block's structure window. Use the Edit > Copy and Edit > Paste commands to copy dialog item names from that pane to use in ModL code.

As of ExtendSim 10 names are required for all dialog items. The system supplies a default name for static text and frames; it can be changed following the rules given in "Options in the dialog item's properties window" on page 17.

Using the names of the dialog items, you can read and set them as variables in the same way as you do connectors. Dialog item names can also be used as message names for use with message handlers.

☞ Any dialog name can be used in the ModL code as both a variable name and a message name.

Dialog messages

Message handlers were introduced on page 33. They interpret messages that come from the simulation, from another block, or from modeler interaction with the dialog.

Dialog messages come from modeler interaction with a block's dialog items. When a button in a dialog is clicked or a parameter is unselected (for example, after it has been changed), ExtendSim sends a message with the same name as the dialog item (e.g. *on DialogItemName*) to the ModL code.

For example, assume a block has a Count button. When that button is clicked in the block's dialog, ExtendSim sends the "Count" message to the block. If the block has an "on Count" message handler, it will be executed; if not, nothing happens.

Names assigned to a block's dialog items are listed in the block structure's Dialog Item Names pane.

Parameters and editable text

Assume that a parameter dialog item has the name "numberOfRecords". In the block's code you could have a statement such as:

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

Dialog items can also be set from inside the ModL code. For instance, to set the value shown in the "numberOfRecords" field to "1000", use the statement:

```
numberOfRecords = 1000;
```

The same methods work for editable text:

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

Parameters and editable text dialog items have a limit of 255 characters; editable text 31 has a limit of 31 characters and is used to save memory. In the dialog item's properties window you can choose the *Display only* option for a parameter or editable text dialog item. With this option selected, the item cannot be changed directly in the dialog; it can only be changed in the block's code, using the same techniques just shown.

Dynamic text

Dynamic text items allow up to 32,000 characters, whereas editable text dialog items are limited to 255 characters each. Dynamic text is useful when you need more text area than an editable text item can have. However, it is a little more complex to work with because ModL code needs to be written to set it up before it is usable and accessible. This is commonly done in the CreateBlock message handler (which occurs when the block is added to the model), but it can be done at other times to suit the functionality of the block.

Dynamic text items can be accessed directly via the string dynamic array assigned to the dynamic text item or using the dynamic text functions (see "Dynamic text items" on page 283). For an example of using dynamic text items, see the Equation block (Value library).

To declare the string dynamic array:

```
string aStringDynamicArray[]; // the dynamic array declaration
....
```

To set up the dynamic text item in the CreateBlock message handler:

```
myDynamicTextItem = DynamicTextArrayNumber(aStringDynamicArray);
```

To directly access the text:

```
first255Characters = aStringDynamicArray[0];
```

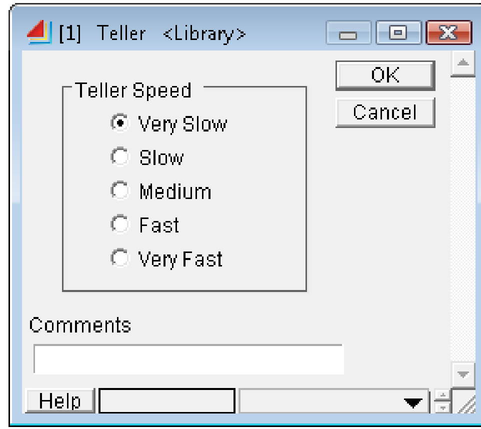
The example above shows a dynamic text dialog item attached to a string array. Dynamic text dialog items can be attached to arrays of any size string.

Use the dynamic text functions to find and replace text. See "Dynamic text items" on page 283.

Checkboxes and radio buttons

Checkbox and radio button dialog items return true/false values: true if the checkbox or radio button is selected, false if not. They also send their dialog item name (as a message name) to the block's code when they are clicked. The code could have an "On DialogItemName" message handler to process the message. The dialog item name could also be used as a variable to query or set the dialog item's value.

For example, if a block represents a teller in a bank, instead of entering a number you could use radio buttons to set the teller's speed. The dialog might look like the one shown at the right.



The five radio buttons would have the dialog names VSlow, Slow, Med, Fast, and VFast. To make sure that only one of them can be selected at a time, they must all have the same Radio Group ID when they are defined. In this example, the group ID was left 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 normal
if (Slow) theDelay = initDelay * 1.1;// slow is 1.1 of normal
if (Med)  theDelay = initDelay * 1.0;// medium is 1 of normal
if (Fast) theDelay = initDelay * 0.91;// fast is 0.91 of normal
if (VFast)theDelay = initDelay * 0.8;// v fast is 0.8 of 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."

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

```
Med = TRUE;
```

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

☞ When setting the state of a radio button group in ModL code, always explicitly state which button is set to True so that the remaining radio buttons in the group will be set to False. Just setting a radio button to False will not affect the state of the other radio buttons in the group. Thus it is possible to have a condition in which all radio buttons in the group are initially set to False. In most cases, this would be an error condition.

Use the DITitleSet function to change the title of a check box or radio button.

If you use the ampersand character (&) in the label of a Radio Button, Checkbox, or Frame dialog object you will need to enter it twice (&&). Otherwise, the character will not show on the label.

Buttons

When buttons are clicked, they send a message to the block's ModL code. The most common buttons in a block are OK and Cancel, which are handled automatically. If you add other buttons, such as shown later in this chapter, message handlers must be added for those buttons.

To change the text label of a button, assign the button's dialog item name as a variable to a string value in the code.

☞ The changed button text is not stored in the block. Thus the read value of the button's text is always what was originally entered when the dialog item was defined, even if the text is changed by setting it to a different value in ModL code. If you use the dialog item name in an equation, you will always get the text that was entered when the dialog item was created.

If you change the text label of a button, your code must set the text when the modeler opens the block. Do this using the On DialogOpen message handler.

```
on DialogOpen
{
myButton = "desired button text"; // set it when the dialog opens
}
```

See also "Changing text in response to a user's action" on page 96.

Data tables and text tables

A data table or text table dialog item represents a two-dimensional array of either real numbers or text. Tables have an interface that allows modelers to type in any input and also, like all dialog items, allows you to display values generated in the code. If your block code provides for it, modelers can change the number of rows and columns and can dynamically link tables to an ExtendSim database or global array.

You define the number of rows, number of columns, number format, and headings for the columns, but all of these are changeable with ModL code, including the ability to create and manipulate extremely large tables with up to 255 characters per cell.

The following code fragment shows how to read and write to a data table named "dataTable" that has 4 rows and 3 columns. Data tables are treated as arrays, which are discussed in detail in "Arrays, pointers, queues, delay, linked list, and string lookup table functions" on page 340. As in the C language, array subscripts 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;
```

Data tables can be attached to dynamic arrays. They can also have variable columns and the behavior of the columns can be extensively customized. For more information, see the description and functions in "Block data tables" on page 274 and "Formatting/interactivity using column and parameter tags" on page 284.

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

The headers for data tables and text tables can be styled and aligned, as discussed in "Stylizing and aligning dialog items" on page 20.

Static text (label)

Static text appears in a dialog as a label and is non-editable by the modeler. The system assigns default names to static text; the names can be changed by the block developer according to the rules given in "Options in the dialog item's properties window" on page 17. Names can be used in the ModL code to change the text label, show and hide it, and so forth.

☞ The changed text label is not stored in the block. Thus the read value of static text is always what was originally entered when the dialog item was defined, even if it is changed by setting it to a different text value in the code. If you use the dialog item name in an equation, you will always get the text that was entered when the dialog item was created.

If you change the text of a label, your code must set the text when the modeler opens the block. Do this using the On DialogOpen message handler.

```
on DialogOpen
{
myLabel = "desired text"; // set it every time the dialog opens
}
```

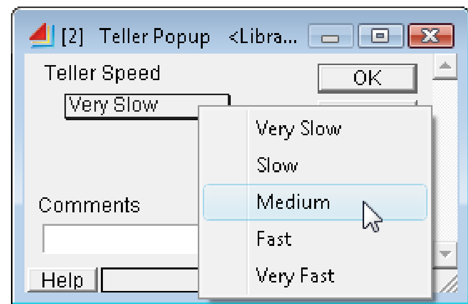
See also "Changing text in response to a user's action" on page 96.

Popup menu items

When an item in a popup menu is selected, the menu's dialog item 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 item name will be set to 1, 2, 3, 4, or 5 based on which menu item the modeler chooses.

☞ For historical reasons, popup menu indexes start at 1, not 0.

Popup menus replace series of radio buttons. For instance, instead of using several radio buttons to represent teller speed, as shown in “Checkboxes and radio buttons” on page 37, you could use a popup menu. The dialog would look like the screenshot at right.



This popup menu has the dialog item 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% normal
if(MyMenu == 2)theDelay = initDelay * 1.1; // slow is 110% of normal
if(MyMenu == 3)theDelay = initDelay * 1; // medium is normal
if(MyMenu == 4)theDelay = initDelay * 0.91;// fast is 91% of normal
if(MyMenu == 5)theDelay = initDelay * 0.8;// v fast is 80% of normal
```

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

```
MyMenu = 3; // defaults to the “Medium” menu item, 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 modeler selects a menu item. For instance, you could use this to report errors to the modeler, show or hide other dialog items, or cause a sound to play.

The text in popup menus can be styled and aligned, as discussed in “Stylizing and aligning dialog items” on page 20.

☞ Using the column tag functions listed in “Formatting/interactivity using column and parameter tags” on page 284, data tables can have popup menus in their columns.There are also functions to dynamically popup a menu on the fly, whenever the modeler clicks something.

Frame

A frame allows you to visually isolate or group dialog items by framing them with a rectangular box. Once a frame has been added to the Dialog tab, it can be resized and positioned to surround the items of interest. If present, the frame’s label is displayed in its upper left corner, as shown at the right.

Select the system variable to output
Current time
Add to result

Frames are required to have names and the system assigns a default. The default name can be changed following the rules discussed in “Options in the dialog item’s properties window” on page 17. Names can be called as a variable in the block’s code. This is helpful for showing and hiding the frame under different circumstances, dynamically repositioning it, or dynamically changing its label.

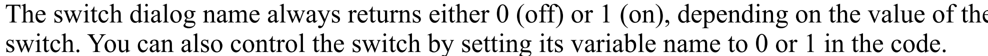
The frame’s label can be styled and aligned, as discussed in “Stylizing and aligning dialog items” on page 20.

If you use the ampersand character (&) in the label of a Radio Button, Checkbox, or Frame dialog object you will need to enter it twice (&&). Otherwise, the character will not show on the label.

Switch

A switch looks like a standard light switch.

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



The switch dialog name always returns either 0 (off) or 1 (on), 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 its length.

The minimum and maximum values can be set from ModL code; the modeler can change the minimum and maximum values by clicking and editing them in the block’s dialog. 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.

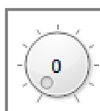
The slider is represented in a three-element array of reals that represent the minimum, maximum, and current values. Thus, if a slider is called “theSlider,” it could be initialized with the lines:

```
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, the value of the slider can be checked by reading the third array element:

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

Meter



A meter allows you to view a value on a 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 can set the value shown on the meter in the third array element, such as:

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

Calendar

The Calendar dialog item takes an ExtendSim date value (discussed in “Calendar Date functions” on page 362) and visually displays it in a combination calendar/clock format. For example, assume a calendar dialog item named MyCalendar has been added to a block’s dialog. That dialog item will display December 1, 2007 12:01 am when the following line of code is executed:

```
MyCalendar = 39417.000694444;
```

The dialog item would then look like the screenshot to the right.

Clock

The Clock dialog item takes an ExtendSim date value (discussed in “Calendar Date functions” on page 362) and visually displays the time component of that date value (hours, minutes and seconds) in a digital clock format. For example, assume a clock dialog item named MyClock has been added to a block’s dialog. That dialog item will display 12:01:00 when the following line of code is executed:

MyClock = 39417.000694444;

If you need to display the complete date value, and not just the time component, consider the Calendar dialog item, discussed above.

Accessing code from other languages

You can use other languages, such as Visual C++, Java, C++, and others, to provide additional functionality to ModL or to make use of a legacy of pre-built code. For instance, 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 the C++ code as a DLL (dynamic link library for Windows) or Shared Library (external command for Mac OS). DLLs and Shared Libraries are the standard interface technologies for communicating between programming environments. When you want to do the calculation, use the ExtendSim built-in function calls to call the DLL or Shared Library.

One advantage of this method is that you don’t have to program the interface in the external language. Instead, you can leverage the ExtendSim built-in interface and dialog creating capabilities.

☞ Even though you are using an external language, the resulting block will fit seamlessly into the ExtendSim environment and be indistinguishable from other blocks.

To learn more see “DLLs” on page 86 and “Sounds” on page 89.

External source code

ExtendSim supports a mechanism for saving the source code of individual blocks or libraries of blocks as external files. This external source code feature is useful for situations where multiple developers work on the code of blocks and/or libraries at the same time. For more information, see “External source code control” on page 83.

Tutorial

Creating a Block

Learn how to build an ExtendSim block
and save it in a library

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

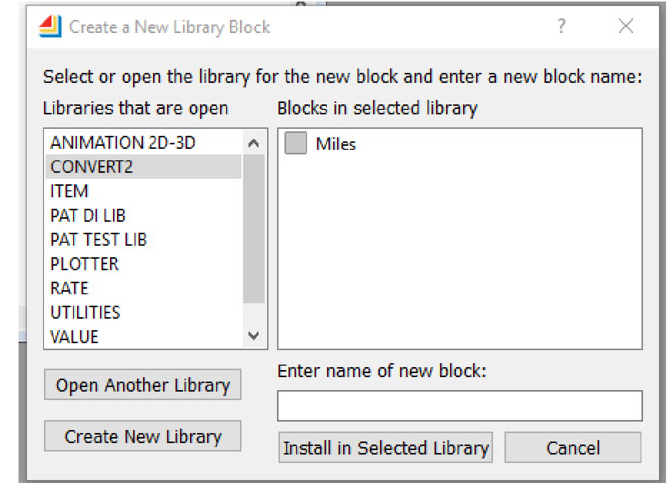
This chapter shows how to create a continuous block and have it perform some tasks. So you have something to compare to, the final block, called “Miles”, is located in the Custom Blocks library which is located in the folder ExtendSim/Documents/Libraries/Example Libraries.

Building a simple block that converts miles to feet

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

Create the block

- Choose the command Develop > New Block.
- At the bottom of the dialog, click the Create New Library button.
- Name the library **Convert** and click Save. Notice that in the pane on the left, the new library is automatically selected.
- Enter the name **Miles** for the new block, as shown at right.
- Be sure the correct library is selected, then click the Install in Selected Library button.



ExtendSim creates the block and opens its structure window.

Dialog tab

By default, the structure’s Script tab opens in front so you can easily edit an existing block. However, when building a new block it is best to start with the Dialog tab.

- Bring the structure window’s Dialog tab to the front

The dialog is empty except for the OK and Cancel buttons, which are in every new block by default, a frame to indicate the default size of the dialog, and tabs at the top.

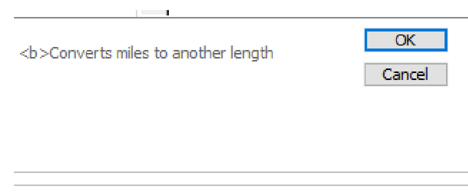
- Double-click the name of the first tab (Tab 1) and rename it to **Converter**

To add some explanatory text for this block:

- If the toolbar is not already open, choose the command Tools > Dialog Items.
- In the Dialog Items toolbar, select the **Static Text** [T] dialog item.
- Click on the structure window’s Dialog tab to create the Static Text dialog item.
- Double-click the dialog item to open its properties window.

To set the properties for this dialog item:

- To be consistent with the ExtendSim style guide, set **X: 10**, **Y: 16**, **Width: 220**, **Height: 16**. (Alternatively you could resize and position the dialog item on the Dialog tab.)
- In case you will add more tabs, check the box so that this text label will be **Visible in all tabs**.
- In the Label field, enter **<b>Converts miles to another length**. (As discussed on “Stylizing and aligning dialog items” on page 20, entering “<b>” in front of the text formats it as bold; this style will appear once the block is compiled.)



- Leave the rest of the options as is and click OK to close the properties window

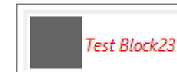
With those coordinates, the label will be placed in the upper left corner of the Dialog tab. It should now look like the screenshot here.

Save the block

It’s a good idea to save the block as you make changes to it. Since there isn’t any code in this block yet, you should save it without compiling.

- Choose File > Save Block

This saves the structure of the block without compiling the code. Alternatively, you could close the structure window and, in the window that appears, choose to Save Without Compiling.



☞ In the library window, uncompiled blocks are displayed with their names in red italics.

Icon tab

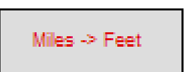
The Icon tab is for creating an icon and adding connectors.

- Go to the structure window’s Icon tab
- The Icon tab has a default icon

Icon and text

For the icon you could use the ExtendSim drawing environment or a painting program, or copy clip art and paste it in. For this tutorial, just keep the default icon and add text.

- Double-click on the Icon tab worksheet to create a text box:
  - Enter the text **Miles->???**
  - With the text selected, use the Bold command in the toolbar to make the text bold and the Fill Color button in the Shapes toolbar to make the text red



- Position the text on top of the icon.
- As needed, resize the icon and text using their resize buttons

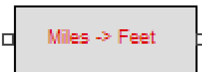
The icon should now look similar to the one shown above.

Connectors

The Miles block needs two value connectors, an input and an output. As discussed on page 21, connectors can be either normal (a single connector) or variable (a row of single connectors). The default is that connectors are normal, and that is what is required for the Miles block.

Adding connectors

- If the Icon tool is not already open, choose the command Tools > Icon.
- In the Icon toolbar select the **Value** connector, the first one in the list.
- Click near the left side of the icon to add that value input connector. Notice that the name of the connector (Con0In) is listed in the Connectors pane.
- Select another Value connector from the Icon toolbar and place it on the right side of the icon.
- Select each connector and either drag it or use the arrow keys so that the connectors are positioned on each side of the icon.



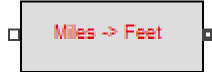
☞ To align the top of the connectors, either drag or use the arrow keys to move one connector until its Y position (as shown in the Cursor Position field of the ExtendSim toolbar) is the same as the other connector's Y position.

Renaming the connectors and changing an input to an output

When connectors are added to an icon, they are by default all input connectors with the default connector names Con0In, Con1In, and so on.

Connector names are not case sensitive. While input connectors must end in some form of the word "In" and output connectors must end in a form of the word "Out", the rest of the connector name can be customized as discussed on page 22.

- In the Connectors pane at the right side of the Icon tab, double-click the connector name **Con0In**.
- Type **MilesIn**. This changes the name of the connector on the left of the icon.
- Double-click the connector name **Con1In** and type **UnitsOut**. This changes the connector on the right to an output connector, as shown.
- Give the command File > Save Block. This saves your work without having to compile the block.



When you select a connector name in the Connectors pane, the corresponding icon connector gets highlighted.

☞ Because ModL is not case sensitive, the connector name "UnitsOut" could just as well have been written as "unitsout" or "unitsOut" or "UNITSout", etc.

Block help

This block has only one tab which is named Miles Converter. You could name another tab Help or Information, and put text there to explain what the block does, who authored it, create and modify dates, etc.

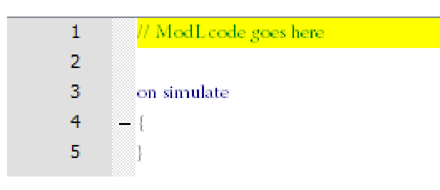
Script tab

☞ As shown on page 78 there is a Script dialog in the Edit > Options menu that is used to specify characteristics for a block structure's Script tab. For example, you can change the colors of

user functions or keywords. When the block's Script tab is the active window, use Alt + O to open the Script dialog.

- Go to the structure window's Script tab

As seen at the right, the Script tab is empty except for a comment ("ModL code goes here") and three message handlers: Simulate, CheckData, and InitSim.



These message handlers get automatically added because most of the blocks you create will use these messages and unused message are ignored.

For this block, the action happens in the Simulate message handler.The other default message handlers can be left blank since there is nothing to check or initialize at the beginning of the simulation run.

- Below the opening bracket that follows the Simulate message, type: **UnitsOut = MilesIn * 5280.0;**
  so that the code looks as follows:

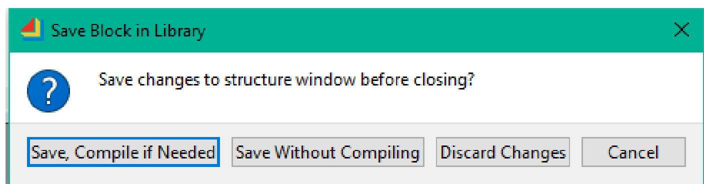
```
on Simulate
{
UnitsOut = MilesIn * 5280.0;
}
```

☞ The spaces on each side of the operators are optional; **the semicolon at the end of the statement is not optional**.

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

Save and compile the block

After the code has been entered:



- Close the structure window by clicking its close box or giving the command File > Close.
- ExtendSim opens a dialog for saving changes, seen at right. Click **Save, Compile if Needed**. This compiles the ModL code and saves the changes to the block in the library.

If there are any compilation errors, ExtendSim will warn you.

☞ While not done for this block, the blocks you create can also be assigned categories, as described on page 55.

Test the block

To test the Miles block:

- Open a new model window if one is not already open.
- If the model opens with an Executive block, delete the Executive block, since this isn't a discrete event simulation.

- Add your Miles block (Convert library) to the model worksheet.
- Add a Constant block (Value library) to the left of the Miles block.
- Connect the output of the Constant block to the input of the Miles block.
- In the Constant block's dialog enter **Constant value: 10**.

Constant block connected

- Add a Line Chart block (Chart library) to the model.
- Connect the output of the Miles block to the top input of the Line Chart block as shown here.
- Run the simulation.

Test model completed

As seen in the Line Chart block, the value 52800 (5280 x 10) should be output for the entire length of the simulation. You can verify that the block works with other numbers by entering them in the Constant block and running the simulation.

Adding user interaction and display features

If moving numbers between input and output connectors was all that ExtendSim did, it would not be a very useful program. As you have seen in the User Reference, robust blocks:

- Let you change their parameters
- Give information about what is happening during the simulation

Dialog tab

The enhanced Miles dialog will have a popup menu for choosing a unit to convert to, parameter fields for obtaining an input from the user and for displaying results, and a frame for organizing the dialog items.

Create a popup menu

- Open the structure window for the Miles block, using one of the methods listed in "How to open a block's internal structure" on page 7.
- Go to the block's Dialog tab
- If the Dialog Items toolbar isn't already open, open it from the Tools menu
- Click in the Dialog Items toolbar to select a Popup Menu and place it on the Dialog tab

Define the popup menu

- Double-click the Popup Menu to open its properties and configure it as below:
  - Name: units_pop
  - X: 24
  - Y: 70
  - Width: 76
  - Height: 16
  - Label: Kilometers;Yards;Feet;Inches

- Close the properties window

☞ While not required, coding conventions help when creating blocks. For instance, the popup's dialog item name (units_pop) is more intuitive to work with in the code, as can be seen later.

Add a text frame

Frames are used to organize dialog items on the window:

- Add a Frame dialog item from the Dialog Items toolbar and stretch its opening so it surrounds Popup Menu.
- In its properties window, enter:
  - Label: **Convert miles to:**
  - It is not necessary to change the dialog item name or any other properties for the frame.

The Dialog tab should now look like the one shown above.

Save the dialog changes

- Give the command File > Save Block to save your work

☞ This is a quick way to save changes without closing the block's structure.

Add two parameter fields

To add entry and reporting fields for the numbers:

- In the Dialog Items toolbar:
  - Click once to select the **Parameter** dialog item, **but then...**
  - While holding down the Alt key, click twice on the Dialog tab to create two parameter fields
  - Press the escape key to stop placing parameters on the Dialog tab
  - Move the parameter fields so that they are at the same horizontal position and below the frame

☞ To create multiple instances of a dialog item, hold the Alt or Option key down while repeatedly clicking on the Dialog tab; then use the Escape key to finish. Alternatively, once you've placed one parameter on the Dialog tab, you could use the Edit > Duplicate command to add copies of dialog items to the Dialog tab, or just get the dialog item from the Dialog Items toolbar again.

Configure the parameter fields

- In the properties window for the leftmost parameter:
  - Enter Name: **InNum_prm**
  - Set the width to 60 and the height to 16.
  - Click OK to close the properties window.
- In the properties window for the parameter on the right:
  - Enter Name: **OutNum_prm** and click OK to close its properties window
  - Set the width to 60 and the height to 16.

- Select the *Display only* choice. This will cause the parameter value to be displayed in the block's dialog without being editable.
- Click OK to close its properties window.

Adding more static text labels

The next step is to define two more Static Text dialog items.

- For the first static text item:
  - Enter **In:** for the label. Click OK.
  - Move it to the left of the first parameter box and resize as needed.
- For the second static text item:
  - Enter **Out:** for the label. Click OK.
  - Move it to the left of the second parameter box and resize as needed.
- Save and compile the block as discussed on page 47.

Script tab

The next step is to enter code in the structure's Script tab.

- Go to the block structure's Script tab

Declare constants

- At the top of the Script tab, before any message handlers, enter the following:

```
constant UNITS_KILOMETERS is 1;
constant UNITS_YARDS is 2;
constant UNITS_FEET is 3;
constant UNITS_INCHES is 4;
```

The constants are set to the values of the menu items (kilometers, yards, etc) in the popup menu.

CreateBlock message handler

The CreateBlock message handler can specify one of the conversions as the default for the block when it is placed in the model. For instance, to make feet the default, enter the following:

```
// feet is the default setting
on CreateBlock// this gets executed when the user gets a new block
{
units_pop = UNITS_FEET;// make the FEET choice TRUE
}
```

You can put the CreateBlock handler anywhere in the block's code as long as it is after the declarations.

☞ The CreateBlock message handler is invoked whenever the Miles block is placed on a model worksheet. If you've already got the block on the worksheet, feet won't be the default setting.

Simulate message handler

For each step in the simulation, you want to update the input parameter and the output parameter, check which conversion is being performed, multiply the numbers. To do this, **replace** the current code in the Simulate message handler with the following:

```
on Simulate
{
InNum_prm = MilesIn;
if (units_pop == UNITS_KILOMETERS)
     UnitsOut = MilesIn * 1.609344;
else if (units_pop == UNITS_YARDS)
     UnitsOut = MilesIn * 1760.0;
else if (units_pop == UNITS_FEET)
     UnitsOut = MilesIn * 5280.0;
else if (units_pop == UNITS_INCHES)
     UnitsOut = MilesIn * 63360.0;
OutNum_prm = UnitsOut;
}
```

Notice that "units_pop" is the name of the popup menu defined on page 48. Also notice that if "Yards" is selected in the block's dialog, the statement "if (**units_pop == UNITS_KILOMETERS)**" evaluates false and its "else" clause is executed. The "if (**units_pop == UNITS_YARDS**)" statement then evaluates true and executes its statement, and the "else" clauses following it will not be executed.

☞ The numbers multiplied by MilesIn can be reals or integers. ModL performs all necessary type conversions automatically. However, since connectors are always of type real, you should use reals for values that are used with connectors. That way, ExtendSim will not need to convert integers to reals on each step. (For more information about type conversions, see page 64.)

Save and compile the block

After you have finished writing the code, close the block by clicking its close box. Choose *Save, compile if needed* to compile the new ModL code and save the block in the library.

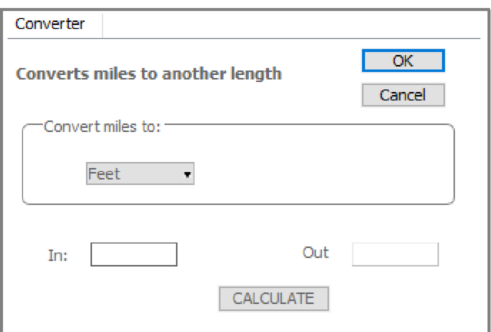
Test the Miles block as you did before, trying all the conversions. Notice that you can keep the Miles dialog open during the test.

Adding an intermediate results feature

You may have noticed something missing from the Miles block: what if you just want to convert a single number without running a whole simulation? As stated earlier, ExtendSim 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.

Add a button to the Dialog tab

- Open the block's internal structure.
- In the Dialog tab, choose a Button dialog item from the Dialog Items toolbar and place it on the dialog.
  - Enter **Calculate_btn** as the dialog name and **CALCULATE** for the button label.
  - Make the width 80 and the height 20.
  - Click OK.
  - Drag this new button below the input and output fields, as shown at right.



Add ModL code to the Script tab

- Add the following message handler somewhere below the Simulate message:

```
on Calculate_btn// Display the values
{
if (units_pop == UNITS_KILOMETERS)
     OutNum_prm = InNum_prm * 1.609344;
else if (units_pop == UNITS_YARDS)
     OutNum_prm = InNum_prm * 1760.0;
else if (units_pop == UNITS_FEET)
     OutNum_prm = InNum_prm * 5280.0;
else if (units_pop == UNITS_INCHES)
     OutNum_prm = InNum_prm * 63360.0;
}
```

- Save the block and compile the ModL code.

☞ See "Other features you might have used" on page 54 for how to avoid typing the same code twice in the future.

To test this new functionality:

- Place the block on a model worksheet and double click to open the block's dialog. You may need to resize it to show the new dialog items.
- Type a number in the **In:** entry box.
- Click the Calculate button.

Note that the number in the *Out:* entry box is updated with the correct data.

☞ See "Accessing code from other languages" on page 42 for an example of how to do this using a DLL.

Adding 2D animation

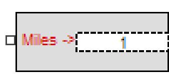
The icon of this block indicates that it converts miles to something, but that's not very informative. Displaying text on an icon is a convenient method to show block conditions – in this case, what kind of conversion the block is performing.

☞ The ModL code will ensure that the animated text is displayed even if the modeler doesn't have Show 2D Animation selected; the display is also independent of the simulation run.

Change the icon and add the animation object

As discussed in "2D animation" on page 10, the Animation Object tool is used to add animation to icons. This tool is the last one in the Icon toolbar, shown below.

- Open the structure window of the Miles block.
- In the Icon tab, delete the *???* part of the icon's text.
- Move the remaining text to the left side of the icon.
- In the Icon toolbar, select the Animation Object tool; it is the last button in the toolbar.
  - Click on the icon to place the Animation Object to the right of the text.
  - Expand the Animation Object to the right. This creates a rectangular animation object. Since this is the first animation object, it will have a "1" in it.
  - Expand the icon as necessary to provide enough room for the animation object.



Animation object added

Add code for the animation

☞ The code below contains a ModL function. It is easiest, and safest, to enter functions using code completion, as described on page 79.

Add additional message handlers

- Add the following message handlers to the end of the ModL code. Each message handler receives a message when the corresponding radio button is clicked.

```
on units_pop
{
     if(units_pop == UNITS_KILOMETERS)
          {
          AnimationTextTransparent(1, "KM");
          AnimationShow(1);
          }
     else if(units_pop == UNITS_YARDS)
          {
          AnimationTextTransparent(1, "Yards");
          AnimationShow(1);
```

```
            }
        else if(units_pop == UNITS_FEET)
            {
            AnimationTextTransparent(1, "Feet");
            AnimationShow(1);
            }
        else if(units_pop == UNITS_INCHES)
            {
            AnimationTextTransparent(1, "Inches");
            AnimationShow(1);
            }
}
```

The ModL compiler will give an error message if code containing “smart quotes” is copied into the Code pane. To fix the problem, replace the copied quotes with new ones in the Code pane.

Createblock message handler

- *Replace* the code in the “On CreateBlock” message handler to initialize the animation text.

```
// feet is the default setting
on CreateBlock
{
units_pop = UNITS_FEET;
AnimationTextTransparent(1, "Feet");
AnimationShow(1);
}
```

- Close the block’s internal structure, saving and compiling the changes.

☞ Notice that the text will show only when a new Miles block is placed on the model worksheet.

Testing the block

To test this new functionality:

Icon with animation

- When you place a new Miles block in a model, the default setting (Feet) should be displayed on its icon.
- Changing the selected radio button in the block’s dialog should cause a corresponding change on the icon.

The final Miles block is located in the ExtendSim/Libraries / Example Libraries / Custom Blocks library. See below for additional features to consider when building custom blocks.

☞ Animation objects can be layered on top of each other, then called in block code to appear or not depending on circumstances. Use each animation object’s zOrder to place it in the layer you want. For more information, see “Creating 2D animation objects” on page 129.

Other features you might have used

The preceding example showed how to build a simple block. This section describes some additional ExtendSim features that could have been used when building this block.

Feature	Description	Page
Block categories	Group blocks by category in the Library menu	55
Colors for dialog items	Custom colors for text labels	17
Styles and alignment	Some dialog items can have their style formatted	20
Tabs in dialogs	Group dialog items by function	13
zOrder for dialog items	Control the forward and backward position of dialog items	17
Add to right-click menu	Cause the block's Button dialog items to appear when a block is right-clicked	17
Dialog item tooltips	Tooltips display dialog item names without having to open the block's dialog	20
Hiding/showing dialog items	Use code to dynamically hide and show dialog items based on model conditions	97
Icon views	Multiple icons per block; user selectable	9
Variable connectors	They act like a row of normal (single) connectors	21
Connector labels	Tooltips display connector information such as results.	23
Connector tooltips	Display custom text through tooltips	23
Column tags	Put strings, checkboxes, buttons, dates, popup menus, the infinity character, and so forth into a parameter field or the cell of a data table	284
Programming tools	Code completion, include files, debugger, conditional compilation, and more	78
DLLs and Shared Libraries	A legacy of code, or programming capabilities that are outside of ModL's range, can be accessed using DLLs (for Windows) or Shared Libraries (on the Macintosh)	86 89
External source code control	Saves the source code of blocks as external files so multiple developers can work on it	83

Linking to a global array or ExtendSim database

Dynamic data linking (DDL) creates a link between a data table or parameter dialog item and an internal data repository (global array or ExtendSim database). For instance, the Data Source Create block (Value library) has a data table that is dynamically linked to a global array. To add dynamic link capability to a block, see “Data table linking” on page 275 and “Dynamic linking” on page 280.

To simply register a block so that it will be notified if there was a database change, see “Registered blocks” on page 113 and “Linking and notification” on page 337.

Block categories

It is common to 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 *category* and cause blocks to be grouped into submenus in the Library menu.

Each block has a 15-character string associated with it called the block category. The category is used by ExtendSim to organize blocks into logical groups that perform a similar function.

For example, the Math block (Value library) performs basic math functions; therefore, its block category is *Math*. ExtendSim uses the block category in two places: 1) to organize blocks within the Library menu, and 2) to organize blocks within reports. Categories of blocks in libraries and reports are discussed in the User Reference.

To set or edit a block’s category:

- With the block’s structure window open, select the command Develop > Set Block Category.
- In the Category dialog, either select an existing category from the popup menu or define a new one by entering it in the text field.
- Click OK.

The Edit > Options > Libraries tab has a checkbox (List blocks in Library menu by category) that is used to determine how ExtendSim will list this block within the Library menu. If checked, the block will be listed in a sub-menu for the block category under the library name. If the box is unchecked, ExtendSim will not use the category but will instead list the block alphabetically directly under the library name.

Defining functions

In the Miles block the Simulate and the Calculate message handlers have similar code. One way to be more efficient and reduce errors would be to define a function to calculate the output value and call it in the both the Simulate and Calculate message handlers.

```
Real CalculateOutput(Real inputValue)
{
        InNum_prm = MilesIn;
    if (units_pop == UNITS_KILOMETERS)
        UnitsOut = MilesIn * 1.609344;
    else if (units_pop == UNITS_YARDS)
        UnitsOut = MilesIn * 1760.0;
    else if (units_pop == UNITS_FEET)
        UnitsOut = MilesIn * 5280.0;
    else if (units_pop == UNITS_INCHES)
        UnitsOut = MilesIn * 63360.0;
        return(UnitsOut);
    }
On Calculate_btn
  {
  OutNum_prm = CalculateOutput(InNum_prm);
  }

On Simulate
  {
  InNum_prm = MilesIn;
```

```
UnitsOut = CalculateOutput(MilesIn);
}
```

Integrated Development Environment (IDE)

The ModL Language

A detailed description of ModL constructs and structure

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

ModL is the ExtendSim programming language; it is structured much like C++.

☞ To get the most out of this chapter, you should first read the chapter "ModL Overview". A table comparing ModL to C++ starts on page 27; a comparison of ModL to other languages starts on page 29.

This chapter describes the ModL structure and constructs. It is intended as a reference to the ModL programming language; it is not a programming tutorial.

The best introduction to programming with ModL is familiarity with programming in C or C++. If you are familiar with programming in some other language, that will help as well. If you have not programmed in any language, a beginning programming class or tutorial would probably be a better starting point than trying to learn programming by building ModL blocks.

Names

Names for variables, constants, and functions:

- Can be up to 127characters
- Can have letters, numbers, and the underscore (_) character
- Must begin with a letter or an underscore (_) character
- Are not case sensitive

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

```
Myname    MYNAME    MyName    myname
```

☞ Some names are reserved for system use, as described throughout this chapter.

Data types: definitions and declarations

As discussed more below, there are four main data types in ModL:

1) real or double
2) integer or long
3) string (Str15, Str31, Str63, Str127, Str255 or String)
4) pointertype

Real data types

Real numbers are stored as Extended IEEE floating point numbers with 16 significant digits. 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.

A ModL real/double variable is the equivalent of a double variable in C++. It occupies 8 bytes of memory.

Integer data types

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

A ModL integer /long variable is the equivalent of a long integer variable in C++. It occupies 4 bytes of memory.

☞ 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.

String data types

StrXX may contain up to XX characters and 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 ""ExtendSim"""
```

is evaluated as:

```
It's called "ExtendSim"
```

For a string literal that you want to extend past one line of the code, put a back slash (\) 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";
```

A string variable is the equivalent of an unsigned char x[256] variable in C++. Str255 occupies 256 bytes of memory; smaller string types (StrXX) occupy XX+1 bytes of memory.

ModL strings are not accessed in the same way as strings in C++. To access the contents of a ModL string beyond assigning its contents, you need to use ModL functions.

☞ ModL strings are internally stored as Pascal style strings, with a leading size byte. In most cases you will not have to worry about the internal storage of the string. However, it does become relevant when you pass a string to outside code through a DLL function call. In that case, the ModL functions DLLPtoCString and DLLCtoPString will be useful - see the function list that starts on page 245.

Pointertype data types

This data type contains the address of data that is held in dynamic arrays and compiled equations. A pointertype can be stored in a real, which means that you can store pointertypes in a real type array or in an ExtendSim database for use by the model.

Declaration examples

For the real/double and integer/long data types, the identifiers are interchangeable. The type declarations and constant definitions are set up like they are in C.

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, multiArray [][3][5][10][2];
pointertype p;
```

As you can see from the example above, ModL has fixed and resizable (dynamic) arrays, and arrays can be up to five dimensions.

ModL already has some pre-defined variables that can be treated just like other static variables. These system variables are described in the "ModL Variables" chapter that starts on page 190.

☞ Each variable type consumes a different amount of data, with integers using 4 bytes, reals using 8 bytes, strings using 256 bytes, and pointertypes using 8 bytes. At the same time, there is a limit on the total amount of static data that can be declared. Thus if you define a string array as string x[200] (which uses 200x256 or 51,200 bytes of memory), it will exceed the local or

static data limit of 32,767 bytes (not including dynamic arrays and pointertypes) and the compiler will report an error condition.

Scope of global, local, and static variables

Variables define memory that can store a value or values. There are several types of ModL variables.

Whether a variable is global, static, or local defines what is known as its *scope*. The scope of a variable determines where it can be accessed by code. The following information is important to keep in mind when using variables in block code.

- *Global variables* are pre-defined in the IDE. A global variable's scope is the entire model. In a model, a global variable can be referenced from within any block's code or within equation-based blocks.
- The scope of a *static* variable (a variable declared at the top of a block's code) or *dialog item* variable (the name of a dialog item) is the code of that specific block. This means that a block's static variable cannot be used directly in the code of other blocks or in the equations used in equation-based blocks.
- A *local* variable's scope is just the message handler or user-defined function in which it is declared.

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 automatically initialized and must be initialized in your block code.

Within each block, and also locally within each function in block code, there is a limit on the total amount of static data that can be declared. If you exceed this limit, the compiler will give an error message when the block code is compiled. The total amount of data that can be defined in static variables in the block, and in local variables within each routine, is 32,767 bytes of data. (See "Declaration examples" on page 61 for the amount of memory used by each type of variable.)

This limitation is not an issue for most users because the more common and usually more useful way to allocate large data structures is with dynamic array, global arrays, and database tables. These structure are not subject to, or effected by, the static data limit.

☞ The advantages of using local variables are that they are not saved with the model and only use memory when the function is called. The disadvantages are that the values of local variables are not remembered after the message handler or function is left, and local variables can override static variables of the same name.

Uninitialized static variables can be subtle but serious bugs. The value of an uninitialized variable will take any random number, causing code to not work in unpredictable ways.

Constant definitions

Constants can be of data type real, integer, or string, but not pointertype. A constant definition might be:

```
constant maxPower is 52.5;
```

Note that in the constant definition the data type is implied by the format of the literal value. In the above example, the type is real or double. However:

```
constant xstr is "x"; // quotes cause the constant to be string type
constant maxPower is 52 // data type is an integer or long
```

- For a constant to be real, it must either contain a decimal point or be in E notation
- A string must be enclosed in quotation marks

ModL includes four general-purpose predefined constants (Pi, Blank, true, and false); they are discussed on page 63.

☞ Regardless of where they are defined in the ModL code, constants are always static

Constants are not directly supported in equation-based blocks; use an include or set the value from a Constant block (Value library). For a list of the differences between equation blocks and custom blocks, see page 31.

Constants that are pre-defined

ModL includes four numerical predefined constants:

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

Setting a dialog parameter to BLANK will make it display as an empty field. BLANK values show nothing in a dialog's text field 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 color constants used with the Animation functions.

BLANK and NoValue

The constant BLANK is a special value that represents “no value”. A NoValue can only be represented by a real variable, not an integer. 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 listed in the section “Basic math” on page 207.

☞ To test for a NoValue, do not compare it to BLANK in an “if” statement. Instead, always use the NoValue function:

```
if (myValue == BLANK)      // WRONG! 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 in “Numeric type conversion”, below.

Numeric type conversion

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

☞ Because conversions are computer intensive, it is best to avoid numerical type conversions. This is particularly true whenever there are repeated or often-used calculations.

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.

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.

☞ In an operation between an integer and a real, the result is always a real.

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.

☞ If a real contains a BLANK or NoValue value, the NoValue will be too large to fit into an integer. Thus the integer will 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 for NoValues 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

There is no loss of information or special concerns when converting an integer to a real number.

However, using integer constants for real expressions is dangerous and can give the wrong results.

For example, consider the following code:

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

In this case, z will always equal 0. The integer 1 divided by the integer 2 equals 0 because the result of integer operations cannot be a fraction; it must be an integer. The statement should be instead written as:

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

Integer or real to string

When the equation calls for it, ExtendSim automatically converts reals and integers to string values. If a real variable is a NoValue, the resulting string will be an empty string.

☞ 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 + ".";
```


String to real or integer

The StrToReal function converts a string value to a real number. If the string contains non-numeric characters, the result will be a NoValue.

Do not assign a non-numeric string to an integer. This will assign an unusable value to the integer.

Arrays

You can use real, integer, or string arrays which can be fixed or dynamic. Any array can have up to five dimensions, that is, up to five subscripts or indexes. Array indexes are limited to 2 billion elements. (There are also other array-structure types: linked lists and global arrays. See “Array-like structures” on page 67.)

Array declarations

The number of dimensions and the magnitude of each dimension are 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 104.

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 several functions for sizing dynamic arrays:

Function	Use
DisposeArray	Frees memory when you are finished with the array
DynamicDataTableVariableColumns	Causes the right dimension to be variable. This is only useful for data tables.
GetDimension	Returns the size of the missing left dimension
GetDimensionByName	Returns the size of the missing left dimension based on the block number and the name of the array
GetDimensionColumns	Returns the number of columns in a two dimensional array
GetDimensionColumnsByName	Returns the number of columns in the two dimensional array, based on the block number and the name of the array
MakeArray	Sets the size of the missing left dimension
MakeArray2	Sets the size of the missing left dimension for an array in any block, including a block other than the one calling the code. It allows resizing of dynamic arrays passed in to functions as a string name.

The GetDimension function can be used with any array, not just dynamic arrays. GetDimension returns the value of the first (leftmost) dimension, whereas GetDimensionColumns returns the value of the rightmost dimension in a two dimensional array.

These functions are described fully in "Dynamic and non-dynamic arrays" on page 340. Also see "Variable column data tables" and "Dynamic data table resizing" on page 275.

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.

☞ To pass an entire array, supply only the array name, without subscripts. To instead 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 element; that is, declared such as real a[2]. Assume you have 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]);
```

Array-like structures

Global arrays

There is another kind of array that can be set up using functions and is useful for data needed globally throughout the model. *Global array*s provide a repository 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 accessed and managed through a suite of functions.
- 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 "Using passed arrays to make structures" on page 107 for an example of using global arrays to make structures. The global array functions start on page 342.

Linked lists

ModL has a suite of functions to support a linked list data structure. A *linked list* is an internal data structure that allows the construction and manipulation of complex lists of data. These queue-like, multiple type structures maintain internal pointers between the different elements, speeding movement of elements (sorting) around within the list.

- Each structure element can simultaneously contain any number of integer, real, and string data types, allowing the creation of complex sorted structures.

IDE

- They are faster than their linear equivalent if their internal sorting functionality is taken advantage of, as in a Queue block (Item library).
- They are owned by individual blocks but can be accessed globally from any block.

See “Linked lists” on page 349 for a suite of functions. See the queue blocks in the Item library or the blocks in the Rate library for examples of using linked lists.

Database tables

ExtendSim database tables are array-like structures. For more information about the ExtendSim database see “Working with databases” on page 113. The list of database functions starts on page 318.

Operators

ModL offers a full set of mathematical and logical operators, as shown below.

Assignment operators

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

Math operators

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. If the MOD operator is used on real values, they will be truncated to integers before the operation is carried out.

☞ 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.

Boolean and magnitude operators

Operator	Type	Description
AND or &&	Boolean (logical)	combination
OR or ||	Boolean (logical)	conjunction
NOT or !	Boolean (logical)	inverse
!= or <>	Magnitude	not equal to
<	Magnitude	less than
<=	Magnitude	less than or equal to
>	Magnitude	greater than
>=	Magnitude	greater than or equal to
==	Magnitude	equal to

The three standard Boolean 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.

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.

☞ Putting any expression in parentheses ensures 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 ...
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 and y 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 and y set to 4 because the test occurs at the end of the loop, after x and y have already been 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 modeler's data is bad.
- In the Simulate message handler, it aborts the current simulation if something goes wrong with the calculations. (See the AbortAllSims function to abort multiple simulations.)

ModL also provides two control statements 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 functions

In addition to the many pre-defined ModL functions that are included with ExtendSim, you can define your own functions and override them by re-declaring them. User-defined functions make ModL code more readable and allow you to create tools that can be reused. They are defined and called as they are in C.

User-defined functions have the following form:

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

zero or more statements

Return(value);
}
```

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

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

Limitations

- Unless they are declared in include files, user-defined functions are local to the blocks in which they are defined. To make one or more user-defined functions available for use by multiple blocks, create an include file as discussed in “Include files” on page 81.
- User-defined functions are not directly supported in equation-based blocks; they must be declared in an include file. See the other limitations of equation blocks on page 31.

Defining

User-defined 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 on page 63.)

If you define a function that calls another user-defined function, both cannot be defined first. The solution is to have a declaration of the function before it is actually called in the ModL code:

```
TYPE (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 definition, but instead of braces, it ends in a semicolon.

Exiting

Exit a user-defined function at any point with a Return statement for Void 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
...
```

If Abort is called in a function that was otherwise going to return a value, the code is halted completely and the return value is no longer necessary. That is, the code execution does not return to the calling routine, so the return value of the function is moot.

Overriding user-defined functions

User-defined functions can be overridden by being re-declared any number of times below the first declaration. In this case, the code that is executed is the final version of the routine. In the code pane, the earlier versions of the routine will be colored brown to show that they have been overridden.

Overriding is useful in that include files can have basic forms of functions and message handlers which can be re-declared and overridden in the main block code. See “Include files” on page 81.

Declaring arrays as arguments for user-defined functions

This section shows how to create a function that has arrays as arguments.

If the array size is not fixed, you can declare arrays as arguments to functions with the first (leftmost) 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 a user-defined 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);
...
}
```

For more information

- “Block-to-block message” on page 112 for a method of defining “global” functions
- “Pass by value and reference (pointers)” on page 104 for information on passing arguments to functions
- The user-defined ActiveX Data Object (ADO) functions listed on page 371

Message handlers

Message handlers group code into sections. They interpret messages that come from the simulation, from another block, or from user interaction with a block's dialog. While messages can originate either from the ExtendSim application or from blocks, it is always a block that is on the receiving end of a message.

ExtendSim runs the message handler whenever one of the messages is passed to the block

The format of a message handler is:

```
on MessageName
{
zero or more declarations and/or statements;
}
```

MessageName must be the name of one of the messages listed in the chapter "Messages and Message Handlers". The code of the message handler is contained between the curly braces ("{" and "}") and tells ExtendSim what to do in the specific circumstance. To exit from a message handler before the ending brace, use a Return statement or an Abort statement, as described above.

When you create a 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. No matter what happens, there is a message handler available to perform an action.

Message handler example

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

```
on CreateBlock // The modeler added this block to a model worksheet
{
     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 is added to the model worksheet and thus receives the "CreateBlock" message from the ExtendSim application.

The variable "checkUsed" is a static variable for the block, defined at the top of the code. The variable "myNumber" is the name of a parameter dialog item in the block's dialog. This statement initializes the parameter to 1 when the block is created (placed in the model).

Messages and message handlers are not supported in equation-based blocks. See the table on page 31 for additional differences when using equation blocks.

Overriding message handlers

Message handlers can be overridden by being re-declared any number of times below the first declaration. In this case, the code that is executed is the final version of the message handler. In the block's Script tab the earlier versions of the message handler will be colored brown to show that they have been overridden.

Overriding is useful in that include files can have basic forms of functions and message handlers which can be re-declared and overridden in the main block code. See "Include files" on page 81.

⚠ You can declare local variables at the beginning of a message handler. However, you should not have a global and a local variable with the same name. The 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 variable will not be modified.)

For more information about message handlers, see:

- "Using message handlers" on page 111
- The list of message handlers starting on page 194

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 careful when writing to any of them.

The list of system variables starts on page 190.

Global variables

There are two types of global variables:

- General use global variables have a name that starts with "global". They can be used any way you want.
- Reserved global variables start with the word "SysGlobal". These system globals are controlled by the libraries that are included with ExtendSim and are reserved for use with those libraries.

The list of global variables starts on page 191.

Conditional compilation

ModL supports several preprocessor directives to bracket ModL code. This allows specific parts of the code to be compiled depending on circumstances. For more information, see "Conditional compilation" on page 82.

Integrated Development Environment (IDE)

Programming Tools

Features and capabilities you can use
as you program in ModL

*"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/C++ programs. However, ExtendSim provides tools to help make writing block code easier and safer, as discussed in this chapter.

Script Editor

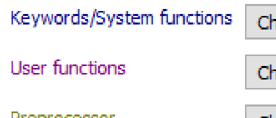
ExtendSim has an internal, integrated script (source code) editor for creating and debugging source code.

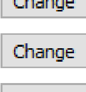
Syntax styling

Syntax styling gives visual cues about the structure and state of a block's code, making it easier to follow the logic.

See the command Edit > Options > Script tab for the following:

- Colorization by syntax, as shown to the right. The colors can be customized using the Change button.
- A Reset Defaults button.
- The ability to specify the font and font size for the text in the Script tab.





☞ When the block's Script tab is the active window, use Alt + O to open the Script tab shown above.

Syntax highlighting

- *Smart highlighting*. Selecting or double-clicking a string causes each instance of that string in that window to be highlighted in green. (To find all the instances of a string in other windows or files, see "Find in Files tab" on page 80.
- *Brace matching*. This feature highlights matching sets of braces (square brackets, curly brackets, or parentheses) to give immediate feedback on misplaced brackets or open-ended code segments. Click the cursor to the right of an opening or closing brace and it, as well as the corresponding brace, will be highlighted in cyan.

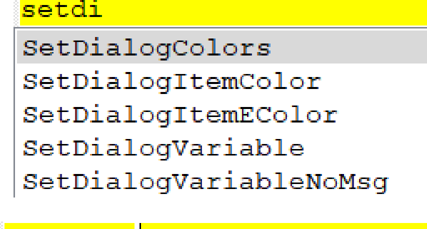
```
{
if(databaseIndex < 1   ||
    address += "<i>";
address += databaseIndex +
address += tableIndex + ":
address += fieldIndex;
data_ttbl[i][TARGET_COL] =
}
```

- *Matching #ifdef, #endif, and #else*. Click on the #ifdef line and it will highlight the closest matching #endif or #else in cyan.
- Show *end of line*. The script editor places invisible characters at the end of each line. If this option is selected in the Edit > Options > Script tab, those characters are made visible.
- Show *white space*. The script editor places invisible characters wherever there is a tab or space. If this option is selected in the Edit > Options > Script tab, those tab and space characters are made visible.

- Show *indentation guides.* This option is selected by default in the Edit > Options > Script tab. It causes vertical lines to be shown in the Script tab's margin, indicating the tab indentation of the code. This is useful for seeing if you've indented the code correctly, especially for long indented sections of code.
- *Code folding*. Click on a code folding marker in the Script tab's left margin to selectively hide and display sections of the code.
- *Auto indentation*. Entering a return at the end of a line takes you to the same indentation level right below that line. Tab or backspace to change the indentation as wanted.
- *Line numbers*. Each line in the code is automatically assigned a number in the left margin of the Script tab. Use the Go To Line button in the Script tab (or the menu command Develop > Go To Line) to find a specified line.

Code completion and call tips

By reducing typos and other common mistakes, *code completion* speeds up the coding process. When you type the first letters for a ModL function or message handler in the block's Script tab, code completion pops up a window with a list of functions that start with those letters. Scroll through the list and double-click to select the desired function.



Once the function has been placed in the script, type an open parenthesis "(" immediately following it. This causes the parenthesis to turn red and *call tips* to display the function's arguments as shown here. The first argument will be bolded. When you enter it, the parenthesis will turn black. As you enter each argument, subsequent arguments get bolded until all are entered.

☞ The opening parenthesis will stay black until you've entered all the arguments and followed them with a closing parenthesis. At that point both parentheses will turn cyan.

Customizing code completion

Code completion and call tips are customizable. All the ModL functions and message handlers are listed in the *application.ini* file located in the ExtendSim/Extensions/CodeCompletion folder. The script editor looks to this text file for the code completion feature.

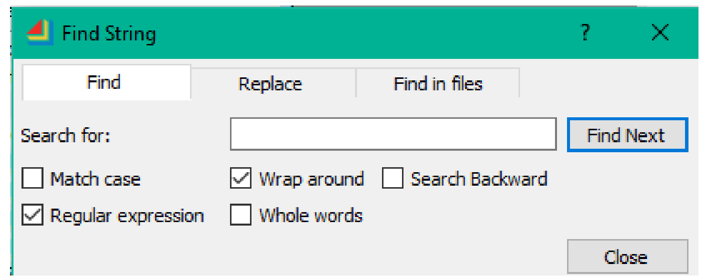
If you define your own functions or includes and want them to be available for code completion, create a new text file that follows the format of the application.ini file. Append the ".ini" extension on the file, then place that new file within the CodeCompletion folder.

☞ Do not change the application.ini file; instead, create a new file. Otherwise, your custom entries will be overwritten when ExtendSim is updated.

Search and replace

If the cursor is in the Script or Help tab, the Edit > Find command opens the Find String dialog for searching and replacing.

☞ The “Wrap around” option is not available when searching the Help tab.



Use the dialog’s Find tab to find a string within the selected window. Or use its Replace tab to replace one string with another within that window.

☞ By default the Find String dialog only searches within the window that has the cursor. For more options, see the Find in Files tab, below.

Find in Files tab

Use the Find String dialog’s Find in Files tab to search through multiple files at once. The options are to search the entire Includes folder, the source file folder, or only through includes used in the block.

This process will find all instances of the searched for string, listing them at the bottom of the dialog. To open a file to the location of the searched string, double-click its name.

Regular expressions

In addition to the normal search options such as matching case, the Find and Replace tabs allow you to search *regular expressions* within the Script tab. This is a sequence of characters that define a search pattern, such as searching for two spaces but only if they occur directly after a period and before an upper case letter.

☞ The regular expressions search option is not available within a block’s Help tab nor within a database table.

Regular expressions are useful in many contexts:

- Validating that a substring meets some criteria, e.g. is an integer or contains no whitespace.
- Advanced search capabilities based on a pattern. For example, match one of the words mail, letter or correspondence, but none of the words email, mailman, mailer, letterbox, etc.
- Replace all occurrences of a substring with a different substring, e.g., replace all occurrences of & with & except where the & is already followed by an amp.

For more information, we suggest *Mastering Regular Expressions* by Friedl.

Enter Selections command

The Edit > Enter Selection command provides an easy method for finding the next instance of some text that is in the code. Selecting the text before giving the Enter Selection command causes the selected text to be placed in the *Search for* box when you give the Edit > Find command.

☞ Alternatively, use smart highlighting— select or double-click a function or message handler in the Script tab, and the script editor will highlight each instance that is in the code.

Miscellaneous script editing features

- The Zoom In button in the Edit toolbar temporarily enlarges the Script pane. Use the Zoom Normal button to return to the default. Note that font size for the script can be permanently enlarged in the Edit > Options > Script tab.
- The Script tab's Functions popup takes you to the first line that uses the selected function or message handler. It also opens the include file if the function or message handler is used in an include.
- The Go To Line button in the Script tab (or the Develop > Go To Line menu command) causes the Script tab to scroll to that line number.
- To use the Develop > Go To Function/Message Handler command, select the name of the function or message handler in the code, then give the command. This is equivalent to holding down the Alt (Windows) or Option (Mac OS) key while double clicking the function or message handler name, or right-clicking on the function or message handler name.
- The Develop menu's Shift Selected Code Left and Shift Selected Code Right commands let you change the tabbed indentation on lines in ModL code. For example, if you copy lines from another block's code that are at a different indentation level, select the lines and use the appropriate command to move them to the correct level.
- You can copy names from the Message pane of a block's structure window for use in the Script or Help tabs by right-clicking on the name and selecting Copy.
- The text on icons and the Help text can have character formatting. Select the text and give commands from the Text toolbar. You can also specify a color for the text from the Graphics toolbar's Fill Color and Border Color tools.

Debugging and profiling

ExtendSim has a built-in source-code debugger, other debugging tools, and a profiling capability that helps to locate code errors or inefficiencies. For more information, see:

- The Debugging Models chapter of the User Reference.
- "Source Code Debugger" on page 172, which works with the code of equation-based blocks as well as with the blocks you build.
- "Debugging block code without the Source Code Debugger" on page 171, which discusses how to use ExtendSim functions to do some debugging.
- "Profiling" on page 170. Profiling generates a text file that shows the percentage of time individual blocks execute during a simulation, indicating if any of your custom blocks need optimization.

Include files

Include files are standard header files. ExtendSim allows you to use include files in ModL code; they can contain all ModL commands such as definitions, assignments, and functions. They can also contain user-defined functions and predefined constants; the functions and constants are accessible by any block that includes that file, including an equation-based block.

Like ModL in general, include files allow for the overriding of functions and message handlers and can contain preprocessor statements for conditional compilation. Include files contain user-defined functions and predefined constants; the functions and constants are accessible by any block that includes that file.

Using include files simplifies programming tasks that are repeated in multiple blocks, such as a library of blocks that use similar variable definitions and functions.

☞ As discussed on page 72, unless they are in an include file, user-defined functions are local to the block in which they are defined.

Creating an include file

To start a new include file, choose the command Develop > New Include File. This opens an untitled include file window.

Type the statements you want in this window and choose the File > Save Include File As command. This saves the include file into the ExtendSim\Extensions\Includes folder.

Naming conventions

For all platforms, the include file's name *must* end with ".h" (as for standard C include files) and the file *must* be either in, or in a subfolder within, the Extensions/Includes folder that resides in the same folder as ExtendSim.

Referencing in a block

To reference an include file from a block's code, enter a line in the code in the format:

```
#include "filename.h" (or #include "subfolder\filename.h")
```

or

```
#include <filename.h> (or #include <subfolder\filename.h>)
```

For example, if the name of the include file is "New_Defs," you would use the command:

```
#include "New_Defs.h"
```

You can have as many include files as you wish, as long as all of them reside in the Includes folder within the Extensions folder.

For an example of how include files are useful, see the Data Import Export block (Value library) which uses an include file named ADO_DBFunctions. The ModL-coded ADO functions for that include file are described on page 371.

☞ Include files are also called header files because they are usually included at the top or head of the file. This is the source of their .h extension.

Conditional compilation

With conditional compilation, segments of code are compiled only if certain conditions are present. Preprocessor directives are used to bracket code segments, causing parts of the code to be compiled only if a particular *symbol* has been defined.

A symbol can be any kind of variable, function name, dialog item name, or constant. As seen below some preprocessor symbols have been pre-defined. You can also define your own symbols using #define, as discussed below.

Preprocessor directives

ModL supports several preprocessor directives to bracket the ModL code that you want conditionally compiled:

- #ifdef *symbol*. If the symbol is defined, compile the code following the #ifdef until you get to a #else or a #endif.

- #ifndef *symbol*. If the symbol is not defined, compile the code following the #ifdef until you get to a #else or a #endif.
- #endif. Marks the end of the current preprocessor directive.
- #else. Used with #ifdef or #ifndef to give an alternative set of code to be compiled.
- #define *symbol*. Used to define your own symbols when you need to change your code based on the existence of that new definition, for example within an include file. After defining a new symbol, you can use it in the #ifdef and #ifndef directives. The syntax of the #define is:

```
#define myNewSymbol // define a new symbol
```

Pre-defined preprocessor symbols

The following preprocessor symbols have been defined in the compiler:

- Compiled_Debug. Defined only if a block has debugging code on (Develop > Set Breakpoints and Add Debugging Code). It is only in release 10 and later.
- ExtendSim_10. This symbol is defined only if the application is release 10 or later.
- Platform_Windows_Defined_Symbol.
- Platform_Macintosh_Defined_Symbol

Examples

For example, you can create an include file that has code to modify a dialog item. But if that dialog item isn't used in a particular block, the code will be ignored by the preprocessor directives.

The following is an include file generalized for different blocks.

```
...
void AFunction(real anArgument)
{
...
#ifdef myDialogItem  // Only if myDialogItem is present
     myDialogItem = anArgument;  // display result in the dialog
#endif
}
```

A common situation is to use #ifdef to compile a message handler only if a dialog variable exists. For example, an include file may be used by many different blocks, only some of which have the dialog variable MyValue_prm. To prevent the compiler from giving an error message when a block does not include the dialog variable, add the following to the include file:

```
#ifdef MyValue_prm
     On MyValue_prm
     {
          //message handler for dialog variable
     }
#endif
```

External source code control

ExtendSim supports a mechanism for saving the source code of individual blocks or libraries of blocks as external files. This external source code feature is useful for situations where multiple developers work on the code of blocks and/or libraries at the same time.

Normally the code of blocks is saved in the library file, along with the blocks' dialog item definitions and icons. When blocks or libraries are recompiled with the external source code option turned on, the library file does not contain the master source code. Instead, the source code is saved in a separate subfolder inside the Libraries folder. It can then be used with a separate source code control or management application.

Externalizing source code for a block

- Open the block's structure and go to the Script tab
- Choose the command Develop > External Source Code
- Close the block's structure window
- In the dialog that appears, choose *Save, compile if needed*

This causes the block to be recompiled with its source code in an external file.

☞ If source code has been externalized, the green initials CM (for code management) will be displayed on the block's icon in the library window.

Restoring the block's code to its structure

- Open the block's structure and go to the Script tab
- Unselect the command Develop > External Source Code
- Close the block's structure window
- Save and compile the block

Externalizing source code for an entire library

- Give the command Library > Library Tools > Add External Code to Libraries
- In the dialog, select the libraries you want to have external source code
- Click Add External Code

This recompiles the library and causes block source code to be placed in an external file.

☞ In the library's Library Window, the green initials CM (for code management) will be displayed to the right of each block's icon.

Restoring the source code to the library file

- Open the Library Window for the desired library
- Give the command Library > Tools > Remove External Code in Open Library Windows

The consequences of saving the code externally

Using the external code option for a block will:

- Automatically create a folder named "Source" within the Libraries folder, if the Source folder does not already exist.
- Place a folder with the name of the library inside that Source folder, if that library folder does not already exist.
- Create a source code text file for each block that has been recompiled with external source code and places it in the library name folder. Each file will be named with the block name and end with a .cm extension.

- Cause the green text *CM* (for "code management") to appear on the icons in the library window, for each block that has been recompiled with external source code.
- Create a backup file, with the extension .ck, each time the block is recompiled.

Using the external source code with code management software

The primary reason for using the external source code option is in conjunction with a separate source code control or management software. This allows multiple people to work on the code of the blocks at the same time. For example, Subversion is an open source version-control system that is available for download from the Web. The basic structure would be a situation where Subversion or some other source code control software would maintain an archive on a server and synchronize the source folders on each modeler's machine with that central archive.

Because the source code for each block is saved in a separate .cm file, which is just a text file, the source code control software can maintain the file and synchronize any changes made by the modelers with what is in the central archive.

Cautions when using the external source code feature

Unlike block code, block dialogs, icons, and help text are not editable by multiple developers at the same time.

Managing non-code parts of blocks

With external source code turned on, the source code for each block is saved as a text file. This facilitates the management of block code by multiple developers using source code control software. However, the block dialogs, icons, and help text are not saved in the external files. Instead, they are saved in the original library file.

When changes need to be made to block dialogs, icons, or help text, the library should be "locked" using the source code control software. That way, no one else can modify those parts of the block while they are being changed. After the changes have been made, unlocking the library releases the blocks so other developers can work on them.

Sharing libraries with others

When you have recompiled a library with the external code option, you need to be careful about how you manage and share that library. For example, if you give the library to someone else without giving them the source code files, they will receive warning messages when opening the structures of the blocks. Before giving the library to another person, recompile the library with the external source code option turned off.

Extensions

Extensions are files of various types, such as text, sound, image files, or DLLs. When you installed ExtendSim, you also installed an *Extensions* folder in the same folder as the ExtendSim models and libraries. The Extensions folder contains various files stored in subfolders, such as DLLs, Includes, and Pictures. In addition to the files shipped with ExtendSim, you can add your own extension files to the Extensions folder.

☞ Extensions should be stored in the appropriate subfolder. If there is no subfolder for the type of extension you are adding, put it at the top level of the Extensions folder.

Supported file types

ExtendSim supports the following types of files:

- Text files, which are used for includes and code completion purposes

- DLLs
- WAV files
- Various kinds of image files, such as BMP, JPG, PNG

Macintosh resource files converted using the ExtendSim MacWin converter utility are supported as pictures for backwards compatibility with earlier versions of ExtendSim. The different kinds of extensions are discussed individually later in this chapter.

Naming

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.

For more information, see

- DLLs, below.
- “Sounds” on page 89
- “Sounds” on page 89
- “Picture and movie files” on page 90
- “Customizing code completion” on page 79

DLLs

As discussed in “Accessing code from other languages” on page 42, ExtendSim provides sets of functions that allow you to call, from within a block’s ModL code, segments of code written in a language other than ModL. 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.

For ExtendSim 10 or later, the DLL must be built for 64-bit execution

Overview

DLLs are libraries of code written and compiled in any language. Their standardized interface provides a method for linking between ExtendSim and languages other than ModL. In order to access these code libraries, they must be stored in the ExtendSim Extensions directory, discussed on page 85.

The ExtendSim 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 interface

The DLL functions allow you to access existing Windows DLLs. Because they have variable argument lists, these functions allow you to call almost any existing DLL. See the DLL functions on page 245 for a list of functions and specific information about this interface.

Turning code into a DLL

Taking existing code and turning it into an DLL involves the following:

- Write or edit the DLL code using a Windows compiler that is capable of compiling DLLs. For ExtendSim 10 or later, the DLL must be built for 64-bit execution.

- Modify the DLL code by adding the DLL calling interface.
- After compiling the code, you will have a DLL file. Place this file in the ExtendSim Extensions\DLLs directory.
- Restart ExtendSim. 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:

- In the argument list, variables (other than strings or arrays) that are passed from the ModL code to a DLL are passed by value.
  - Reals are 8-byte double precision
  - Integers are 4-byte long integers (converted to 8-byte for the DLL)
  - Pointertypes are 8-byte integers
- In the argument list, strings and arrays are passed to DLLs as 8-byte pointers to data that has been allocated by ExtendSim. Modifications to that data will affect the original information in ExtendSim.
  - Arrays that are passed to a DLL come through as pointers to the original data in ExtendSim. Accessing and modifying the data is fine, but you should not try to resize the pointer. If you do, ExtendSim 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. To convert strings to C strings and back again, see the DLLCtoPString and DLLPtoCString functions in the function list “DLLs” on page 245.
- The most common problem associated with building and using a DLL is making sure that the names of the routines that you want to call are exported and that they are exported *without* Name Mangling or Name Decoration. Name Mangling is an option for how names are exported from a DLL; it adds information about the arguments to the exported name.

☞ When building a DLL for use with ExtendSim, the Name Mangling option should be off.

DLL example for Miles block

The following code calls a DLL that performs the same function as the ModL code (shown on page 51) that you used to build the Miles block.

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

IDE

```
long proc;
on Calculate_btn // Display the values
{
proc = DLLMakeProcInstance("convert");
if (proc > 0) // Check if proc>0 before you make the call
     OutNum_prm = DLLDoubleCFunction(proc, inNum_prm,
          kilometers_rbtn, yards_rbtn, feet_rbtn, inches_rbtn);
else
     Beep(); // Couldn't find the DLL function "convert"
}
// This message occurs for each step in the simulation.
on simulate
{
InNum_prm = MilesIn;/* Display the value input by setting the dialog
               variable "InNum" to the input connector value */
proc = DLLMakeProcInstance("convert");
if (proc > 0)// Check if proc>0 before you make the call
     unitsOut = DLLDoubleCFunction(proc, inNum_prm,
          kilometers_rbtn, yards_rbtn, feet_rbtn, inches_rbtn);
else
     Beep(); // Couldn't find the DLL function "convert"
OutNum_prm = UnitsOut;  /* Display the value output by setting the
          dialog variable "OutNum" to the output connector value */
}
```

The code of the 64 bit DLL, written in C++, is:

```
double _export _fastCall 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);
}
```

DLL example for DLL Add block

This section demonstrates how to call a DLL, written in different languages, from within an ExtendSim block using DLLMakeProcInstance and the DLL calling function DLLDoubleC-Function().

DLL Add block

The *DLL Add Block* (Libraries/Example Libraries/ModL Tips) calls a DLL (Add.dll) that adds together the two numbers entered in the block's dialog. By default the block calls a DLL written in C++; as required, the DLL is located at ExtendSim/Extensions/DLLs.

ExtendSim ships with the source code, projects, and Add.dll's written in 4 languages (C++, C#, Python, and VB.Net). The project folders are located at Documents/ExtendSim/Examples/How To/Developer Tips/DLLs. These DLLs all do the same thing (add two numbers together); they are just built in different languages. By changing which DLL is in the ExtendSim/Exten-

sions/DLLs folder, you can explore how DLLs written in other languages are created and called.

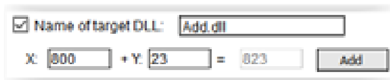
☞ Also see the DLL Add block's Help for hints when calling DLLs.

Calling the C# DLL

By default the *DLL Add Block* calls a DLL written in C++. To instead have the block call the C# DLL:

- Quit ExtendSim
- Go to the ExtendSim/Examples/How To/Developer Tips/DLLs folder
- Open the folder named *AddDllCSharp-Binary-Src-006 unblocked*
- Inside that folder, open the folder named *Binary*
- Copy the three files (add.cs.dll, add.dll, and vcruntime140.dll) from the Binary folder and paste them into the ExtendSim/Extensions/DLLs folder, making sure that the existing add.dll gets replaced.

- Launch ExtendSim and either:
  - Put the block named *DLL Add Block* (from the Libraries/Example Libraries/ModL Tips folder) onto a model worksheet.
  - Or, open the model named *Add.mox* (located at the ExtendSim/Examples/How To/Developer Tips/DLLs folder).
- In the block's dialog, enter numbers for X and Y, then click the Add button.



- See the block's Help for additional information.

Calling other DLLs

To use the other DLLs supplied with ExtendSim, remove any Add DLL files you've placed in the Extensions/DLLs folder. Then follow the same procedure as above for the C# DLL, copying the appropriate files from the project's Binary folder to the ExtendSim/Extensions/DLLs folder:

1) For Python, copy the files named "Add.dll", "addm.py", and "python37.dll"
2) For VB.Net, copy the files named "add.dll" and "add.vbnet.dll"

Target the DLL by name

You could use the DLLMakeProcInstance (string procName) function to search for the procedure. However, for situations where two different DLLs have the same procName, or to be really sure which DLL you're calling, it is safer to call the function DLLMakeProcInstanceLibrary. This function is similar to DLLMakeProcInstance except it has a libraryName argument so you can specify which specific dynamic-link library (DLL) should be opened and searched for the procedure.

Sounds

Depending on your operating system, ExtendSim has access to several sounds. You can play sounds using the Notify block (Value library) or by calling the PlaySound function in the code of a block.

There are two sources of sounds that ExtendSim can access:

- ExtendSim contains a “click” sound extension.
- You can also use .WAV files created by other applications or obtained from user groups.

You must copy the sound file into the Extensions directory, as discussed on page 85.

Picture and movie files

Pictures and movies are used for specialized 2D animation. See the functions in “2D Animation” on page 250, for more detail. To see an example of using pictures and movies for animation, see “Showing a picture on an icon” on page 134.

Naming conventions and limitations for pictures

You can have any number of pictures in the Extensions folder. Pictures with names that start with a “@” will not show up in the block’s animation tab popup menu. This prevents the animation tab menus from being filled with other, larger types of pictures that are not suitable for animation between blocks.

As mentioned in “Extensions” on page 85, pictures must be stored as a file in the Pictures subfolder of the Extensions folder. The AnimationPicture function will recognize most picture formats.

Protecting libraries

ExtendSim normally keeps both the ModL code and compiled code in the block; if you remove the ModL code, the compiled code is still there. When you build custom blocks and do not want others to have access to your block code, you can protect your libraries by removing the ModL code of the blocks. When a modeler attempts to edit a protected block, ExtendSim displays a dialog message that the block is protected and cannot be opened. The structure of the protected block is 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 modelers 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.

☞ After a library is protected, *you can never un-protect it*. Thus it is good practice to make a copy of the original and store it on some other media.

The steps for protecting a library are:

- Select the command Library > Library Tools > Protect Library.
- When you give the Protect Library command, ExtendSim 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.
- The Create New Library dialog allows you to rename the library. However, since models will be expecting the original library name, it is suggested that you leave the library name as is. The protected library will have the extension “.lbrpr” rather than the normal “.lbr”, so your original library won’t be overwritten.

- When the library is saved, ExtendSim protects the library's 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 the structure of a block in the library.

Integrated Development Environment (IDE)

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

This chapter focuses on using ModL functions to create custom blocks and simulation effects in ExtendSim.

Data source indexing and organization

Communicating between various types of data sources has been greatly assisted by standardized technologies, such as the ability of diverse applications to exchange data through a standard text file format. However, it is important to keep in mind that each standard has its own conventions. This can cause data-confusion when transferring data from one type of source to another.

Transferring data between a data table and a spreadsheet

ExtendSim data tables have zero-based indexes and are organized by row and column. Spreadsheets are also organized by row and column, but they are one-based. When transferring data from an ExtendSim data table to an Excel worksheet, the row and column numbers for Excel must both be increased by 1 compared to their location in the ExtendSim data table.

☞ The first row in an ExtendSim data table could be labeled as 0 or as 1. However, no matter how the row is labeled, the index for that first row is still 0.

Transferring data between a spreadsheet and a database

Databases are one-based like spreadsheets, but are organized by fields and records (equivalent to columns and rows) rather than being organized by the spreadsheet convention of rows and columns.

Indexing and organization

The following table lists indexing and organization conventions for different data source types.

Data Source Type	Indexing	How organized
ExtendSim Data Tables	0-based	Row/Column
Spreadsheets	1-based	Row/Column
ExtendSim Databases	1-based	Column/Row (Field/Record)
External Databases	1-based	Column/Row (Field/Record)
Arrays	0-based	Row/Column
Text Files	N/A	Row/Column

Equation block programs

The equation-based blocks are: Equation, Optimizer, Query Equation (Value library); Equation(I), Queue Equation, Query Equation(I) (Item library); and Buttons (Utilities library). The ExtendSim User Reference discusses how to use the equation-based blocks and how to debug them.

Equation blocks can handle more than just the definition of an equation—they are capable of compiling and executing complex ModL programming logic. Think of the equation blocks as a method for including small programs on the fly without having to create a new block.

You can also use the functions described in “Equations” on page 218 to create your own blocks that take in equations and even allow for debugging. For example, you might want to do this when building scientific or engineering blocks and can’t predict ahead of time what formulas you will need.

To allow larger equations or programs into a block’s dialog, see “Equations” on page 218 for a list of equation functions that support up to 32000 characters in a dialog item. Also, see the Equation block (Value library) for an example of using a Dynamic text item and a syntax coloring window to edit a user-entered equation.

Equation-based blocks can call any of the ExtendSim built-in functions. However, user-defined functions and procedures cannot be defined directly in an equation-based block; they must be defined in an include file. For a list of the other differences between equation-based blocks and the custom blocks you might create, see “Differences between equation blocks and programmed blocks” on page 31.

Working with dialogs

With a bit of creativity, you can make dialogs that provide a great deal of information about the state of a simulation or that help you debug models-in-progress.

In addition to the following information, modifying and creating dialogs, as well as ModL code interaction with dialogs, is described in more detail in “Accessing dialog items from a block’s code” on page 35.

Changing text in dialogs

The Miles block example on page 44 showed the usefulness of keeping a dialog open and watching the numbers in the entry boxes change as the simulation progresses. Dialogs can also be used for displaying text or messages that might change during the simulation.

Changing text as the simulation runs

Changing the text displayed in a dialog is often more useful than displaying alerts since alerts stop the simulation until the modeler clicks one of their buttons.

For example, assume that:

- You are modeling a factory that has three shifts, where the name of the shift (day, night, swing) changes depending on the current time.
- You have a block with a static or editable entry box named *Shift* that you want to modify.
- 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 text in response to a user's action

ModL lets you change the text and titles of dialog items at any time. This allows you to decide what to show based on what is done in the dialog. An example of this is the Random Number block (Value library), which displays different parameter labels depending on the statistical distribution that is selected.

When a control item (radio button, switch, meter, etc.) 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.
- ExtendSim 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 (for gasoline) or cetanes (for diesel fuel). 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 modeler
on DialogOpen
{
   // Static labels are not "remembered" when dialog is closed
   //  so, when the modeler 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 38 and "Static text (label)" on page 39.

☞ 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.

Hiding/showing dialog items

As discussed on page 19, by default dialog items are visible in at least one tab; they may be hidden by unchecking the Visible checkbox. Dialog items where Visible is unchecked are shown in red.

You can also choose to dynamically show and hide dialog items depending on a user action or on the value of other dialog items. You do this using functions (such as HideDialogItem) in the list that starts on page 267. When the dialog is opened by the modeler, you use a DialogOpen message handler to show and hide certain items depending on the values of the controlling items.

For example, the Decision block (Value library) has a parameter field for Relax that appears if Use Hysteresis is checked. For more complex dialogs, see "Moving dialog items".

In some cases you may want a dialog item to not be visible ever. This is common when you don't want to use a dialog item anymore, but shouldn't delete it because the block is already used in models. In this case, just uncheck the Visible checkbox.

Use caution when deleting a dialog item if the block is being used in any model. Deleting the item could disrupt the order of the dialog's data. The data will have to be reentered for each instance of that block in all models that use it. Instead of deleting the dialog item from a block used in a model, hide it by unchecking the Visible checkbox in its properties window. (Since they don't store data, text frames and static text labels may be safely deleted.)

When you copy or duplicate dialog items

You can copy/paste or duplicate dialog items within a tab, from one tab to different tab within the same block, or from one block to another. To do this, select the dialog item, then give the Duplicate (or Copy and Paste) commands.

When you duplicate or copy/paste a dialog item, its Label stays the same but a number is appended to the original dialog item's Name. Unless you change it, the new name (with the appended number), is how the copy of the dialog item will be referenced in ModL code. The name and Label can be changed by double-clicking the dialog item to access its definition.

Moving dialog items

- To manually move a dialog item within a dialog tab:
    - Either select and drag it to the new location within the same window or tab
    - Or, double-click it and change its X and Y coordinates in the properties window
- To manually move a dialog item from one tab in the Dialog tab to another, first select the item, then use the Develop > Move Selected Items to Tab command. Then select the tab to move the item to.
- You can also write code that moves a dialog item depending on a block's settings. This is often easier and more organized than placing all the dialog items on top of each other and then hiding and showing them depending on the setting. For instance, the parameter fields in the Random Number block (Value library) move based on which distribution the user

selects. The DIPosition functions, as well as DIMoveTo and DIMoveBy, can be used to move dialog items. These functions are included in the list that starts on page 267.

Resizing dialog items

Many dialog items can be resized. Do this either through code or in the Dialog tab:

- Either select the item and drag one of its handles (the squares in the corners of the item)
- Or, double-click the item and change its W (width) and H (height) coordinates
- Or, through the code using the function DIPositionSet.

Changing the title of a radio button or checkbox

The function DITitleSet is useful for setting the title of a radio button or checkbox. ModL code can thus change the title, and meaning, of the control at any time.

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 that have a specified commonality.

☞ In addition to the GetDialogVariable function, see the GetStaticVariable function.

For example, as seen in the Get-Set Dialog Variable model located at Documents/ExtendSim/Examples/Developer Tips, you can build a block that can cause specified Activity blocks (i.e. Activity blocks whose labels include a specific wording, such as ABC) to double their entered delay value. Below is sample code that does this:

```
on doAllButton     // modeler clicked 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_prm", 0, 0);

        if (paramStr != "")    // not empty means it was found
            {
            value = StrToReal(paramStr);
                               // now set it to double value
            SetDialogVariable(i, "waitDelta_prm", value*2.0, 0, 0);
            }
        }
    }
}
```

Remote access to dialog variables

Sometimes it is useful to allow a modeler to use a stand-alone block to get or set the value of a dialog variable in a remote block. For instance, this is how modelers set model factors and get model responses using the Scenario Manager block, discussed in the User Reference.

In ExtendSim there are three mechanisms for interfacing dialog variables between a stand-alone block and a remote block:

1) Manually enter the remote block number's and dialog variable name into the stand-alone block
2) Clone-drop, using the Clone tool to drag the parameter onto the icon of the stand-alone block
3) Shift-click a dialog parameter and select from the popup menu of available options to add that parameter to the stand-alone block. This is especially helpful when one or more of the blocks is within a hierarchical block.

The Find and Replace (Utilities library), Optimizer and Scenario Manager (Value library), and the Statistics block (Report library) reference dialog variables in other blocks. And all of the Value, Item, and Rate blocks have interfaces that support this feature. These interfaces are also available to developers, and taking advantage of them requires only minor additions to your custom block code.

Custom remote blocks interfacing with a stand-alone block

The blocks in the Value, Item, and Rate library have code that allows a stand-alone block to get or set their dialog variables; they support all 3 of the above options. Your custom block can also take advantage of this interface, allowing a stand-alone block to interface with your block's variables:

- Options 1 and 2 will automatically be implemented without any modification to your blocks.
- For option 3, you need to add the following code to your custom block:

▸ Include MouseClick v10.h

```
#include "MouseClick v10.h"
```

▸ Add the following code

```
On DialogClick // called whenever the modeler clicks a dialog item
{
if(do_keydown_mouse_click())
return;
}
```

IDE

☞ If you have additional code in this message handler, put it after this code.

Custom stand-alone block referencing remote dialog variables

If you create a stand-alone block that will reference (get or set) remote dialog variables in other blocks, you can implement any or all of the above methods for collecting the information from the remote block, as discussed below. (While it is also possible to develop your own interface, using one or more of the ExtendSim interfaces will be easier and more consistent with existing blocks.)

Manual data entry

Implementing a manual method for referencing from the list in a custom stand-alone block depends on your particular implementation, interface, and needs. Some blocks, such as Find and Replace (Utilities library), only work with one remote dialog variable at a time. Other blocks, such as Statistics (Report library), have a list of remote dialog variables. For coding examples, look at one of those blocks.

Clone drop

Adding a clone-drop interface to a custom block requires a *DragCloneToBlock* message handler. This is called whenever a clone is dragged to the icon of that block.

Inside of this message handler, call *GetDraggedCloneList*. This function requires two arguments – an integer dynamic array and a string dynamic array. It returns the number of clones (usually 1) dropped onto the block. You then use this information to reference a dialog item in another block.

Shift-click

The shift-click feature is more complicated to implement because it requires using a reserved database (discussed on page 114) and an include file. The reserved database (_leftClickDB) provides the necessary information for the shift-click action. The include file (MouseClick.h) has two functions defined in it that are required for managing the reserved database:

1) RegisterBlockInLeftClickDB(String blockNumberDialogVariable, String staticStringVariableForRemoteBlockName, String sStaticStringVariableforRowAndColumn, String popupMenuLabel, integer optionNumber). Registers a block in the database of blocks that add a popup to a shift-click action. This is called in the CreateBlock, PasteBlock and OpenModel message handlers. See the MouseClick include file for an explanation of the arguments.
2) UnRegisterBlockInLeftClickDB. Removes a block from the database of blocks that add a popup to a shift-click action. This should be called in the DeleteBlock message handler.

Coding for Shift-click
The key to the Shift-click functionality is a hidden dialog variable in the stand-alone block. When this dialog variable is set by the remote block, the stand-alone block receives a message that it has been sent a reference to a dialog variable in the remote block. The name of the hidden dialog variable is stored in a table in the reserved database (_leftClickDB) by the RegisterBlockInLeftClickDB function.

The information sent by the remote block must be processed in the message handler for the hidden dialog variable. This information has been set in static variables in the stand-alone block by the remote block. The names of the static variables are set in the RegisterBlockInLeftClickDB function. The information includes the block number, dialog variable name, and database table's row and column. These values can be found in the Dialog and Static variables that are arguments to the RegisterBlockInLeftClickDB function.

In the following code example, the stand-alone block gets:

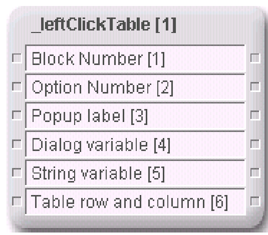
- The block number for the remote block in the dialog variable AddFactor_prm
- The name of the remote block's dialog variable in the static string variable DialogVarName
- The row and column of the remote dialog variable in the RowColumnDialogVar static string variable

And the popup menu is labeled "Scenario Manager: Add Factor". Because the Scenario Manager has two options (one for adding a factor and one for adding a response), this is option 1.

```
On OpenModel
{
RegisterBlockInLeftClickDB( "AddFactor_prm","DialogVarName",
     "RowColumnDialogVar", "Scenario Manager: Add Factor", 1);
}
On AddFactor_prm
{
// Receive message from remote block and process hidden variables
// AddFactor_prm is the block number of the remote block
// DialogVarName is the dialog variable in the remote block
// RowColumn is the row and column in the remote block
}
```

Shift-click example

When the Optimizer block (Value library) is added to a model it creates the reserved database *_leftClickDB*. The entries in the database table cause options, such as *Optimizer: Add Parameter*, to be added to a menu that appears when an appropriate dialog item is Shift-clicked.



Through block code, the Shift-click action causes the selected variable to be used in the Optimizer block. Using this type of architecture makes it easy for developers to add their own block options to the Shift-click menu.

Working with connectors

Most blocks have connectors. You add and change connectors using the connector tools in the Icon Tools tool at the right of the ExtendSim toolbar.

☞ For information about connector types, options, and names, see the writeup that starts on page 21.

Variable connectors

By default new connectors are added as a normal (single) connector. This works well for simple blocks, but often a block will need several inputs or outputs.

As discussed in "Connector options" on page 21, when you select a type of connector (Value, Item, etc.) it is by default a normal connector. You can then change the connector to be variable. Variable connectors act like a row of normal connectors, where the row can be expanded or contracted to provide a required number of connectors. To work with variable connectors, see the writeup on page 35 and the functions on page 264. For an example of how variable connectors are implemented, see the Math block (Value library).

Initializing connectors

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. All connectors are initialized by ExtendSim to 0.0 after the CheckData message handlers are executed. To initialize connectors *before* the simulation runs, do it within the InitSim or PostInitSim handler.

For example, to initialize a connector before the simulation runs:

```
On InitSim     // Or use On PostInitSim
     {
     conOut = initialValue;
     }
```

The blocks in the Item library initialize their connectors in the InitSim handler.

Deleting connectors or changing connector types

ExtendSim 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.

For blocks used in existing models, changing connector order could cause a problem. When you only need to ***change*** connector types, *do not delete the connector*. Instead, simply select it and click on the correct connector tool. If you have to delete a connector on a block that is used in a model, carefully examine the block's connections afterwards.

Bidirectional connectors

All ExtendSim connectors are bidirectional. This is 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 ExtendSim does not allow multiple output connectors to be connected together, you could use only input connectors for all the blocks in the network.

Or 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, 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, which is located in the Documents/ExtendSim/Examples/How to/Developer Tips folder. In that 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
     {
                          // Out of power. Tell when this occurred.
     UserError("Brownout occurred at time= "+CurrentTime);
     abort;               // Stop the simulation.
     }
```

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. ExtendSim provides lots of features and functions that use arrays, especially regarding discrete event modeling. Note that, while ModL does not directly support arbitrary user-defined structures, it supports a rich linked list structure (see "Array-like structures" on page 67 and "Linked lists" on page 349) and it emulates other types of structures using arrays of arrays. You will find that working with arrays in ExtendSim is simpler and safer than using the data structures that are available in C.

☞ As shown in the table in "Data source indexing and organization" on page 94, arrays use zero-based indexing.

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.

☞ There is no overhead for using arrays.

Type	Memory used
Real	8 bytes (double)
Integer	4 bytes (long)
String or Str255	256 bytes per string containing up to 255 characters
Str15	16 bytes per string containing up to 15 characters
Str31	32 bytes per string containing up to 31 characters
Str63	64 bytes per string containing up to 63 characters
Str127	128 bytes per string containing up to 127 characters

The total of all static arrays (non-dynamic arrays) and variables in a block, including any static data tables in the block's dialog (which are real arrays, 8 bytes per element), cannot exceed 32,767 bytes. Dynamic data tables are not included in static memory allocation. See "Block data tables" on page 274.

When a user-defined function 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 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 does. When you pass by reference, however, the function 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 341. The Passing Arrays model in the Documents/ExtendSim/Examples/How To/Developer Tips folder 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.

☞ If you pass arrays, those arrays can only be resized or disposed of in the same block where 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 (Item 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 as easily as you can single values, but it is not 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 `theArray` variable 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 that are available to all blocks. If you want an array to be globally accessible, 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 use this technique.

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.

☞ 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 that 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.

☞ The following is an example of what NOT TO DO:

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;
}
```

The above code 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 160) 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, except for linked lists (see “Linked lists” on page 349). 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 structure-
Array.
          // 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 67, global arrays provide a repository for model-specific data; they are accessed and managed through a suite of functions listed on page 342. 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 index 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 myGlo-
balArray.
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:

IDE

```
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.

Working with linked lists

As discussed on page 67, linked lists are complex structures that can enable sophisticated sorting rules. The concepts behind linked lists are beyond the scope of this manual. See “Linked lists” on page 349 for a basic strategy in working with linked lists. Also, see the Queue blocks (Item library) for actual linked list code.

Using message handlers

☞ Messages and message handlers were discussed on page 33 (introduction) and page 75.

ExtendSim uses a sophisticated messaging architecture to signal blocks into action. For instance, when you add a block to a model, ExtendSim sends the “CreateBlock” message to the new block. If the block’s code contains a message handler for the “CreateBlock” message (that is, a bracketed set of lines that are preceded by “on CreateBlock”), the code is executed; if not, nothing happens.

While messages can originate either from the ExtendSim application or from blocks, it is always a block that is on the receiving end of a message. When you run a simulation, some messages are sent to all blocks; others are sent only to a specific block. For instance, the “CreateBlock” message is sent only to the block that was added to the model. However, the “InitSim” message, which tells the blocks that a simulation is starting, is sent to all blocks.

☞ The messages that a block’s dialog items uses are listed in the Dialog Item Names pane in the block’s structure window. Use the Copy and Paste commands to copy the message names from the pane to use in ModL code.

Categories of messages

While there are dozens of messages, they could be thought of as falling into one of three categories:

1) Messages that are sent to a block when the modeler interacts with the block’s dialog. For instance, when the modeler clicks the block dialog’s Cancel button.
2) ExtendSim allows blocks to “call” other blocks by sending them a message, whether the blocks are connected or not. Blocks use this feature to cause an action in another block, such as having it perform a calculation or open a plot. For example, the UpdateStatistics message is typically sent to an Activities block by the Statistics block (Report library) when its statistical variables need to be recalculated and updated.
3) Messages that are typically sent from the application to one or more blocks. For instance, when a block is added to the model, while the simulation is running, during interaction with an ExtendSim database, and so forth. For example, the LinkContent message sends the message that data has changed in an ExtendSim database.

☞ A complete list and description of ModL messages is in the chapter “Messages and Message Handlers” that starts on page 193.

Message sent during user interaction with dialog

This example shows what happens when the modeler clicks a button in the dialog. The dialog item name of the button is “MyExportButton”. (The text label of the button is “Export”; it is not used in the code.)

```
// Sent when the modeler clicks a button in the block's dialog
on MyExportButton // The modeler clicked the Export Data button
{
MyExportFTP(); // Call my function to export the data to the web
}
```

☞ For examples of code for each dialog item, see “Dialog messages” on page 36.

Block-to-block message

When a button is clicked in the calling block, its code sends a message to the receiving block and causes that block to return a value. Global variables are used to pass arguments to the called block as well as to get results from the called block’s actions.

See the Item library for examples of blocks sending messages using connectors.

Calling block

```
on Button  // called when the block’s Button is clicked
{
// globalInt0 contains the block number of the receiving block
// UserMsg0 is the message handler that calculates it
SendMsgToBlock(globalInt0, UserMsg0Msg);     // Call it
myResult = global1;             // Get the result of the call
}
```

Receiving block

```
on UserMsg0  // called from the calling block
{
global1 = 123.5; // assign value to global1
}
```

Message sent by the application

The following code uses the CheckData message that is sent by ExtendSim to all the blocks in a model at the beginning of a simulation. If the block has a CheckData message handler, this message tells the block to check the validity of its data before the simulation starts.

```
// Sent by ExtendSim to allow checking data before simulation
on CheckData
{
if (myParm < 0.0)    // this parameter should not be negative
     {
     UserError("Data in " + MyBlockNumber() + " can't be negative.");
     Abort;    // Stop simulation and select the offending block
     }
}
```

Working with databases

As discussed in the User Reference, an ExtendSim database provides a repository for model-specific data.

☞ See also the separate document ExtendSim Database Tutorial and Reference.

Using the database API to read and write

ExtendSim databases can be accessed and managed through the Read and Write blocks in the Value and Item libraries. You can also use the database API (see “Database functions” on page 318) to create databases via execution of a block’s code.

The following is an example of how to create and access a database to store output from a model:

```
// Declare static variables
integer databaseIndex;// index value for the output database
integer tableIndex;// index value for the output table
integer fieldIndex;// index value for the output field
integer recordIndex;// index for setting our record values

on initsim// create the database, table, field if not there
{
// Creates database parts only if it doesn’t exist already
DBDatabaseCreate(“myDB”);
databaseIndex = DBDatabaseGetIndex(“myDB”);// get index
DBTableCreate(“myDB”, “myTable”);
tableIndex = DBTableGetIndex(databaseIndex, “myTable”);
DBFieldCreate(“myDB”, “myTable”, “myField”, DB_FIELDTYPE_REAL_GENERAL,
  8, FALSE, FALSE, FALSE); // real number field
fieldIndex = DBFieldGetIndex(databaseIndex, tableIndex, “myField”);

// create records as we need them during the run
recordIndex = 0;// initialize record index
}

on simulate
{
recordIndex++;// increment our record index first (one based)

// append one record to our table
DBRecordsInsert(databaseIndex, tableIndex, 0, 1);

// write one data value to our database
DBDataSetAsNumber(databaseIndex, tableIndex, fieldIndex, recordIndex,
myDataValue);
}
```

☞ As shown in the table in “Data source indexing and organization” on page 94, databases use one-based indexing. Keep this in mind when transferring data between databases and ExtendSim data tables, which are zero-based.

Registered blocks

Block registration is a method for keeping a block informed when there is a change in the linked data source. Unlike user-defined or code-defined links (which link a particular dialog item to a data source), block registration functions link an entire block to a data source.

The block registration functions start with DBBlockRegister or GABlockRegister (see “Linking and notification” on page 337). Depending on the function chosen, the block gets a Link-

Contents message if the content of the data source changes or a LinkStructure message if the structure of the data source (such as the name of a table or the location of a field) changes.

An example of block registration for content changes is the Read block (Value library). When Link Alerts is checked in its Options tab, the block registers itself so that it will be alerted if/when changes are made to its source data.

Registered blocks can be located using the Find Links dialog (Edit > Open Dynamic Linked Blocks) discussed in the User Reference.

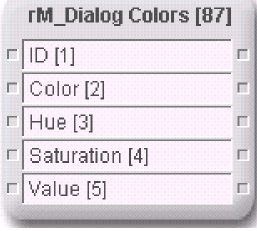
Use registered blocks judiciously. Due to the extra messaging, a registered block can significantly slow the simulation run.

Reserved databases

ExtendSim databases are internal repositories for storing, managing, and controlling model data. A *reserved database* is a specialized type of ExtendSim database that can be hidden from the modeler. Reserved databases provide database capabilities without the modeler having to use, or even be aware of, the reserved database.

Typically a database would be created by a modeler for a specific model. A reserved database, on the other hand, is usually created by a programmer using the database API.

Example



A common use of a reserved database is to support the architecture of a block. For example, when the Resource Manager block (Item library of ExtendSim DE and ExtendSim Pro) is added to a model it creates a reserved database named _rM_Database. There are numerous tables in that database, each focusing on different aspects of the block and how it functions. For instance, the Dialog Colors table, shown here, stores the HSV values of the colors used for text labels in the block's dialog. Other tables track filtering conditions, store resources with their ranking and skill levels, and so forth. When the modeler enters data and makes selections in the Resource Manager block, these are tracked in the reserved database. This process is invisible to the modeler.

The Optimizer block (Value library) is an example where a reserved database is used to provide a feature for the modeler. This is discussed in "Shift-click example" on page 102.

Creating and editing

Reserved databases are created and edited in much the same manner as you would create or edit any ExtendSim database. Some differences are:

- To notify ExtendSim that the database is to be reserved, enter a leading underscore (_) at the beginning of the database's name. For example, the name would look like "_AReservedDB".
- To prevent modelers from accidentally writing to reserved databases, they require special write functions. These are listed on page 327. An error message will be displayed if the special write functions are used for non-reserved databases, and vice versa.
- Since they are intended for developers, ExtendSim doesn't support using blocks (such as Read or Write) to access reserved databases. It also doesn't allow modelers to link dialog items to a reserved database.

- When a block that requires a reserved database is added to a model, the code of the block creates the database in the model. In most cases, unless a block that requires a reserved database is placed in the model, the model will not have any reserved databases.
- If a model has reserved databases, they will not be displayed in the Database List or at the bottom of the Database menu unless you first give the command Develop > Show Reserved Databases. By default, this command is not selected. Furthermore, the command is re initialized to off each time the model is opened.
- Even if a reserved database is not listed in the Database List or at the bottom of the Database menu, it is always accessible through ModL functions.

Changing anything in a reserved database is equivalent to changing the code of a block. It is likely to corrupt any blocks that uses it.

Reading text blocks as commands

Since every piece of text that you add to a model gets its own number, 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. Because each piece of text gets a number, and the text is equivalent to a block's name, you can use BlockName to read text. This can be useful if you want to see how you have changed some text on the model.

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, 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+namLength+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,
      //  GetTypedValue stops the simulation with a message
on Checkdata
{
      // SpeedValue & MassValue are names of dialog parameters
SpeedValue = GetTypedValue("Speed");
MassValue = GetTypedValue("Mass");
}
```

Global function block

```
          // Before 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)
```

IDE

```
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, result goes into global1
          global1 = cos(global0)^2.0+sin(global0)^2.0;
          break;

     case 2:          // The Lemski-Pemski factor
          // global0 has the argument, 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, ExtendSim handles this by allowing you to 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. ExtendSim provides a mechanism to tell the block that its 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, ExtendSim sends a ResumeSim message to that block before the simulation can resume. It also sends a ResumeSimAllBlocks message to all of the blocks in the model. These are optional message handlers because most blocks do not need to recalculate intermediate data, they just use the dialog values directly. But if the block has a ResumeSim or ResumeSimAllBlocks 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 resumes after a change in parameters. It recalculates the coefficient when the modeler 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,calculate coeff
return(coeff);                       // Done, return it
}
```

```
on InitSim       // Initialize 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 ExtendSim, the process of building models typically relies heavily upon modeler 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 modeler participation. However, models can also be built, modified, and controlled indirectly using ExtendSim'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 modelers
- Develop self-modifying models

By using the scripting functions (see "Scripting" on page 300), you can tell ExtendSim which 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 229 or "OLE/COM (Windows only)" on page 232), the scripting functions can be used to build and run entire models based on information from another application.

The scripting functions can also provide a means for developing "wizards" – blocks that help the modeler to perform a task by gathering information, then building or modifying a model based upon that information.

You can also develop blocks that help models achieve desired results. You could 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 Tutorial block (Example Libraries > ModL Tips library > Scripting category) for an example of using the scripting functions.

OLE and ActiveX Automation

ActiveX automation is the process of using the ExtendSim OLE functions, or the scripting environment of another application, to communicate with, exchange data with, or control the other application or ExtendSim.

☞ ExtendSim supports ActiveX automation as either an automation server or as an automation client.

☞ The Examples\How To\Developer Tips\OLE Automation folder contains code examples of ActiveX Automation using VB.net and VBA. Also see discussions in this document on page 120 (C++), page 125 (COM DLL using VB.net), page 125 (VBA), and page 125 (Visual Basic).

Controlling Embedded Objects from ModL script

As of ExtendSim release 10, embedded objects are no longer supported. The functions that supported embedding in prior releases are noted in the Functions chapter of this manual as being "obsolete".

ExtendSim as an Automation Client

To use ExtendSim as an Automation Client, start with a call to the function OLECreateObject. This will return a dispatch Handle to the OLE Automation Server. Once you have the base Dispatch Handle for the server, you can then use the functions defined in "OLE/COM (Windows only)" on page 232 to control the object.

ExtendSim as an Automation Server

ExtendSim also supports a simplified version of Automation as a Server. This is the ability for another application to control the ExtendSim application from outside via OLE.

The automation supported in ExtendSim as an Automation Server is five methods:

- Poke
- Request
- Execute
- BlockMsg
- GetObjectHandle (OBSOLETE AS OF EXTENDSIM 10)

These are used in a fairly simple single object model. Execute, Request, and Poke are the primary means of controlling ExtendSim via ActiveX/OLE Automation. BlockMsg and GetObjectHandle (OBSOLETE AS OF EXTENDSIM 10) are slightly more obscure, but can also be useful. See "C++ examples" on page 120 for an example of how these methods are used.

Object

For use with C++ or other related languages, the Object is the ExtendSim application, referenced by the following GUID:

```
{E167B362-7044-11d2-99DE-00C0230406DF}
```

In Visual Basic, or other environments where ProgIDs are supported, this GUID can also be referenced by the ProgID "Extend.application". ProgIDs are a simplified and easier to remember way of accessing a GUID.

Even though the application name is ExtendSim, for backward compatibility the ProgID is Extend.application. In the future, the ProgID ExtendSim.application will be supported as well, but at the time of this writing the correct ProgID is Extend.application.

Because ModL scripts can be executed with the Execute method, complete control of ExtendSim is available through the following methods:

DispatchID	Method Name	See Page
1	Execute	121
2	Request	122
3	Poke	122
100	BlockMsg	123
101	GetObjectHandle (OBSOLETE AS OF ES10!)	124

Topic and Item

The Poke and Request methods require two strings (Topic and Item) to identify where the data should be poked or where it should be requested from.

Topic is the name of the worksheet or model (or “system” if you don't need to specify a model). If you specify system, the data will be poked to or requested from the top model.

The Item string 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 neither a data table object nor an array.

If the item is not a table, RowEnd and ColEnd can be left off. In this case the string would look as follows:

```
"VariableName:#BlockNumber:0:0"
```

Starting in ExtendSim 7.0.2 there are new forms of the Item string which will allow you to poke directly to, or request directly from, an ExtendSim database table or a global array. These forms are as follows:

```
"GA:#GlobalArrayIndex:RowStart:ColStart:RowEnd:ColEnd"
"DB:#DBIndex:DBTableIndex:RecordStart:FieldStart:RecordEnd:Field-
End"
```

Note that the database case has an extra argument.

C++ examples

The following examples show how to access an IDispatch interface and use the five methods (Poke, Request, Execute, BlockMsg, and GetObjectHandle) in C++. For examples using VB.net and VBA, see the Examples\How To\Developer Tips\OLE Automation folder.

Retrieving the IDispatch interface

The following C++ sample code shows one possible way you could access an IDispatch interface on an ExtendSim application.

☞ The GetActiveObject code attempts to find a running copy of ExtendSim in the ROT (running object table). The CoCreateInstance code creates a new instance of ExtendSim 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 an 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,
                    MAKELANID(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 ExtendSim 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 do any of a number of things.

The C++ code you would use to call the execute method would look something like this (note that this code will cause ExtendSim 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, as specified a in “Topic and Item” on page 120, 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. Topic and Item are specified as in “Topic and Item” on page 120; Value is the string that is going to be poked into the

location specified by Topic and Item. The dispID of poke is 3. Value, Topic, and Item are all strings.

The C++ code you would use to call the Poke method would look something like this (note that this code will set the value of Global0 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"Global0:#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);
```


BlockMsg

BlockMsg sends a message to a specific block in the active ExtendSim model. The DispID of BlockMsg is 100. This message will execute a message handler in the block called OLEAutomation.

The BlockMsg method takes two arguments:

- The first is a block number (integer) and specifies the block that is to receive the message.
- The second is a value (integer, real, or string) and can be used to communicate with the block code.

ExtendSim will set the value of one of three globals to be equal to the value you pass in as the Value argument. Which global will be set is based on which type of variable passed in. The example below uses a string variable. The three globals are OLEGlobal, OLEGlobalInt, and OLEGlobalStr.

As reflected in the code below, the integer values used by the blockMsg method are long integers. However, by default an integer variable declared in Visual Basic is a short integer. So VB programmers should take special care to declare the variables as longs in their VB code.

```
msgVariant      = malloc(sizeof(VARIANTARG) *2);
VariantInit(&msgVariant[0]);
msgVariant[0].vt = VT_I4;
msgVariant[0].lVal = 23; // blockNumber
argStr = SysAllocString((WCHAR *) L"userText ");
VariantInit(&msgVariant[1]);
msgVariant[1].vt = VT_BSTR;// could be a long, or a real as well
msgVariant[1].bstrVal = argStr;

//   Set the DISPPARAMS structure that holds the variant.
dp.rgvarg       = msgVariant;
dp.cArgs        = 2;
dp.rgdispidNamedArgs = NULL;
dp.cNamedArgs   = 0;

//   Call IDispatch::Invoke()
hr = m_pDisp->Invoke( msgID, IID_NULL, LOCALE_SYSTEM_DEFAULT,
                        DISPATCH_METHOD, &dp, NULL, &ei, &uiErr);
SysFreeString(argStr);
```


GetObjectHandle

GetObjectHandle returns the Dispatch Handle value for an embedded object within ExtendSim. This is useful if your outside code is dealing directly with Dispatch Handles, since it will allow you to communicate directly with the embedded object inside the ExtendSim application without going through the ExtendSim interface. The DispID of GetObjectHandle is 101.

Embedded objects are no longer supported as of ExtendSim release 10. The functions that supported embedding in prior releases are noted as being “obsolete”.

```
getVariant      = malloc(sizeof(VARIANTARG)*3);
argStr = SysAllocString((WCHAR *) L"Dialog Item Name");
VariantInit(&getVariant[0]);
getVariant[0].vt = VT_BSTR;// could be a long or a real
getVariant[0].bstrVal = argStr;

VariantInit(&getVariant[1]);
getVariant[1].vt = VT_I4;
getVariant[1].lVal = 23; // blockNumber

argStr2 = SysAllocString((WCHAR *) L"model-1.mox");
VariantInit(&getVariant[2]);
getVariant[2].vt = VT_BSTR;// could be a long or a real
getVariant[2].bstrVal = argStr2;

//   Set the DISPPARAMS structure that holds the variant.
dp.rgvarg       = getVariant;
dp.cArgs        = 3;
dp.rgdispidNamedArgs = NULL;
dp.cNamedArgs   = 0;

//   Call IDispatch::Invoke()
hr = m_pDisp->Invoke( getID, IID_NULL, LOCALE_SYSTEM_DEFAULT,
                        DISPATCH_METHOD, &dp, NULL, &ei, &uiErr);
SysFreeString(argStr);
SysFreeString(argStr2);
```

COM DLL example

The “VB.net COM DLL example” model (Windows only) shows how to interface from ExtendSim with a COM DLL created in VB.net. The COM DLL folder includes the model plus an ExtendSim library with a custom block as well as the source code used to create the COM DLL. ExtendSim is the Client in this automation example; this is the inverse of the Client App example where VB is the Client.

☞ See the folder Documents/ExtendSim\Examples\How To\Developer Tips\OLE Automation\VB\COM DLL

Using VBA for ActiveX/OLE Automation

On page 120 you saw how to control ExtendSim using C++ and the five methods that ExtendSim supports. ExtendSim can also be controlled using Excel and VBA.

The workbook “Excel Client-Server Model Workbook.xls” (Documents\ExtendSim\Examples\How To\Developer Tips\OLE Automation\VBA) contains examples of how to communicate with, send data to, and receive data from an ExtendSim model using Excel VBA and OLE automation. For these examples, Excel is the Client application and ExtendSim is the Server application.

☞ This workbook provides a form-based interface to drive communication between Excel and ExtendSim.

The VBA code in this workbook contains numerous examples that illustrate how to use OLE Automation to remotely interact with an ExtendSim model to perform the following:

- Start and stop the ExtendSim application
- Open and close a model
- Make ExtendSim run asynchronously
- Determine if ExtendSim is running or paused
- Change the end time of a simulation model
- Get a database component, (e.g. table or indexes) from an ExtendSim model's database
- Receive the contents of a database table into a range of worksheet cells
- Poke the contents of a range of worksheet cells into an ExtendSim database table
- Get the path and name of an ExtendSim model
- Get the current time of a simulation run
- Set a dialog parameter value in an ExtendSim model's block

Using Visual Basic for ActiveX/OLE Automation

On page 120 you saw how to control ExtendSim using C++ and the five methods that ExtendSim supports. ExtendSim can also be controlled using Microsoft Visual Basic (VB).

In VB, the syntax for creating or connecting to instances of ExtendSim and for controlling the ExtendSim application is identical to Visual Basic for Applications (VBA) discussed above. The primary difference between VB and VBA is that VB code can be compiled to create executable and DLL files.

The executable file “ExtendSim OLE.exe” (/Documents/ExtendSim/Examples/Developer Tips/OLE Automation/VB/VB Client App) is a VB program that creates an ExtendSim object and uses ExtendSim’s execute, poke and request OLE methods to communicate with and con-

trol the ExtendSim application. This program starts ExtendSim (if necessary), loads a model, sets a dialog parameter, executes a command, and gets a dialog parameter.

The VB source code and all of the Visual Studio files required to build “ExtendSim OLE.exe” are provided in the same folder as the executable file. View the source code to see examples of how to implement ExtendSim OLE methods and how to create and connect to instances of ExtendSim using Visual Basic.

Integrated Development Environment (IDE)

Animation Using ModL

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

This chapter is specific to using ModL functions for performing 2D animation.

2D animation

The functions listed in “2D Animation” on page 250 make it easy to add 2D animation to blocks. This section discusses the general procedures in creating 2D animated blocks, then shows some examples of how you might add 2D animation to the blocks you build.

Overview

☞ Even if Run > Show 2D Animation is not selected, it is still possible to display 2D animation.

The Show 2D Animation command (Run menu) and the Show 2D Animation button in the Model toolbar only control whether animation is shown *during the simulation run*. At all other times, the block will still show animation if the block creator has coded it to do that. For instance, animation is available when the modeler makes any changes in a block's dialog, whether Show 2D Animation is selected or not and regardless of whether the simulation is running. (Of course, when the command Show 2D Animation is selected, animation is available at all times.)

Showing animation outside of a simulation run is a powerful feature because it lets you build blocks which show their initial status or final values without having to turn on 2D animation. For example, if a check box 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 for the Miles block in “Adding 2D animation” on page 53.

Steps

As shown in “Adding 2D animation” on page 53 and described in detail below, the basic steps for adding 2D animation to a block are:

1) Decide how you want the block to animate, including the shape and color that the animation objects should have.
2) Create the 2D animation objects in the Icon tab of the block’s structure window. Do this either by placing an animation object on the Icon tab manually or dynamically through block code.
3) Initialize the animation objects in the CheckData or InitSim handlers.
4) 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 or discrete rate blocks, this code will be in the function 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, 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 change colors or text.

The AnimationPoly function is especially useful. As shown in the Select Value In and Select Value Out blocks (Value library), it allows the animation of an arbitrary shape.

☞ Some animations (such as moving or stretching) will cause simulations 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 (located in the folder ExtendSim/Examples/Continuous/Custom Block Models) for an example of a model that shows animation outside of the icon. Also see the blocks in the Item library for examples of animating from one block to the other, along connection lines.

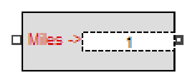
Creating 2D animation objects

All 2D animation is done through the use of one or more animation objects that you put in the Icon tab of the block's structure. The animation object itself is a resizable rectangle with dotted sides, identified by a unique number .

☞ Animation objects are always rectangles in the icon pane. As discussed below, the animation functions 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.

To create an animation object:

- In the Icon toolbar, select the Animation Object button; it is at the bottom or right side of the toolbar, depending on how the toolbar is oriented
- Click in the Icon tab to place the object and resize as desired. Since this is the first animation object, it will have a "1" in it, as shown at right. You will use that object number in all of the animation functions that call this object.



☞ 2D animation objects can also be created "on-the-fly" using the AnimationObjectCreate function.

Each animation object has a *Properties* dialog that can be accessed by right-clicking on the object. This displays information such as the Object ID and allows you to set the exact position and dimensions of the object directly in the structure of the block. If animation objects are layered on top of each other, the *zOrder* allows you to set where in the layer each object will be.

Type: Animation
Name:
X: 45 Width: 38
Y: 27 Height: 13
zOrder: 403

Initializing animation objects

In the block's code, initialize the object in the InitSim message handler. If you always want the animation object visible, even before an animation is run, put the same code in the CreateBlock handler.

The initialization will generally consist of a call to one of the object definition functions (AnimationOval, AnimationRndRectangle, AnimationPoly, and so forth), a call to AnimationEColor (to set the color for the object). It might also include a call to AnimationShow so that the object is visible at the beginning of the simulation (for example, for showing initial text or color). Here is an example:

```
on initSim
{
     AnimationOval(1);              // Set 1 to oval
     //create an EColor value of red with an opaque alpha channel
     Color = EColorFromHSV(0, 255, 255, 255);
     AnimationEColor(1, color);     // Set 1 to the EColor value
     AnimationShow(1);              // Optional
                                    // makes object visible
}
```

☞ If you define the animation object in the code as an oval, and resize it on the icon as a square, 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 animation object is set with numbers for hue, saturation, and brightness (value), often called HSV. Use the Fill Color button in the Shapes toolbar to determine the HSV values of any color.

If you want a block to see whether or not the Run > Show 2D Animation command 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 preceding initialization, you might have:

```
on initSim
{
     AnimationOval(1);           // Set 1 to oval
     //create an EColor value of red with an opaque alpha channel
     Color = EColorFromHSV(0, 255, 255, 255);
     AnimationEColor(1, color); // Set 1 to the EColor value
if (AnimationOn)    // Animation is on
     AnimationShow(1);// Optional: show 1 now
else                        // Animation is not on
     AnimationHide(1, FALSE); // Hide object; it is not
                              //   outside of icon
}
```

Updating the animation object

The Show 2D Animation command only affects 2D animation that is specified in the Simulate message. As mentioned earlier, if Show 2D Animation is not selected, animation is still available except during the Simulate message. During Simulate, ExtendSim blocks check the state of the Run > Show 2D Animation command to determine whether or not to perform animation.

In the Simulate message handler (for continuous blocks) or in the function or message handler appropriate for the data being animated (for discrete event and discrete rate blocks), put the code that checks whether you want to change the animation object (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 hierarchical block's submodel, a block that has code to control the animation on the hierarchical block's icon.

The block in the submodel is used to read the values from other submodel blocks; its code controls 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 its ModL code.

Animating hierarchical blocks is the only time when you would use negative values to reference an animation object. As described in the User Reference, the Animate Value and Animate Item blocks (Animation library) contain code that allows them to animate hierarchical blocks.

The block that controls the animating on a hierarchical block's icon can be deeper than one level in hierarchy. However, searching from the lowest to the highest level, that block will try to animate the first animation object with the correct number that it finds.

☞ If more than one hierarchical block has an animation object with the same number, the lowest one above the controlling block will be the one that is animated.

Showing and hiding a shape

An animation object could be displayed 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 could have a small object near the connector to indicate item arrivals or meeting a condition, such as the Select blocks (Value library). Or you could have a larger object indicating a true value or processing, such as the Activity blocks (Item library).

This example uses 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
     //create an EColor value of red with an opaque alpha channel
     Color = EColorFromHSV(0, 255, 255, 255);
     AnimationEColor(1, color);    // Set 1 to the EColor value
AnimationHide(1, FALSE);           // Hide object; it is not
                                   //   outside of icon
}
on Simulate     // Or appropriate function or message handler
{
...;
if (AnimationOn && ConditionIsTrue) // if both true
     AnimationShow(1);              // Show object now
else
     AnimationHide(1, FALSE);       // Else 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
     //create an EColor value of red with an opaque alpha channel
     Color = EColorFromHSV(0, 255, 255, 255);
     AnimationEColor(1, color);    // Set 1 to the EColor value
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 function 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 change
     {
     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, 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
     //create an EColor value of red with an opaque alpha channel
Color = EColorFromHSV(0, 255, 255, 255);
```

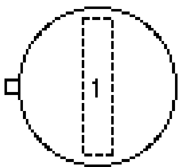
```
AnimationEColor(1, color);    // Set 1 to the EColor value

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 function or message handler
{
AnimationLevel(1, con1in);     // Input connector value (0.0 to 1.0)
                               //   controls the height of the level
}
```

To have the level reflect values other than 0 to 1, scale the input values to correspond to that range. See the Holding Tank block (Value library) for an example of changing a level.

IDE

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, do this to show the relative size of an item (for instance, based on an attribute value) or to indicate a direction of flow.



Object for stretching

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 the image shown here.

```
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
     //create an EColor value of red with an opaque alpha channel
Color = EColorFromHSV(0, 255, 255, 255);
AnimationEColor(1, color);    // Set 1 to the EColor value
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 function or message handler
{
. . .
clipped = Min2(con1in, 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);
}
```

To have the shape stretch outside of the icon, 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.

☞ See also the AnimationPoly function which can be used for animating an arbitrary shape or for dynamically changing the shape of an icon. Examples are the Select Item In and Select Item Out blocks (Item library).

Showing a picture on an icon

You can also have a picture show on an icon in response to some occurrence in the block. For example, the code for a picture might look like:

```
on initSim
{
// Set object 1 to picture, not scaled
AnimationPicture(1, "MyPicture", FALSE);
AnimationHide(1, FALSE);      // Hide object; it is not
                              //   outside of icon
}
on simulate     // Or appropriate function or message handler
{
. . .
if (ConditionIsTrue)
     AnimationShow(1);                  // Show picture
else
     AnimationHide(1, FALSE);           // Hide object; it is not
                                        //   outside of icon
}
```

☞ In order to use a picture, it must be a resource in the System, in ExtendSim, or in a file in the ExtendSim/Extensions folder as discussed in "Picture and movie files" on page 90.

Moving a picture along the connection line between two blocks

A picture traveling along the connection line between two blocks 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, the animation picture can move from one block to the next along the connection line.

☞ In order to use a picture, it must be a resource in the System, in ExtendSim, or in a file in the ExtendSim/Extensions folder as discussed in "Picture and movie files" on page 90.

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 (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 object 1 to picture
myNumber = MyBlockNumber();            // determine "sending" block #
// determine "sending" connector #
outCon = getConNumber(myNumber, "con1Out");
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
{
// animate picture between blocks
AnimationBlockToBlock(1,myNumber,outConn, rBlock, rConn, 1.0);
}
```

Changing a color

To use a change in color instead of motion, include a call to AnimationEColor with a variable for one of the hue, saturation, or brightness (value) arguments:

```
on InitSim
{
hueVal = 0;
AnimationOval(1);                          // Set object 1 to oval
     //create an EColor value of red with an opaque alpha channel
Color = EColorFromHSV(0, 255, 255, 255);
AnimationEColor(1, color);    // Set 1 to the EColor value
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
}
```

IDE

```
on simulate     // Or appropriate function or message handler
{
. . .
hueVal = hueVal+10;                     // Different colors
if (hueVal > 255)
     hueVal = 0;
Color = EColorFromHSV(hueVal, 255, 255, 255);
AnimationEColor(1, color);
. . .
}
```

Changing text

You can also animate by changing text in the animation object created at the beginning of this section.

☞ Using AnimationText() causes a white background around animated text. To not have this, instead use the function AnimationTextTransparent() as is done below.

```
string          objText;

on InitSim
{
AnimationTextTransparent(1, "");     // Set object 1 to blank text
//create an EColor value of black with an opaque alpha channel
Color = EColorFromHSV(0, 0, 0, 255);
AnimationEColor(1, color);    // Set 1 to the EColor value

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 function 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";
AnimationTextTransparent(1, objText);       // Set the text
. . .
}
```

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, 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 located at Documents/ExtendSim/Examples/Continuous/Custom Block Models.

Integrated Development Environment (IDE)

Simulation Architecture

Keep this information in mind as you create and modify ExtendSim blocks

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

This chapter describes how the ExtendSim simulation engine works and how discrete event and discrete rate blocks pass messages and resolve logic issues. This is important information when you are creating or modifying blocks, since you need to know:

- How the block will work with the ExtendSim simulation engine
- How the block will work with the other blocks that ship with ExtendSim

Running a simulation

It is useful to understand the steps that ExtendSim takes when it runs a simulation so you can get a feeling for what parts of the process are relevant to models. The following sections give a step-by-step breakdown of how ExtendSim runs simulations and how it decides what to do next.

☞ Running continuous simulations starts on this page; running discrete event and discrete rate simulations is discussed starting on page 142.

How ExtendSim runs continuous simulations

ExtendSim keeps many system variables handy so that it can compute how to run the model.

Five of the options in the Continuous tab of the Run > Simulation Setup dialog become variables that are used to determine how the model is run:

Option	Variable
End time	EndTime
Start time	StartTime
Runs	NumSims
Time per step (dt)	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, ExtendSim calculates the related variable DeltaTime; if *Time per step* is selected, ExtendSim 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 PreCheckData, CheckData, and StepSize simulation messages.

Pseudocode of the simulation loop for continuous simulations

The following is a *pseudocode* description of ExtendSim's continuous simulation loop. Pseudocode is used by developers to describe the controlling logic of a program in a form that is a cross between English and a programming language.

```
Calculate simulation order of blocks;

for CurrentSim = 0 to NumSims-1
    {
    if (continuing saved paused run) // this was saved while paused
        {
        CurrentTime = SavedCurrentTime;
        CurrentStep = SavedCurrentStep;
        Send ContinueSim message to blocks
        }
    else
        {
        CurrentTime = StartTime;
        CurrentStep = 0;
        Send PreCheckDataMsg to blocks;// Prepare for validation
        Send CheckDataMsg to blocks;// Validate block variables
        if (CheckDataMsg aborts)
            Abort Simulation and select bad block;
        Send StepSize to blocks;//Continuous models can change
                              // stepsize
        Send InitSimMsg to blocks;// Initialize block variables
        Send PostInitSimMsg to blocks;// Check initializations
        }

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

        CurrentTime = CurrentTime+DeltaTime;
        }

    Send FinalCalc and FinalCalc2 to blocks;// Performs final stat
calculations
    Send BlockReport to reporting blocks;// Report results
    ExecuteEndSims:
    Send EndSimMsg to blocks;// Clean up variables, dispose arrays
    }
```

☞ The above comments indicate one purpose of the specific message handler, but each of them can be used for a variety of purposes, as shown in the table on page 141.

The next several sections discuss the meaning of this pseudocode.

Simulation order

Simulation order defines the sequence in which messages are sent from the ExtendSim application to the blocks in a model. The simulation order section of the pseudocode calculates the simulation order, either flow or left-to-right, as discussed in the User Reference.

Initialization

The pseudocode:

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

instructs ExtendSim to execute the contents of the “for” loop NumSims times. You set Num-Sims in the *Runs* entry box in the Continuous tab of 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, it will halt the current simulation, increment currentSim by 1, and run any remaining simulations.

When messages are sent

Different messages are sent, depending on whether the run is paused or is a new run:

- If the simulation was previously paused and saved before the run was completed, ExtendSim will send a ContinueSim message to all of the blocks in the model. Since the model was already initialized and only needs to set up its data structures to continue the previous run, the ContinueSim message is sent in lieu of the PreCheckData, CheckData, StepSize, InitSim, and PostInitSim messages that would follow if it were a new run, described below.
- If the run is a new run, the first thing done inside this loop is to send the PreCheckData and CheckData system message to all the blocks in the simulation. These message handlers in the various blocks should check the values of dialog items to see if they are valid. If any of them abort, ExtendSim recognizes that fact and the simulation aborts with the block selected. Connection status of the block can be tested during CheckData: Input connectors that are connected will show a non-zero value, and unconnected input connectors will show a zero value. This is useful in determining if a block’s inputs are actually connected in the model.

The StepSize message is then 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 Filter blocks (Electronics library) use the StepSize message in this way because they need a calculated value of DeltaTime to get accurate results.

Finally, 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. After the InitSim message has been sent to all blocks, the PostInitSim message is sent and gives the developer a chance to initialize any variables that needed all of the other blocks to be initialized first.

Table of message handler purposes

The following table lists the major purposes of the messages sent during the initialization loop for Value library blocks. Notice that some purposes only apply if the Value library block is used in a discrete event, rather than a continuous, model.

☞ While the table lists the major purposes of the message handlers, a purpose can sometimes be accomplished using a different message handler than the one indicated.

Message Handler	Purpose in Value Library Blocks
CheckData	Check validity of parameters Assign position in future events calendar (discrete event models) Check to see which connectors are connected Determine parameter links with database Check for duplicate seeds Determine if dialog variables are cloned
StepSize	Throw and Catch Value blocks create data structures
InitSim	Determine if model is discrete event or continuous Turn off simulate messages in discrete event models Schedule events in discrete event models Initialize variables Calculate local block seed value
PostInitSim	Initialize connector values Schedule events in discrete event models

☞ For the Data Import Export and Command blocks, initialization message handlers can be selected using options in the blocks' dialogs.

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 Number of steps option in the Continuous tab of the Simulation Setup dialog, NumSteps becomes 2, the simulation step loop sends Simulate messages to the blocks for CurrentStep equal to 0, and than 1. (If you want to run a simulation for only a single step, set Number of steps 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. 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, 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 the ExtendSim global variables to help establish some safeguards against these conflicts. For instance, this is how the Executive block (Item library) is able to control the progression of time in discrete event models.

Final messages

Upon completion of the simulation, ExtendSim sends out the FinalCalc and FinalCalc2 system messages to all blocks. These message handlers are used to perform final calculations for all time-dependent statistics.

The Report Manager block (Value library) then sends the BlockReport system message to any blocks that have been selected for a report. This message handler is responsible for writing all report data to the appropriate ExtendSim database tables.

Finally, ExtendSim 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 stops 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 ExtendSim runs discrete event or discrete rate simulations

ExtendSim keeps many system variables handy so that it can compute how to run the model. Three of the options in the Setup tab of the Run > Simulation Setup dialog become variables that are used to determine how a discrete event or discrete rate model is run:

Option	Variable
End time	EndTime
Start time	StartTime
Runs	NumSims

For discrete event and discrete rate models, the DeltaTime and NumSteps variables (used in continuous simulations and described on page 138) are ignored, because the Executive block (Item library) calculates CurrentTime based on the time of the next event.

Pseudocode of the simulation loop in discrete event/rate simulations

The following is a *pseudocode* description of ExtendSim's discrete event/discrete rate simulation loop:

```
for CurrentSim = 0 to NumSims-1
    {
    if (continuing saved paused run) // this was saved while paused
        {
        CurrentTime = SavedCurrentTime;
        CurrentStep = SavedCurrentStep;
        Send ContinueSim message to blocks
        }
    else
        {
        CurrentTime = StartTime;
        CurrentStep = 0;
        Send PreCheckDataMsg to blocks;// Prepare for validation
        Send CheckDataMsg to blocks;// Validate block variables
        if (CheckDataMsg aborts)
            Abort Simulation and select bad block;
```

```
    Send StepSize to blocks;//Continuous models can change
                          // stepsize
    Send InitSimMsg to blocks;// Initialize block variables
    Send PostInitSimMsg to blocks;// Check initializations
    }

// simulation Phase is controlled by the Executive block
// Do until simulation end reached
Do While (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 and then FinalCalc2 to all blocks;
Send BlockReport to all reporting blocks;
ExecuteEndSims:
Send EndSimMsg to all blocks;
}
```

☞ The above comments indicate one purpose of the specific message handler, but each of them can be used for a variety of purposes, as shown in the table on page 144.

The next several sections discuss the meaning of this pseudocode.

Initialization

The pseudocode:

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

instructs ExtendSim to execute the contents of the “for” loop NumSims times. You set NumSims in the Runs field of either the Setup or Continuous tabs of 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 currentSim by 1, and run any remaining simulations.

When messages are sent

Different messages are sent depending on whether it is a new run or the run is paused. In addition, the sequence of sending messages is block dependent. To see the order of messages being sent, look at the block’s code.

☞ PostInitSim is the only message handler where messages can be passed through block connectors.

Table of message handler purposes

The following table lists the major purposes of the messages sent during the initialization loop for Item and Rate library blocks, as well as for the Executive block in discrete event and in discrete rate models.

☞ While the following tables list the major purposes of the message handlers, a purpose can sometimes be accomplished using a different message handler than the one indicated.

Message Handler	Purpose in Item Library Blocks
PreCheckData	Initialize Resource Pools
CheckData	Check validity of parameters Assign position in future events calendar Check to see which connectors are connected Determine parameter links with database Check for duplicate seeds Determine if dialog variables are cloned Check the version of the Executive Get Executive block number Determine if costing is used in the model Initialize DB Equation variables Check Animation Register blocks with ExtendSim database
StepSize	Initialize attribute data structures Initialize Animation Initialize Shifts
InitSim	Initialize variables Initialize attribute animation conversion table Schedule initial events Set local block seed value
PostInitSim	Initialize connector values Check if a downstream block controls the flow of items (blocker) Schedule initial events

Message Handler	Purpose in Rate Library Blocks
CheckData	Update the indexes of the global arrays used for the Rate library system Update information about the flow and value connections If the block starts a section, create the section in the appropriate global array Determine if the type of Shift is On or Off Assign position in future events calendar Check parameters in the block
StepSize	Initialize animation Update the unit conversion factors (flow unit and time unit) Initialize Shifts Initialize attribute data structures in Interchange block
InitSim	Initialize variables (static variables, dialog items) Initialize next event Initialize value connectors Initialize attribute animation conversion table in Interchange block

Message Handler	Purpose in Executive Block
PreCheckData	Dispose Throw Flow and Catch Flow block information (discrete rate models)
CheckData	Initialize system global variables Post Executive block number Send CheckData message through event message connector Check for duplicate random number seeds Create the global arrays used for the Rate system (discrete rate models) Initialize the global variables (discrete rate models) Rebuild and update Throw Flow and Throw Catch info (discrete rate models)
StepSize	Initialize attribute data structures From the beginning, propagate through each section (discrete rate models) Create the Unit Groups in the model (discrete rate models) Create the Lookup List for flow units and time units (discrete rate models) Create and update global arrays linked to the sections (discrete rate models)
InitSim	Initialize variables Pass event arrays to other blocks
PostInitSim	Launch and perform calculation of all effective rates (discrete rate models)

Simulation phase

The Item and Rate libraries rely on the Executive block rather than ExtendSim to control CurrentTime and NumSteps. As opposed to continuous simulations, which rely on time being incremented uniformly between each step, discrete event and discrete rate simulations are event based. In discrete event and discrete rate simulations, NumSteps is modified constantly and the Executive block monitors and manipulates CurrentTime based on the events the blocks have posted in the Executive's event queue.

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 146 for a detailed discussion).

☞ The code of the Executive can be seen by opening the block's structure.

Final messages

Upon completion of the simulation, ExtendSim sends out the FinalCalc and FinalCalc2 system messages to all blocks. These message handlers are used to perform final calculations for all time-dependent statistics.

The Report Manager block (Value library) then sends the BlockReport system message to any blocks that have been selected for a report. This message handler is responsible for writing all report data to the appropriate ExtendSim database tables.

Finally, ExtendSim 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 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 alter the blocks that come in the Item library.

☞ Always put copies of blocks from the Item library into your own library and alter the copies rather than the originals.

The Make Your Own category of blocks in the Example Libraries > ModL Tips library have sample code and comments that explain how the blocks work. You can use these blocks as templates for your own discrete event blocks.

The following sections discuss 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 event-based, add the Executive block (Item library) to the model worksheet. This block does two things:

1) It maintains a data structure of information about the items in the model
2) It takes control of the time clock from the ExtendSim 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 uses three dynamic arrays to store future events in the model.

- The TimeArray contains the event times
- TimeBlocks contains the block numbers of the blocks that post events
- TimeEventMsgType stores the constant for the message handler that is called when the event time for a block is reached

The Executive block passes the arrays to the system globals with the statements:

```
SysGlobal0 = passArray(TimeArray);
SysGlobal7 = passArray(TimeBlocks);
SysGlobal13 = passArray(TimeEventMsgType);
```

☞ 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.

Scheduling events

Each block that needs to post an event at a specific time in the future needs to add itself to the Executive block's TimeArray, TimeBlocks, and TimeEventMsgType arrays. This is done by reserving a position in the arrays and then setting that position in TimeArray to the next event time (this needs to be done each time a new event time is posted) and TimeBlocks to the block number of the event scheduling block.

The position in the arrays is reserved in the CheckData message handler with the following code:

```
on checkdata
{
// SysGlobalInt0 is current number of event posting blocks in model
// Reserves a position in TimeArray & TimeBlocks arrays
myIndex = SysGlobalint0;
SysGlobalint0 += 1;
}
```

SysGlobalInt0 has been initialized to 0. By incrementing SysGlobalInt0, each block will get a unique value for myIndex.

In the InitSim message handler, the two arrays are passed in from the Executive, the block number is assigned to this block's position in TimeBlocks, and the initial event time is assigned to TimeArray:

```
on initsim
{
// get the pointer to the TimeArray and TimeEventMsgType arrays
     if(getPassedArray(SysGlobal0, timeArray) > 0)
          {
          // blocks in de models do not get Simulate messages
          GetSimulateMsgs(False);
          // set the first event time to the start of the simulation
          timeArray[MyIndex] = StartTime;
          // get the pointer to the TimeBlocks array
          getPassedArray(SysGlobal7,TimeBlocks);
          // put this block's # in reserved position in TimeBlocks
          TimeBlocks[myindex] = myBlockNumber();
          //Get the pointer to the TimeEventMsgType array
          getPassedArray(SysGlobal13,TimeEventMsgType);
          //reserved position in TimeEventMsgType
          TimeEventMsgType[myIndex] = BlockReceive1Msg;
          }
     else
          GetSimulateMsgs(True);
}
```

If this block is used in a continuous model, the GetPassedArray call will return 0 and the on Simulate message handler will be called at every simulation step.

When the Executive sends this block a message (at the time specified in TimeArray), the block will receive the message in TimeEventMsgType. If you do not set a value in TimeEventMsgType, the block will receive a BlockReceive0 message.

Put the event code in this message handler and reschedule the block for the next event time:

```
On BlockReceive1
{
// process event
ShowTime = CurrentTime;
// schedule event in the future
TimeArray[MyIndex] = CurrentTime + EventTime;
}
```

The Event block (ModL Tips library) illustrates the event scheduling procedure.

Residence, passing, and decision blocks

There are three types of item-handling discrete event blocks: *residence blocks*, *passing blocks*, and *decision blocks*:

1) 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 and Activity blocks.
2) Passing blocks pass items through without holding them. These blocks implement modeling operations that are not time based and usually do not post future events. Examples include setting an attribute or getting information from an item (Set or Information blocks). Passing blocks 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.
3) Decision blocks control the flow of items in the model; they do not post future events. Examples include limiting the number of items (Gate block) or selecting an output (Select Item Out blocks). Note that some decision blocks can behave as passing or residence blocks, depending on which options are selected for the particular block.

Blocks that post future events

Each 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 checkData message handler, called “myIndex”. Each 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, TimeArray (the event list) is searched to find the next event time. A third dynamic array, NextTimes, 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 message is sent to each block in the NextTimes array, one block at a time.

After receiving its event message, each block processes its own unique code based on that event type. For example, a Convey Item block will attempt to pull in an additional item when the event for an open input occurs, or the Activity block will attempt to push an item out when that item's processing duration has completed. Some blocks will conditionally post an event based on the options selected. An example of this is the Queue, which will post an event if reneging is selected. Blocks do not have to process items for an event to occur – the Clear Statistics block (Value library) generates an event at the clear time when used in a discrete event model.

Residence blocks that do not post future events:
Some of the residence blocks that do not post future events, such as the Queue Matching and Resource Item blocks, also attempt to move items at event times.

If a residence block needs to return from a message but also needs 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 "zero time event" (discussed below) ensures that the residence block will receive an additional message before the next time step so that additional items can be processed.

Zero time events

Sometimes a block needs to post two events at the same simulation time yet have those events be sequential. In addition, the other blocks in the model need to be given a chance to complete the posting of their own events after the second event.

An example of this is an Activity block that has just released its item after the specified time delay. The Activity will need to do two things at the same event time: send the item out to the next block (if that path is not blocked) and pull another item in (if an item is available). To do this, it sends the item to the next block and then posts a "current" event to the Executive. Before the simulation clock advances, the Activity will receive a BlockReceive0 message and will then try to pull another item in.

To post a zero time event, the block assigns SysGlobalInt8 to its block number and sends a BlockReceive3 message to the Executive. The Executive maintains a list of all the blocks that have sent a BlockReceive3 message and before it advances the simulation clock sends a BlockReceive0 message to each of these blocks in turn. In the example below, the "rescheduled" flag is used to prevent the block from posting more than one zero time event at a time:

```
if ( ! rescheduled)
     {
     // remember block is scheduled as a zero time event
     rescheduled = TRUE;
     // set the block number for this block
     SysGlobalInt8 = MyBlockNumber();
     // send message to Executive for a zero time event
     SendMsgToBlock(Exec,BLOCKRECEIVE3MSG);
     }
```

Item data structures and indexes

The Executive block stores information about each item in a discrete event model in a set of dynamic arrays that it passes into the reserved global variables SysGlobal3, SysGlobal4, SysGlobal6, SysGlobal9, and SysGlobal12 (see "Global variables" on page 191), as well as a set of two global arrays (see "Global arrays" on page 342). 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 a connector value of 0 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 receives index number 5, it accesses 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] and itemArrayC[][10]) and three arrays of integer information (itemArrayI[][5], itemArrayI2[][5], and itemArray3D[][10]). These arrays are described below:

Real array

The real array itemArrayR[][3], passed through SysGlobal3, contains the following information:

Slot #	Description
0	Quantity: The number of items that the current item represents. This is used, for example, by the Set block (Item Library). Item quantities are used to copy items in queues and resource blocks. They are also used by certain input connectors (such as on the Activity block) to convey additional information to a block.
1	Priority: Used by the Set, Get, and Queue blocks. Note that in ExtendSim the lower the number (including negatives), the higher the priority.
2	Reserved for future use.

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 that 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 block.
6	Unused
7	Original Cost: Used in calculating cost when unbatching items using the Unbatch block (Item library).
8	Unused
9	Unused

Integer arrays

There are two integer arrays: itemArrayI and itemArrayI2.

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 (Item library) 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 Item library.
3	Unused except where needed to provide backwards compatibility with Extend 5 or earlier.
4	Block number where item is.

ItemArrayI2[][5] is passed through SysGlobal18 and has 5 columns:

Slot #	Description
0	Resource Order ID. If an item has requested at least one advanced resource requirement, the integer held in slot 0 represents the record associated with the last requested requirement in the "Resource Orders" database table. Resource Order ID is used with Advanced Resource Management (ARM).
1	Unique item ID used by the Report Manager block
2	Report Manager block's log record index
3	RBD. Event item's record index into "RBD event registry"
4	RBD. Event item's record index into "RBD event occurrence log"

Item attribute global arrays

There are three global arrays that are responsible for item attributes:

- The first (_AttributeList) stores attribute names
- The second (_AttribType) stores the attribute type (value, string, or db address)
- The third (_AttribValues) stores attribute values

The attribute names are stored in a String15 type global array called "_AttributeList". This array is created whenever a block that 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 modeler 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.

The "_AttribType" global array is used to store the attribute's type at the time the user defines a new attribute. There are 3 types of attributes: value, string, and db address.

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 row index) in alphabetical order. Each block then calls the Attrib_GetColumnIndex procedure which searches the AttribList global array for the attribute that is referenced by the calling block. This index is used when referencing the attribute value during the simulation.

In addition, if either of the two costing attributes ("_cost" or "_rate", as discussed in the User Reference) 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 third 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 and 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: the following is pseudocode for setting an attribute:

```
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
```

IDE

Flow attribute global arrays.

The blocks in the Rate library use flow attributes which are implemented using the same logic as discussed above for item attributes. The global arrays responsible for flow attributes are:

- The first (_FlowAttList) stores attribute names
- The second (_FlowAttType) stores the attribute type (value or string)
- The third (_FlowAttValues) stores attribute values

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.

For single connectors, messages are sent using the functions *SendMsgToInputs(connectorname)* and *SendMsgToOutputs(connectorname*). For variable connectors, messages are sent using *SendMsgToInputs(connectorname, whichConnector)* and *ConArraySendMsgToOutputs(connectorname, whichConnector)*. 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.

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:

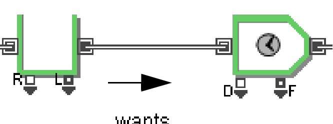
Value	Sending (SysGlobalInt3)	Returning (SysGlobalInt0)
0 (rejects)	(Not sent)	Block rejects item
1 (wants)	Does block want item?	(Not returned)
2 (taken)	Item has been taken	(Not returned)
3 (needs)	Item needs to be taken	Block needs item
4 (query)	What is the index of the next item?	Item index for 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 following figures show a Queue block connected to an Activity block with a single item capacity.

Push mechanism

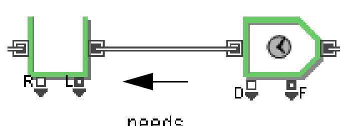
When an item is pushed through the model, the upstream blocks (in this case, a Queue) try to push their items out into any downstream blocks.

For example, assume that an item has just arrived at the Queue and the Queue is attempting to pass that item along to the Activity. The first action is for the Queue to send a *wants* message through its item output connector to the Activity. This is accomplished 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.



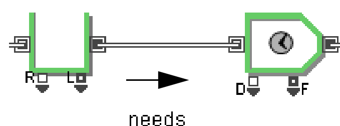
Wants message sent to Activity

The Activity receives the message in its “on itemIn” message handler. If the Activity in the above example is not currently processing an item and thus is 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

If the Activity is currently busy processing another item, instead of a needs value it will return a *rejects* value . This is

accomplished 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.

taken

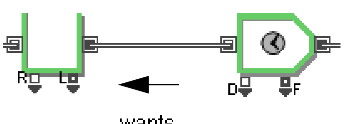
Taken message sent back to Queue

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.

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 *SendMsgToOutputs* function.



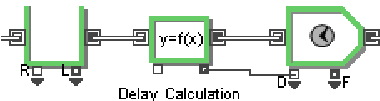
Wants message sent from Activity

If an item is available in the Queue, its 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 an Equation(I) block reading multiple attribute and other property values to calculate the delay needed for the Activity. The Equation(I) 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.



Passing block between Queue and Activity

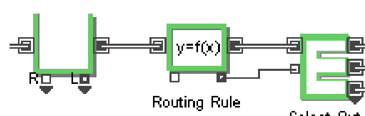
Blocked and query messages

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.

If *Predict the path of the item before it enters this block* is selected in its dialog, the Select Item Out block will use a special sequence of messages to determine which direction the item will go before the item leaves any upstream residence blocks.

Traditional discrete event architectures would require dummy resources to overcome these problems. The ExtendSim 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 at right, the Equation(I) block reads multiple attribute and other property values to calculate the one of three paths for that item to follow (Select Item Out) based on the value of the properties for that item. Without the ability to send a Query message, the Equation(I) block would read the value of the properties on the item, but only after the item has already entered the block. In this case, this could cause a problem as the Select Item Out block may be blocked down the path that the item will need to travel. If the Select Item Out is blocked, and the item has to move into the Equation(I) block to present its property values, then the item will be stuck in the Equation(I) block. And since property-manipulating blocks are only meant to pass items, not hold them, the Queue will understate the number of items available.

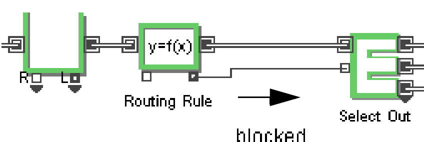


Select Item Out controlled by Equation(I)

The solution to this problem is to use the Equation(I) block which 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

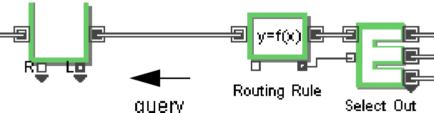
The first part of this process involves sending a *blocked* message downstream from the Equation(I) to see if there are any blocks that could cause this situation. This is accomplished by setting SysGlobalInt3 to 6 (blocked) and calling the *SendMsgToInputs* function. The Equation(I) does this the first time it gets an incoming message (i.e. the first time the Select Item Out block requests a calculated value from the value out connector.) The Equation(I) sends a blocked message from its item output connector in its on AttribOut message handler. (The on AttribOut message handler is called when the value out connector on the Equation(I) 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 Equation(I) block sets a flag that records that it is blocked.



Blocked message sent to Select Item Out

Query messages

If it has been determined that blocking can occur, each time there is a request for a calculated value, the Equation(I) 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

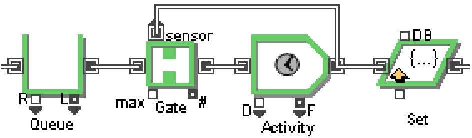
The block responding to the *query* message will check to see what the item index of the next item to be released will be and will return that value to the querying block by setting the global SysGlobalInt0 to the item's index value. The Equation(I) block will then access the property values for that item. From this point on, each time that a property value is requested from the Equation(I) block, it will send out the *query* message and check the property values of the next item. It will not need to re-send the *blocked* message again.

ExtendSim will notify modelers 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 this 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

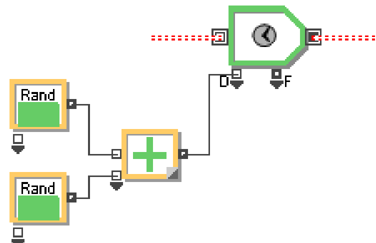
As an item travels from the Activity to the Set block, a *taken* message will be sent to the Activity. In response to this message, the Activity will send out the *notify* message. The normal item input connectors in the blocks will ignore the *notify* message, but sensor connectors will respond to it 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 that perform numerical calculations 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 that follows, two Random Number blocks are added together to specify the delay for an activity. Whenever an item arrives to the activity, the Random Number and Math blocks will need to be recalculated. Whenever a discrete event block detects a condition where an update to the input value of a connector is needed (in this case, an item arriving to the activity), it sends a message out its input value connector (in this case, value input connector “D”) using the *ConArray-SendMsgToOutputs*() function. This message propagates backwards via the Math block and causes the Random Number blocks to recalculate their output values so that the Math block can add them and update its output connector value for the Activity block to use.



The sum of two random variables specifying an activity’s delay

Message emulation

A default feature (“message emulation”) for continuous blocks in a discrete event or discrete rate model causes the messages to be propagated throughout all of the continuous blocks used in the calculation. Message emulation is used whenever the 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.

How message emulation works:

- If a message is received on an output connector:
    - Echo to ALL input connectors to get their latest values.
    - Send a SIMULATE message to the block to recalculate values to its output connectors.
- If a message is received on an input connector:
    - Echo to all other inputs THAT HAVE NEVER RECEIVED A MESSAGE to get their latest values.
    - Send a SIMULATE message to the block to recalculate values to its output connectors.
    - Echo message out the outputs.
- If Additional messages are received on any connector while processing current message:
    - DON'T echo these additional messages.
    - Send a SIMULATE message to the block to recalculate its latest values.

This message emulation capability improves performance and reduces redundancy.

The example code below illustrates an On Simulate message handler adding one to its input by assigning the output connector (Con1Out) to the input connector (Con1In) plus one. There are two cases:

- The block received a message from its input connector to recalculate, so it calls its Simulate message handler and then propagates that input connector message by then sending a message via its output connector to any connected block's inputs so they can act on that message.
- The block received a message on its output connector from a downstream block that needs a new value. It first propagates the message backwards through its inputs so upstream blocks can recalculate, then calls its Simulate message handler to recalculate its outputs for the downstream block to use.

Whenever either connector receives a message, the Simulate message handler will be executed and a message will be sent out the other connector through message emulation. This will propagate messages where appropriate.

```
On Simulate
{
Con1out = Con1In + 1; // calculate the output value
}
```


Explicit connector messages

Overriding message emulation gives you, as a block developer, 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 of the output connector whenever a message is received on the input connector, and a message must be sent out of the input connector whenever a message is received on the output connector.

The code below uses message handlers to make a block behave similarly 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
{
}
```

Functions in discrete event blocks

The following are some, not all, of 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).
SendMsg	Sends the messages out through the connectors. It sends out messages based on the values of its arguments.

Globals in discrete event blocks

Several reserved global variables (sysGlobals) are used in discrete event blocks. This table lists the reserved 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 blocks in the Make Your Own category of the Example Libraries > ModL Tips library or the Executive block and other blocks in the Item library.

In addition to the global variables reserved by ITI, there are general use global variables. To see the complete list of global variables, go to “Global variables” on page 191

☞ Global variables are integers if they contain the “Int” designation (SysGlobalIntX) and are strings if they contain the “Str” designation (SysGlobalStrX). Otherwise, they are reals.

Use of system globals during Simulate message

Global	Used In	Definition during Simulate message
SysGlobal0	DE blocks	Used to access the TimeArray of posted events.
SysGlobal1	Blocks with reports	Report file number.
SysGlobal2	Blocks with traces	Trace file number.
SysGlobal3	DE blocks	Used to access the real item array of discrete event item data.
SysGlobal4	DE blocks	Used to access the integer item array of discrete event item data.
SysGlobal5	Do Not Use	DO NOT USE. (Was used in versions prior to 7.0.3 to pass the TimeEventMsgType array between the Executive and event scheduling blocks. Use SysGlobal13 instead.)

Global	Used In	Definition during Simulate message
SysGlobal6	DE blocks	Used to access the string item array of discrete event timer data.
SysGlobal7	DE blocks	Used to access the TimeBlocks array of event posting blocks.
SysGlobal8	Resource pools	Used in communication between the Resource Pool, Queue (in Resource Pool mode), and Resource Pool Release blocks.
SysGlobal9	Costing blocks	Used to access the cost item array for discrete event item data.
SysGlobal10	Global arrays	Communicates the value, row, and column to other global array blocks when a value in a global array has changed.
SysGlobal11	Throw and Catch blocks (Value library)	Passes a value between blocks.
SysGlobal12	DE blocks	Used to access the integer item array of discrete event itemArrayI2 data.
SysGlobal13	DE blocks	Passes the time event message type array between Executive and event scheduling blocks.
SysGlobal14		Not used
SysGlobal15		Not used
SysGlobal16	Blocks with attributes	Passes array containing names of attribute arrays.
SysGlobal17	Catch Item	Passes Throw block nums array.
SysGlobal18		Not used
SysGlobal19	Proof Animation blocks	Passes ProofString as an array.
SysGlobal20	Executive	Used in BlockReceive5 as a DB Address argument for the calling block and used by the Executive as the return value.
SysGlobal21		Passes Resource Orders from Batch block to Resource Manager
SysGlobal22	Item Log Mngr	Used to pass arrays from Item Log Manager to satellite blocks (e.g., the History block).
SysGlobal29		Not used
SysGlobalStr0	Blocking blocks	Name of attribute being checked upstream.
SysGlobalStr1	Any block	Used as an argument for the block table info (BlockReceive4) message handler.
SysGlobalStr2	Blocks with attributes	Name of new string attribute passed to Executive.

Global	Used In	Definition during Simulate message
SysGlobalStr4-9		Not used
SysGlobalInt0	DE blocks	Return code from the messages that discrete event blocks send to each other.
SysGlobalInt1	DE blocks that create items	Index value for the first free row in the item arrays maintained by the Executive block.
SysGlobalInt2	DE blocks	Total number of rows of data that have been allocated to the item arrays. This will always be rounded up to the next allocation level.
SysGlobalInt3	DE blocks	Argument to the messages that discrete event blocks send to each other.
SysGlobalInt4	Blocking blocks and priority	Tells whether the priority is being checked.
SysGlobalInt5	Used for v6 compatibility	Global batch count.
SysGlobalInt6	Blocking blocks	Specifies whether or not there is a blocked block (see the Get block).
SysGlobalInt7	DE blocks	ID of the item currently being disposed.
SysGlobalInt8	DE blocks	Used to pass the block number to the Executive when the block is rescheduling itself in the CurrentEvents array.
SysGlobalInt9	Random Number, Holding Tank (Value library)	Flag when the graph is sending a message. (See the code of the Line Chart, Scatter Chart, and Random Number blocks for more information.)
SysGlobalInt10	Resource pools	Used in communication between the Resource Pool, Queue (in Resource Pool mode), and Resource Pool Release blocks.
SysGlobalInt11	Select Item Out	Used to control whether or not a new random value is generated in a connected Random Number block.
SysGlobalInt12	Throw and Catch (Item library)	Passes item index in Throw Item and Catch Item blocks.
SysGlobalInt13	Random Number, Equation, and Select blocks	Set to TRUE if a new random number should be generated for a select block.
SysGlobalInt14	Throw and Catch (Item library)	Used by Throw Item and Catch Item blocks to determine which Throw block is sending the message.

Global	Used In	Definition during Simulate message
SysGlobalInt15	Resource pools	Used in communication between the Resource Pool, Queue (in Resource Pool mode), and Resource Pool Release blocks.
SysGlobalInt16	Blocks with costing	Set to TRUE during CheckData message if discrete event model is calculating item costs.
SysGlobalInt17	Blocks with attributes	Holds the number of attributes in a discrete event model.
SysGlobalInt18	Blocks with tables	Argument to BlockTableInfo message handler that chooses the type of information to return.
SysGlobalInt19	DE blocks	Used as an argument for blockreceive4 (on queueFunction message handler).
SysGlobalInt20	Queues, Resource pools	Used in direct communication with queues and resource pools.
SysGlobalInt21	Queues, Resource pools	Used in direct communication with queues and resource pools.
SysGlobalInt22	Set, Get, Executive	Executive and attribute blocks.
SysGlobalInt23	DE blocks	Block number of the Executive.
SysGlobalInt24	Create, Set, Get, Execu-tive	Attribute info command number.
SysGlobalInt25	Set, Create, Executive	Number of attribute name arrays in a block.
SysGlobalInt26	Any block	Block number of item tracing block.
SysGlobalInt27	DE blocks	Item index sent to tracing block.
SysGlobalInt28	Resource Pool, Queues	List number when resource pool tries to send item. Used in BlockReceive1 message.
SysGlobalInt29	Resource Pool, Queues	Set in Queue in PreCheckData to tell Pools the Historical Log has been turned on.
SysGlobalInt30	Resource Mngr	Resource Requirement record.
SysGlobalInt31	Resource Mngr	Item Index.
SysGlobalInt32	Resource Mngr	Pointer to Resource method.
SysGlobalInt33	Item library	Block number for Resource Manager.

Global	Used In	Definition during Simulate message
SysGlobalInt36	Resource Mngr	Number of selected resources.
SysGlobalInt37	Unique ItemID	Keeps a count of how many items have been created. Used to assign a unique ID to items in ItemArrayI2.
SysGlobalInt38	Item blocks	Msg type. Tells the remote satellite block what to do in Block-Receive6 for item logging.
SysGlobalInt39	Item blocks	The table index of the Item Log block's central log table.
SysGlobalInt40		Global setting for whether or not to track string attribute values.
SysGlobalInt42	Mean & Variance block	Track the number of blocks that have relative error turned on. This makes sure that the simulation runs continue until the all of the relative error conditions have been met.
SysGlobalInt43	Item & Rate	True when the addition of a block is being scripted. Prevents the automatic redrawing of connection lines.

Use of Global variables during CheckData or InitSim messages

Some of the variables in the above table are also used in the CheckData and InitSim message handlers and have a different meaning than when used in the Simulate message handler.

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.
SysGlobalStr1	Used by equation to send a value to Proof Animation.

Creating blocks for discrete event models

A Make Your Own block (Example Libraries > ModL Tips library > Make Your Own category) 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.
- To control how the block receives and sends messages, add at least one message handler (which 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
{
...
}
```

☞ Using options in the Run > Debugging command, you can cause models to display block messages as they run. This gives an idea of how messaging works in discrete event models.

How discrete rate blocks and models work

The blocks in the Rate library are for creating discrete rate models. LP technology, which has global oversight over discrete rate models, as well as messaging in discrete rate models, is discussed in the User Reference.

Globals in discrete rate blocks

Several reserved global variables (SysFlowGlobals and others) are used in the Rate library blocks. The table below lists those global variables with a brief description of their use in Rate library blocks during the Simulate message (most have undefined values during the CheckData message).

Global	Definition during Simulate message
SysGlobalInt41	Rate library
SysFlowGlobal0	To store the maximum rate defined in the Executive
SysFlowGlobal1	To store the rate precision to be considerate as 0
SysFlowGlobalStr0	Used for the propagation of the name of the flow unit in a section
SysFlowGlobalInt0	Used to propagate the section# (= corresponding to the row# in _FlowSection global array)
SysFlowGlobalInt1	Used to propagate the row# in the _FlowAttValues global array for each outflow connector that can change attributes
SysFlowGlobalInt2	Used to stop the propagation when the message is received twice in the same block => count the LP calculations made in the Executive
SysFlowGlobalInt3	{1/2/3} Option defined in the Executive block to show or not the bias order above the Merge and Distribute icons using a mode with bias order
SysFlowGlobalInt4	Used to get the type of propagation which is made in FlowBlockReceive0 and FlowBlockReceive3 messages
SysFlowGlobalInt5	Count the number of constraints in an LP calculation
SysFlowGlobalInt6	Row # in siListTCConnection_GA global array. Used to update when change occurs in Throw Catch connections.
SysFlowGlobalInt7	{0/1} True if the Rate blocks have to update the state starved or blocked after each new calculation of the rates
SysFlowGlobalInt8	Used to inform the block on the number of the section which is concerned by the message handler (FlowBlockReceive0_1_3_4_6_7

Global	Definition during Simulate message
SysFlowGlobalInt9	Used to inform which is the sense of the propagation when a message is sent from a block
SysFlowGlobalInt10	Unused
SysFlowGlobalInt11	Used to inform from which connector number the message has been sent (in FlowBlockReceive0 msg)
SysFlowGlobalInt12	Value of the popup menu in the Executive
SysFlowGlobalInt13	Used to inform on the option taken in the Executive for the merge percent option
SysFlowGlobalInt14	Used to inform on the option taken in the Executive for the merge/diverge bias order
SysFlowGlobalInt15	{0/1} True if the definition of the LP area is in process
SysFlowGlobalInt16	Unused
SysFlowGlobalInt17	Unused
SysFlowGlobalInt18	To stop the propagation of the FlowBlockReceive5 message in the block. Each block has to express its constraints only once per LP resolution
SysFlowGlobalInt19	To know if there is a block in the area which requires the extra calculation of an LP with downstream and upstream differentiation
SysFlowGlobalInt20	To count the number of slacks in the LP
SysFlowGlobalInt21	To inform from which type of connector (Flow or Throw/Catch) number the message has been sent (in FlowBlockReceive0 message)

Globals for ARM (Advanced Resource Management)

Advanced Resource Management (ARM) is a complete system for organizing resources, distinguishing between them, and allocating them throughout the model. It provides a convenient and straightforward method for defining complex resource requirements for items as well as a flexible set of rules for how resources get allocated to them. ARM is discussed in the User Reference and in the separate document *Advanced Resource Management Tutorial and Reference*.

Global	Definition during Simulate message
SysGlobalStr3	dB table name used to query if table fields can be expanded
SysGlobal23	Used to pass array pointers to the Resource Manager block
SysGlobal25	To enable updating of locked resource properties
SysGlobal26	To enable remote deletion of resources
SysARMGlobal0-19	Unused

Other reserved global variables

Mainly used for event monitoring and reliability.

Global	Definition during Simulate message
SysGlobal24	To pass array pointers to the Event Monitor block
SysGlobal27	To pass optional argument 1 to the Item Event Monitor block
SysGlobal28	Return value
SysGlobalInt44-52	Reserved for reliability
SysGlobalInt53	Event Monitor block number
SysGlobalInt54-56	Reserved for reliability
SysGlobalInt57	Argument used for passing "EventType" to the Event Monitor block
SysGlobalInt58	Argument used for passing "itemIndex" to the Event Monitor block
SysGlobalInt59	Argument used for passing "blockNumber" to the Event Monitor block
SysGlobalInt60-79	Reserved for reliability
SysDBNGlobalInt0-19	Unused

Integrated Development Environment (IDE)

Debugging

A guide to debugging models and blocks

"We have gone beyond the absurd...
our position is ridiculous."
— John Vacarro

Debugging models

This section discusses ways to debug models and the code necessary to add Trace features to blocks you build. These methods are also useful when you debug block code.

Features that are discussed in the User Reference

The Debugging Tools chapter of the User Reference has a list of blocks that are useful for debugging models. These blocks display values, help validate item flow, and speed debugging of a model. That User Reference chapter also contains other information for debugging models such as showing simulation order.

Adding Trace code

The ExtendSim Trace feature can be useful in debugging a model. If you create your own blocks, those blocks can take advantage of the ExtendSim tracing features. When building a new block, you could include code similar to the following.

For the code that follows:

- For continuous blocks, such as those in the Value library, put the code in the Simulate message handler
- For discrete event blocks, such as those in the Item library, put the 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 "+(myBlockNum-
ber()) + ".  CurrentTime:"+currentTime+".","",True);
   if(getBlockLabel(myBlockNumber()) != "")
      fileWrite(SysGlobal1,"Block Label: "+
                        getBlockLabel(myBlockNumber()),"",True);
....
....
   }
```

☞ The blocks that ship with ExtendSim contain all the necessary Tracing code.

Profiling

Use profiling to determine the amount of time that each block in a model is used, then use that information to optimize the block's code. Profiling a model generates a text file showing the percentage of time individual blocks execute during a simulation. This helps developers who want to determine if they should optimize custom blocks, although non-developers might also use it to find the areas of their models that are most heavily used.

To generate a profile text file, choose the Run > Model Debugging > Profile Block Code command, 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 a Bank Line model might look like:

Block Name	Block Number	Time (seconds)	Percent
Create	0	2.42	9.30
Queue	2	7.65	29.50
Activity	3	2.27	8.80
Activity	5	1.15	4.40
Activity	6	1.82	7.00
Exit	7	4.15	16.00
Line Chart	8	5.83	22.50
Executive	10	0.62	2.40

You can use the information in this profile to look for anomalous results. For example, if one of the three Activity 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. And 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.

Debugging block code without the Source Code Debugger

The ExtendSim Source Code Debugger, discussed beginning on page 172, has a lot of advantages in block debugging, offering conditional breakpoints, stack crawl, and viewing variable values in specified blocks. The following methods may also be useful for debugging code.

Using DebugMsg functions

If you don't want to use the Source Code Debugger, you can use 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.

DebugWrite is the same as the DebugMsg function except that the data is written to a file so the simulation run isn't interrupted. The advantage of using DebugMsg or DebugWrite rather than UserError is that ExtendSim warns if the Debug functions are present when the library is loaded. See the functions in "Debugging" on page 367.

Viewing intermediate results

When developing a block's code, there is an easy way to view intermediate results of 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. Then open the block's dialog, tab to the **Comments** field, and run the 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 (discussed on page 171) to display data, as it doesn't interrupt the model for each new number displayed.

Source Code Debugger

No matter what the language, code often does not work the first time. This section shows how the ModL Source Code debugger can save you time when locating the source of an error in one of your blocks or the equations in equation-based blocks. It provides a tutorial that shows how to step through lines of source code, examine values of variables, create breakpoints with conditions, and analyze block problems. It also discusses the various Debugger windows and dialogs.

In addition to being useful when creating or editing the source code of blocks, the Source Code Debugger is available when using equations in equation-based blocks. See the How To: Debugging Tools section of the ExtendSim User Reference for more information.

Overview of the debugger

A source code debugger makes it much easier to determine the causes of block malfunctions. 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.

Definition of terms

A *breakpoint* is where the debugger will initially stop execution of the block's code and open the Debugger window. You 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 block's dialog and connectors.

The *breakpoint condition* is 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.

☞ See also the descriptions of the Debugger windows and dialogs in "Source code debugger reference" on page 183.

Steps for debugging

As illustrated in the examples that follow, the steps to debug code are:

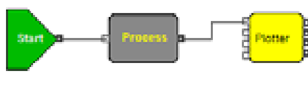
1) Use a menu command to recompile a block, a library, or several libraries in debugging mode.
2) Add breakpoints (indicated by red circles) by clicking on the code markers in the left margin of the Set Breakpoints window. (To remove a breakpoint, click it again.)
3) Add a condition to the breakpoint in the Set Breakpoints window so that it will occur at the appropriate time and model state. Conditional breakpoints display as blue circles.
4) Run the simulation. Or click an item in the block's dialog to execute the code.
5) When the breakpoint is reached, step through the code and examine the variables.
6) After you have resolved the problem, use the menu command to remove the breakpoints and any debugging code from the libraries.

Debugger tutorial

This tutorial shows how to add debugging code to a block and to an entire library as well as how to use the Source Code Debugger to track down and fix an error in a block's code.

The Debug Tutorial model

- Open the Debug Tutorial model located in the Documents/ExtendSim/Examples/How To/Developer Tips folder. As shown to the right, the model consists of three connected blocks which generate data, process the data, and plot the result.



Debug Tutorial model

- Run the model. You should see the error message at the right, followed by a second message that stops the simulation.

Array exceeded dimensional bounds. Variable: array, at line 16. Occurred in [0]Start block during DataOut message at CurrentTime = 8

- Click OK to close those error messages.

Debugging one block

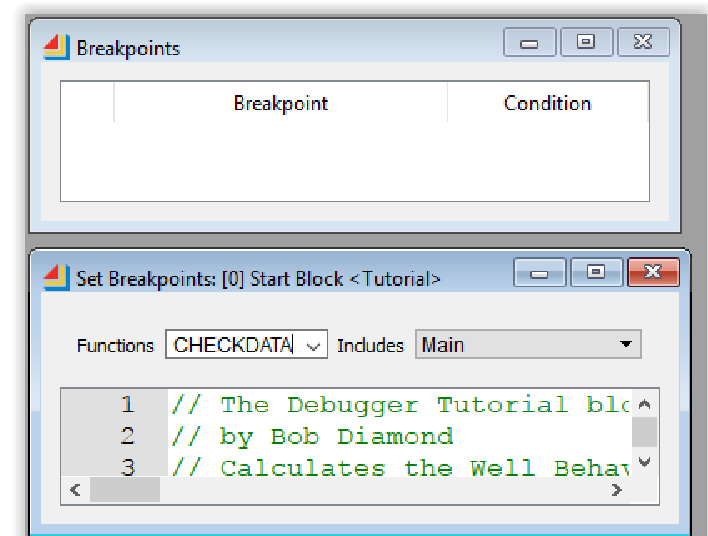
The error message indicates the offending code is in the Start block, so start with setting the block to be in debugging mode.

- On the model worksheet, right-click the Start block and choose **Set Breakpoints and Add Debugging Code** (or select the block and choose that command from the Develop menu).

IDE

This recompiles the block in debugging mode and opens two windows:

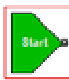
- The Breakpoints window shows the breakpoints for all the blocks in the model
- The Set Breakpoints window is for setting breakpoints in the selected block; in this case, the Start block



These windows are described more in “Source code debugger reference” on page 183.

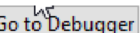
- For now, since the debugger automatically stops at an error message, you don’t need to set breakpoints. Close the Set Breakpoints window

Notice that on the model worksheet and in the Tutorial library window, the Start block’s icon is now surrounded by a red border, indicating that the block is in debugging mode. Blocks in debugging mode will execute more slowly—the red markings are a reminder that you should remove debugging code when finished.

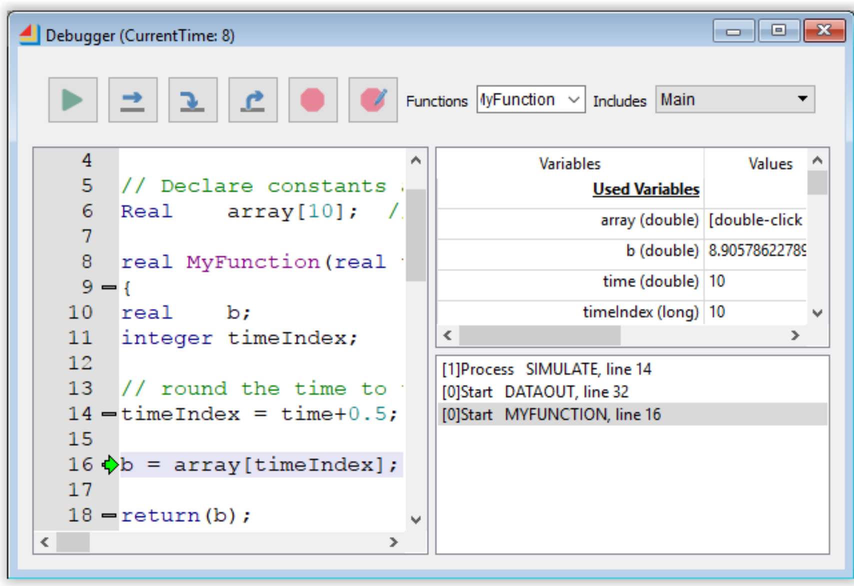


Going to the Debugger

- Run the model again. You'll get the same error message but it now has an Go to Debugger button.



- Click the Go to Debugger button. This opens the Debugger window shown below.



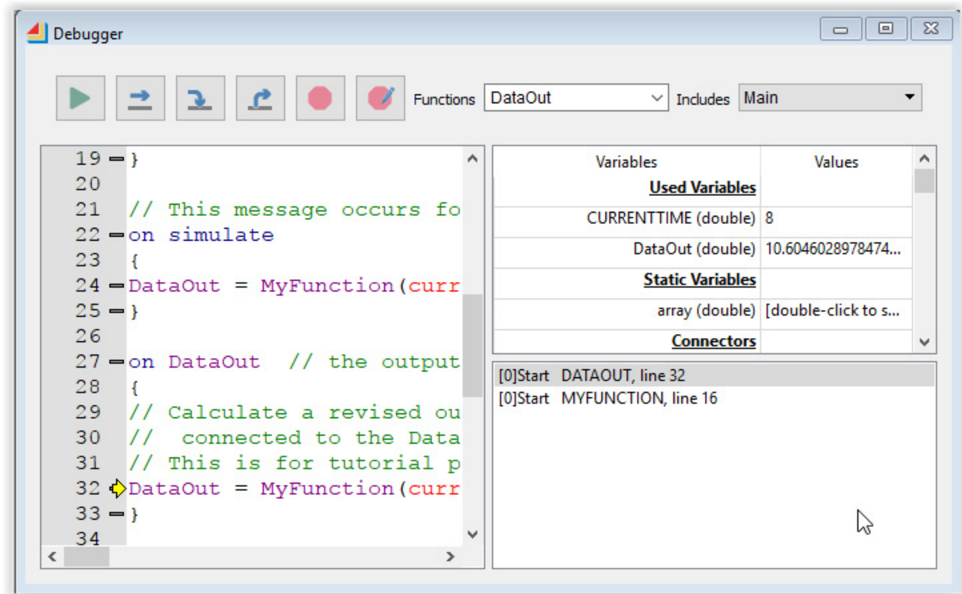
Debugger window after error message

- The title bar indicates that the error occurred at currentTime: 8
- In the *Source* pane at the left:
  - A green *location arrow* in the *Breakpoint margin* on the left points at line 16 ("b = array[timeIndex];").
  - The variable Array is declared at the top of the script and has 10 members (Real array[10]).
- In the *Variables* pane at top right, the index variable *timeIndex* has a value of 10. If you double-click the Value for the Array variable, you will see that the number of members is indexed from 0 to 9.
- The point where the error occurred ([0]Start MYFUNCTION, line 16) is selected in the bottom right pane, known as the *Call Chain* pane.

In order to debug this, you need to find out how MyFunction got called and see the chain of events that led to this error message.

Going up the call chain

- In the Call Chain pane, select the "DATAOUT, line 32" entry (the top entry above the originally selected entry), as shown below.



Debugger window after clicking top entry in Call Chain

Notice that the location arrow in the Breakpoint margin is now yellow for this entry, indicating it is the previous entry in the Call Chain. The selected message handler is called *DataOut*, which is the name of one of the connectors on the block. It indicates that a connector message was received from a block that is connected to the Start block.

To see the code of the block that sent this message, you need to compile it in debugging mode too. Then it can be seen in the Call Chain and you can click its entry to see the code that caused the chain of events. But rather than setting individual blocks to debugging mode, it is often easier to set the entire library.

Debugging an entire library

- If the Debugger window is open, stop debugging or close the Debugger window. That way you won't be in the middle of debugging code when the library is changed.
- Choose the menu command Library > Library Tools > Add Debug Code to Libraries
- In the window that appears, choose the libraries the model uses (in this case, only the Tutorial library) and click **Add Debugging Code**

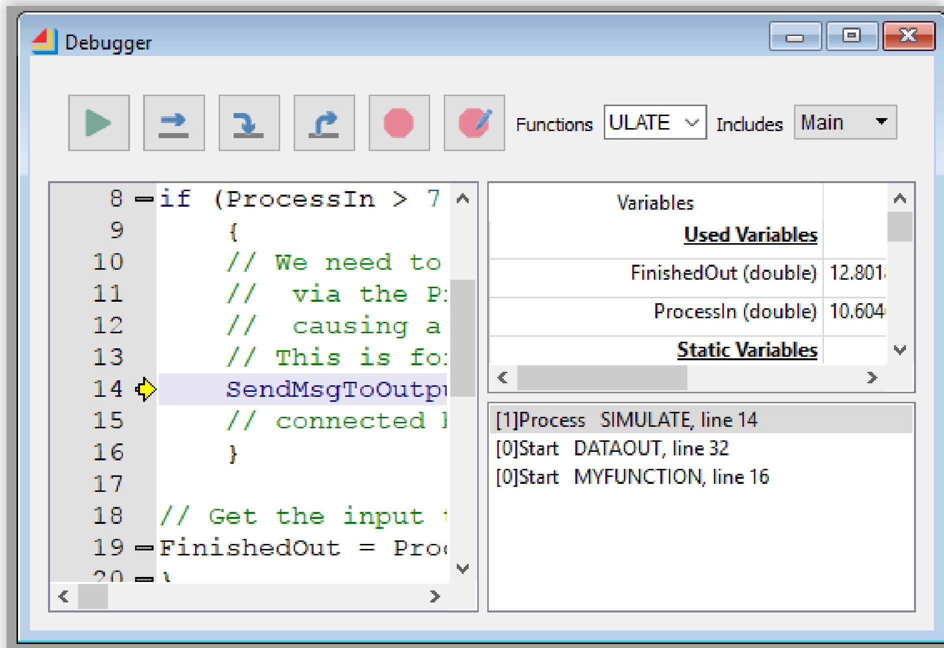
After ExtendSim finishes compiling all of the blocks in the Tutorial library, the three blocks on the Debug Tutorial model will be outlined in red, indicating that they are in debugging mode.

- Run the model again.
- When the error occurs, click the Go To Debugger button to access the Debugger window.

Following the Call Chain

The Source Code Debugger allows you to follow the chain of events that caused the error.

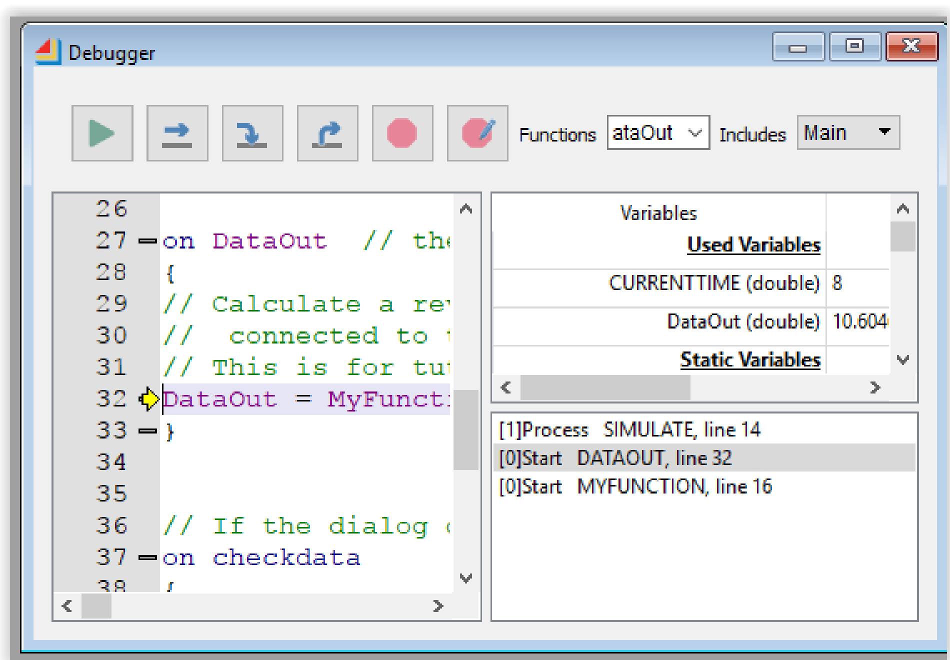
- Click on the top Call Chain entry (Process SIMULATE, line 14); this is one that started this whole chain of events.



Debugger window with top Call Chain entry selected

In the Source pane's Breakpoint margin, the yellow location arrow points to line 14, as indicated in the Call Chain. There was a SendMsgToOutputs function call from the Process block because its input (ProcessIn) was greater than 7.0. (The actual value can be seen in the Variables pane as 10.6046.)

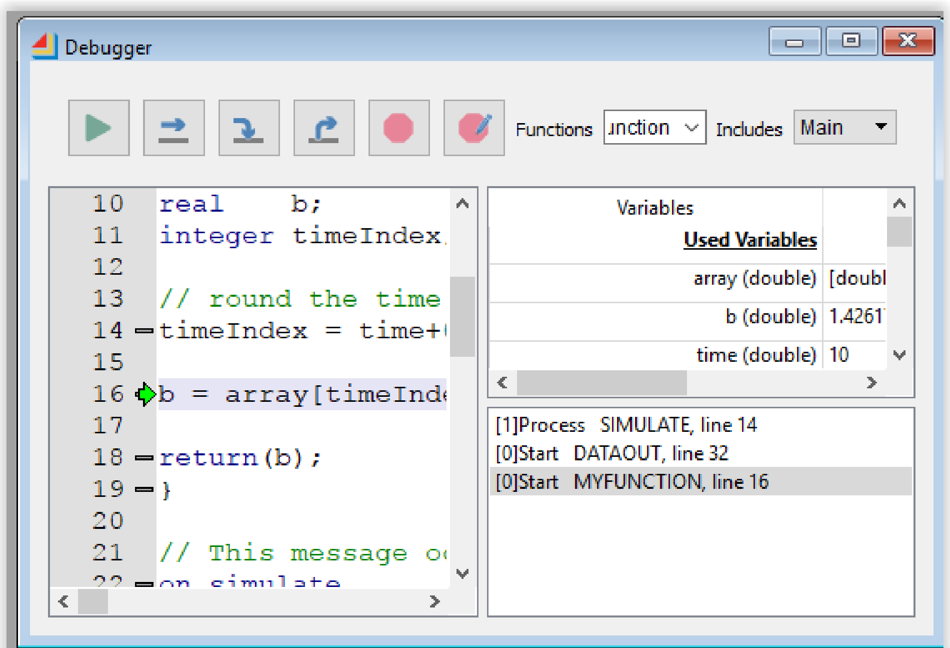
▸ Click on the second Call Chain entry (DATAOUT, line 32):



Debugger window with message handler selected

The Variables pane indicates that CurrentTime is 8. Line 32 of the Source pane shows that the code called MyFunction with an argument equal to10.0 (CurrentTime+2.0).

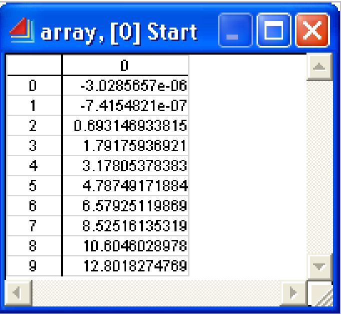
- Click on the bottom Call Chain item where the error occurred (MyFunction, line 16):



Debugger window with error point selected

- Since the error message indicated that Array was indexed beyond its bounds, inspect *Array* by double-clicking its value in the Variables pane.

In the Array window, shown at right, notice that the array is indexed from 0 to 9. However, the code tried to index it with a 10, which is outside its bounds, resulting in an error message.



Array window

Fixing the block's code #1

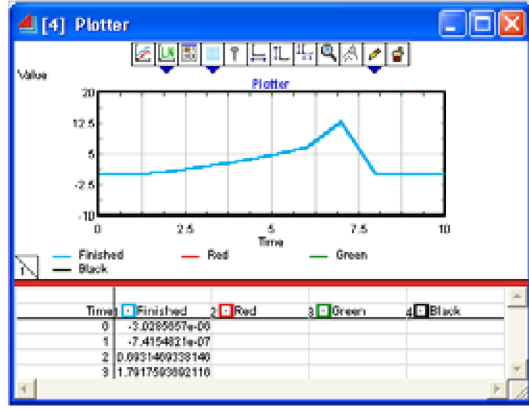
The model is being run from a CurrentTime of 0 to a CurrentTime of 10, and the code could be adding 2.0 to the CurrentTime in some cases. The way to fix the problem is to increase the size of *Array* so that it can be indexed from 0 to at least 12.

- Click the Stop Debugging and Edit Code button at the top of the Debugger window.

This takes you to the location of the error in the Start block's structure window

- Change the declaration of Array[10] to Array[13]
- Close the block's structure window and compile the block

- Run the model again and note that this time there is no error message. However, the plotter shows a drop off in values around time 7 and you know from experience that the Smedley function is supposed to be increasing.



To see what the problem is you need to set breakpoints.

Setting breakpoints

- To open its Set Breakpoints window, right-click the Start block on the model worksheet and select **Set Breakpoints and Add Debugging Code**.

(Since the entire library already has its code set for debugging, this just reopens the Start block's Set Breakpoints window.)

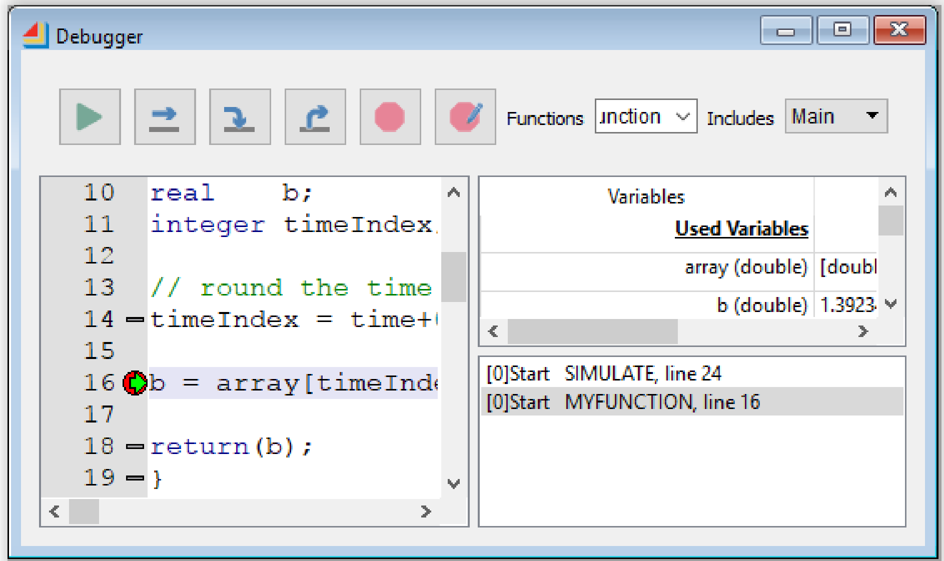
- To set a breakpoint, go to the Breakpoints margin on the left side of the Set Breakpoints window and click on the *code marker* for line 16—the location of the *B = array[timeIndex]* statement.

Start block debugging window

A red circle appears where you clicked. Running the model will now cause the simulation to break (pause) at that point, opening the Debugger window.

- Run the model

The Debugger window opens and the breakpoints margin has a green arrow on top of a red circle. The green arrow indicates that this is the part of the code that will be executed next; the red circle indicates that there is also a breakpoint there.



Breakpoint reached

In the Variables pane, timeIndex is 0, which means the simulation is stopped at the beginning.

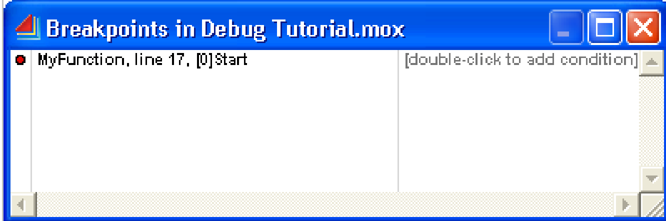
- You can either:
  - Click the **Continue** button in the Debugger's toolbar 10 times until the breakpoint occurs at timeIndex 10
  - **Or**, set conditions as described below

Setting conditions

Clicking until a timeIndex of 10 is reached can get tedious. Instead, set a condition on the breakpoint so the code only breaks when timeIndex is greater than or equal to 10.

- Close the Debugger window or click the **Stop Debugging** button in the Debugger window's toolbar (this also stops the run)

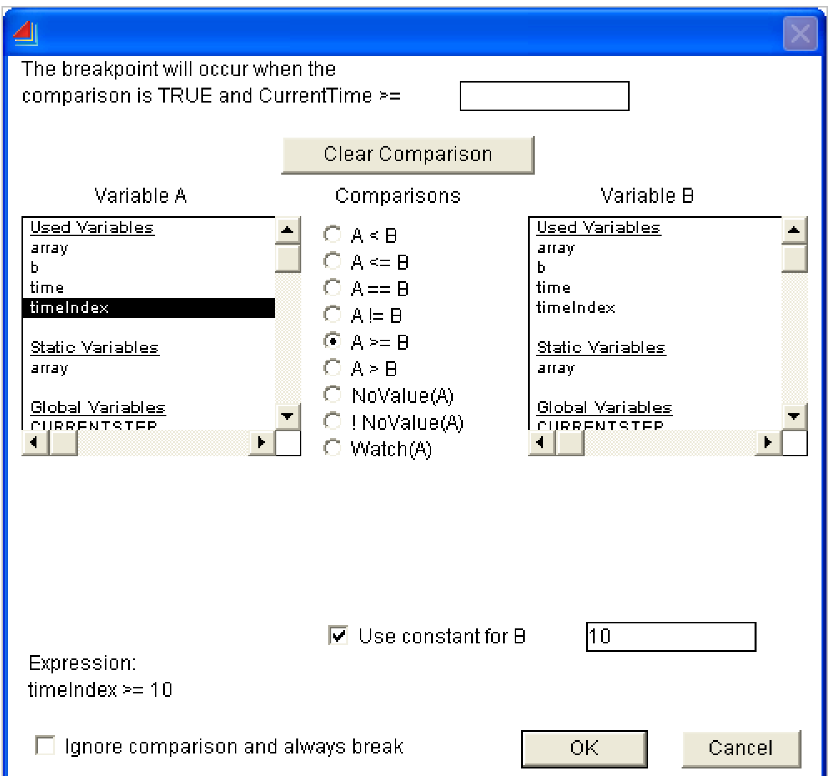
- Choose **Develop > Open Breakpoints Window** (or bring the Breakpoints window forward if it is open)



Breakpoints window showing the breakpoint

- The Breakpoints window has two columns: Breakpoint and Condition. Double-click in the Condition column that relates to the breakpoint (in this case, there is only one breakpoint).

A new window appears for setting conditions:



Condition window

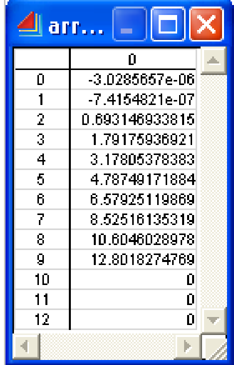
- In this window, do the following:
  - In the Variable A column, select **timeIndex (long)**
  - For the comparison, choose the radio button **A>=B**

- Below the Variable B column, check the checkbox **Use constant for B** and enter **10** as the constant
- Click OK to save your changes and close the Breakpoint Conditions window
- Run the model again
- When the breakpoint occurs, click the *Step over* button in the Debugger window's toolbar to see the value of b over time. (Note that, in this case, the *Step over* and *Step into* buttons do the same thing, because there is no function call to step into.)

The calculated value of b at timeIndex 10 is zero. This should not happen because, as noted earlier, we know that the value of the Smedley function should increase over time, not decrease.

Fixing the block's code #2

- Double click the Array variable to look at its values – the result is shown at the right.



	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

In the array, elements 10-12 are zero, indicating that the Smedley values for the new larger array have not been calculated. The code should be changed so that no matter what the size of the array is, it will be filled.

Changing the block's code

- Click the *Stop Debugging and Edit Code* button in the Debugger's toolbar. This closes the Debugger and opens the Script tab of the Start block's structure.
- In the Start block's code, change the Initsim message handler (line 43 to line 50) to read:

```
on initsim
{
integer i, length;
length = GetDimension(array); // get length of array
// initialize the array so we don't have to recalculate it
for (i=0; i<length; i++)// use length to limit loop
     array[i] = Log(GammaFunction(i+1));
}
```

Instead of hard coding i <10, this code calls a function to return the array length into a new variable called *length* which is used to limit the calculation loop. Then any time the size of the array is changed the entire array will be filled.

- Close the block's structure and Save and Compile the block.

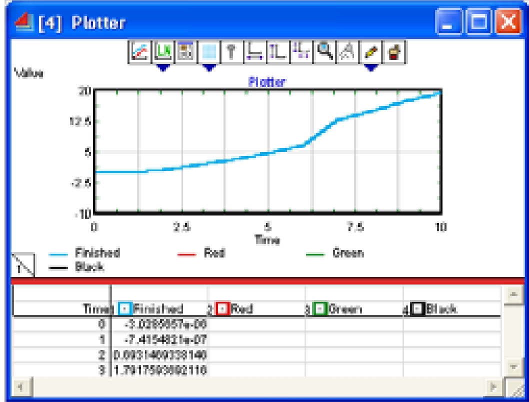
Disabling and removing breakpoints

To temporarily disable a breakpoint, click once on the red circle in the margin of the Breakpoints window. This turns the red circle into an empty circle.

To remove a breakpoint, select the breakpoint's name in the in the Breakpoints window and delete it. Or click once on the red circle in the left margin of the Debugger window.

Since the Debugger window has closed:

- Remove the breakpoint by selecting the breakpoint information (MYFUNCTION, line 16, [0]Start) in the Breakpoints window and clicking the Delete key
- Run the model again to see the correct output:



The well-behaved Smedley function

☞ Making Array a dynamic array and then resizing it in InitSim would allow the model to be run for any EndTime value without overrunning Array.

Source code debugger reference

The following sections discuss the Debugger and its windows and dialogs.

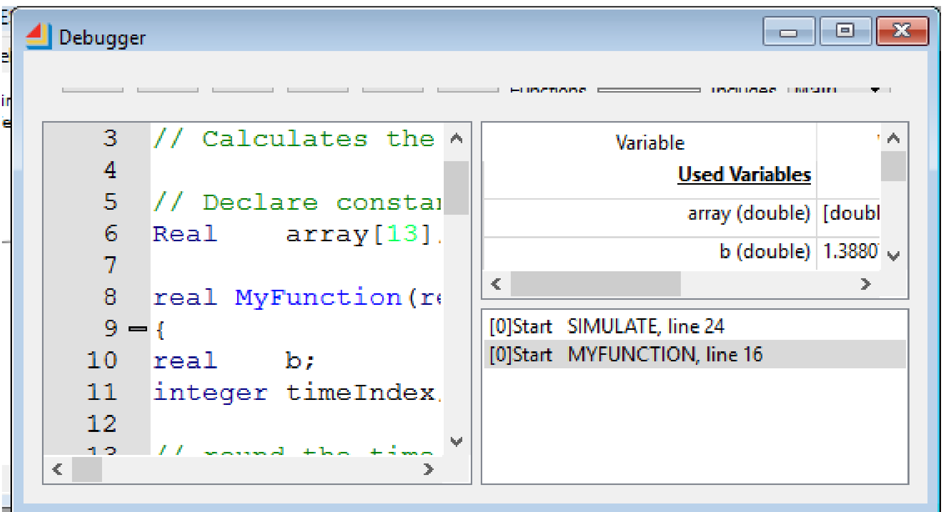
Setting a block or library to be in debugger mode

In order for you to use the Source Code Debugger, ExtendSim has to generate debugging code for a block or blocks.

- To add debugging code to one or a few blocks, select the block or blocks on the model worksheet and choose the command Develop > Set Breakpoints and Add Debugging Code (or right-click the block). This automatically generates debugging code for the selected blocks and opens a Set Breakpoints window for each block selected.

- To add debugging code to an entire library, choose the command Library > Library Tools > Add Debug Code to Libraries. (This takes longer than setting a block to debugging mode. But it is especially useful if you are trying to debug blocks in a discrete event model, since the messages will be traceable back to their originators via the Call Chain in the Debugger window.) After the library is in debugging mode, right-click the blocks on the model worksheet that you want to add breakpoints to, or select them and choose the Develop > Set Breakpoints and Add Debugging Code command.

Debugger window



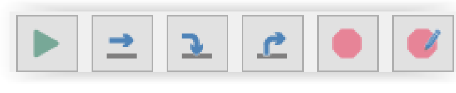
When the model is run and execution reaches the breakpoint, the Debugger window opens and shows the ModL code, values of variables, the Call Chain, and (in the title bar) the current-Time.

☞ The only way to access the Debugger window is to run a model with one or more blocks in debugging mode, where there is also either an error message or breakpoints that have been set.

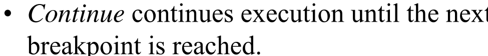
You can set a breakpoint in a block's Debugger or Set Breakpoint window. However, since the Debugger window is only open during the model run, it is more common to set breakpoints in the Set Breakpoint window.

Toolbar

There are six tools at the top of the Debugger toolbar. From left to right, they are:



Debugger toolbar



- *Continue* continues execution until the next breakpoint is reached.
- *Step Over* executes a function call without stepping into the function code. It is used when you want the debugger to execute the line of code with a function call, but not to trace the code of the function.
- *Step Into* steps into the function call. It is used when you want the debugger to trace the actual function call, including the code of the function.
- *Step Out* continues execution until it gets to the caller of the function. It is used when you want to complete the function or message handler that is currently executing and step back to the calling function.
- *Stop Debugging* stops the debugging session and returns to the ExtendSim program.
- *Stop Debugging and Edit Code* stops the execution of the block, opens the block's structure window, and positions the cursor at the point where execution stopped. This is particularly useful when you have found the error and want to edit the code to make a correction.

Popup menus

- The *Functions* popup menu can scroll to any function and shows which function you are in.
- The *Includes* popup menu shows which include files are used in the block. It allows you to set breakpoints in an include by displaying its contents in the source pane.

Margin indicators in the Debugger window

At the left of the Debugger window, the Breakpoints margin provides the following indicators:

- Active, unconditional breakpoints are shown as a red circle
- A blue circle indicates an active breakpoint that has a condition
- An empty white circle indicates the breakpoint is temporarily disabled
- A green location arrow indicates where the code is going to execute next
- A yellow location arrow indicates where the previous entry in the call chain is going to execute

To add breakpoints, click on one of the code markers in the Breakpoints margin. To remove a breakpoint, click it again. To add a condition to a breakpoint, use the Breakpoints window, shown on page 186.

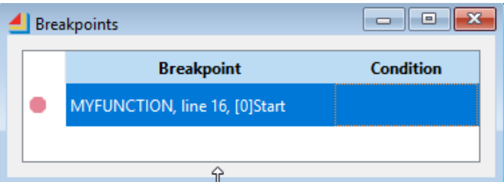
Set Breakpoints window

```
Set Breakpoints: [0]   Start
Functions [          ]   Includes  Main

// 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
timeIndex = time+0.5;

b = array[timeIndex];    // output the pre-calculated Smedley

Line: 1    Code Completion (F8)
```

This window shows the ModL code and the breakpoints for the indicated block. The Breakpoints margin, with code markers and any breakpoints, is on the left. The popup menus at the top of this window are the same as for the Debugger window on page 184.

To add a breakpoint, click on a code marker in the Breakpoints margin. This places a red circle on the code marker, as seen above. To remove the breakpoint, click its red circle in this window once so it returns to a code marker, or delete the breakpoint from the Breakpoints window, discussed below.

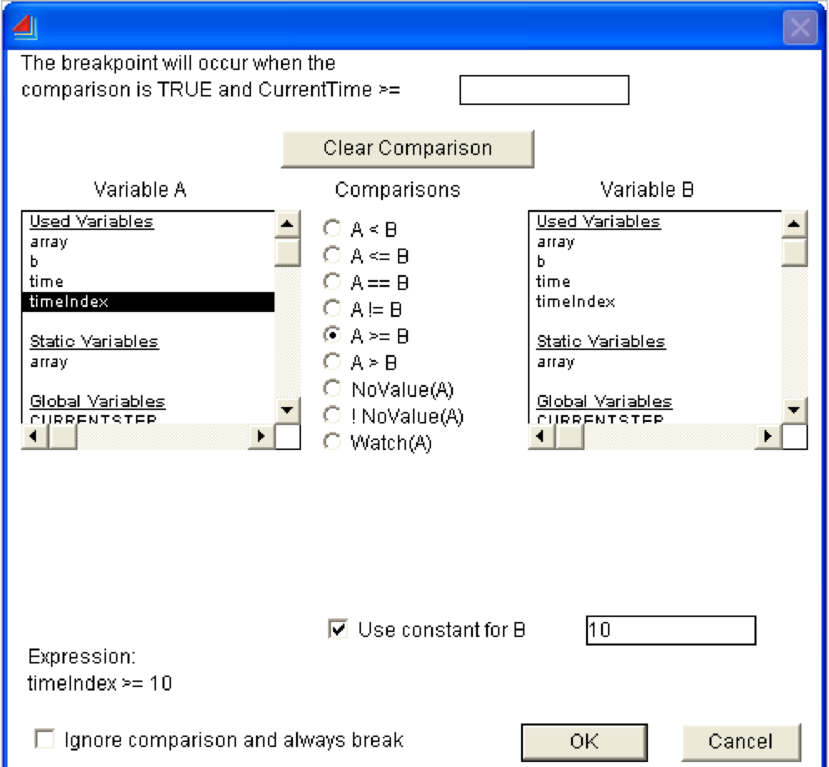
Breakpoints window



The Breakpoints window shows breakpoints for the entire model as well as any conditions for those breakpoints. It is used for enabling, disabling, and deleting breakpoints and for adding conditions to breakpoints. For a model with blocks in debugging mode, the Breakpoints window is opened by the command Develop > Open Breakpoints Window.

- A red circle indicates an enabled breakpoint.
- To temporarily disable a breakpoint, click the red circle once– it becomes an empty white circle. To return the breakpoint to active status, click the empty circle once.
- To delete a breakpoint, select its name from the Breakpoint column and click the Delete key.
- To enable a condition for a breakpoint, double-click in its Condition column, opening the Breakpoint Conditions dialog.

Breakpoint Conditions dialog



Conditions allow a breakpoint to be ignored unless the condition is TRUE. This is useful when you 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 Breakpoint Conditions dialog makes it easy to construct a conditional breakpoint with no coding. You can enter a *currentTime* value and/or any other comparison that might be helpful in narrowing down the problem.

Accessing the dialog

To access the Conditions dialog, double-click in the Condition column of the Breakpoints window shown on page 186.

To enter a comparison:

- Select a variable from column A
- Click the desired comparison operator radio button
- Choose a variable from column B or check the *Use constant for B* check box and enter a constant in that field

Optionally you can enter a currentTime value and/or click the *Ignore comparison and always break* check box so that your entered condition will be saved but ignored for the present.

WatchPoints

A WatchPoint condition detects when the variable A changes value and will break to the Debugger whenever that occurs. To use this, select *Watch(A)* from the comparison operator list between the Variable A and Variable B columns. This is most useful in finding a statement in a different block that is changing a variable incorrectly, and the statement cannot be found easily by normal means.

In order for watchpoints to work correctly, all blocks in a library should be compiled with debugging code turned on. WatchPoints slow model execution as they have to be checked continuously while code is executed.

Arrays

If a variable used in a condition is an array, you need to specify which cell is being referenced. Do this by filling out the Array Indexes section below the Variable A and/or Variable B columns, as appropriate. For example, to watch the 2nd row and 3rd column of MyArray, you would enter [1][2] in the Array Indexes field.

Ignore conditions checkbox

Check *Ignore conditions and always break* when you've met the condition once and may not meet it again, but you want to keep exploring that breakpoint. This saves the conditions so you can use them later.

Debugging tips

- To quickly debug a block, select the block and choose Develop > Set Breakpoints and Add Debugging Code. 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 > Library Tools menu. That way the Call Chain will contain the entire chain of messages and it will be obvious which block sent what to whom.
- When a breakpoint occurs, click the *Step over* button in the Debugger toolbar to see how the values of the variables change. To jump to a later CurrentTime, go to the Break Point Conditions dialog.
- When finished debugging, you should remove all of the debugging code as it slows execution considerably. To do this, choose Library > Library Tools > Remove Debug Code from Libraries.
- Be proactive and use the Debugger to step through code even when you think it is working properly. Try different cases just to make sure that the block is working as you have intended. This will give you confidence in what you have written.

Variables, Messages, & Functions

ModL Variables

A detailed description of the ModL variables
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 includes a complete list and description of the system and global variables.

- System variables provide information about the state of the simulation
- Global variables are used to pass information between blocks

System variables

System variables give you information about the state of the simulation. They are declared by ExtendSim and can be viewed or modified by any block in a model.

☞ You can read or write to these variables, but you should be careful when writing to any of them.

Table of system variables

Name	Description
AnimationOn	Tells the state of the Run > Show 2D Animation command. 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 speeds simulations by preventing needless animation when the modeler can't see it anyway.)
AntitheticRandomVariates	If TRUE, ExtendSim's random number functions generate antithetic random numbers.
CurrentScenario	In models where the Scenario Manager is running a series of scenarios, this is the current scenario number. This will be equal to the row in the Scenarios table that is currently providing the factors to the model.
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 –ExtendSim 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.
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.
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 (Item 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 or discrete rate simulation.

Name	Description
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 or discrete rate simulation runs out of items or events.
GlobalProofStr	Set by the Proof Animation code of a block.
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 ExtendSim User Reference for a discussion about the random number generator.)
MovieOn	This legacy variable is currently unused.
NumScenarios	In models where the Scenario Manager is running a series of scenarios, this is the total number of scenarios that will be run.
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 or discrete rate simulation.
OleGlobal, OleGlobalInt, OleGlobalStr	These variables are set by the external application sending an OLEAutomation message to a block. They act as arguments that the block can access when it gets the message.
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 Simulation 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

Global variables are useful for passing information between blocks. They are predefined by ExtendSim and stored within the model. They can be used by any block anywhere in a model. The ExtendSim application never changes the values of global variables except to initialize them when a new model is created.

Types

There are two types of global variables

- General use global variables have a name that starts with “Global”. They can be used any way you want.
- Reserved global variables start with “SysGlobal”. These System Globals are controlled by the libraries and features that are included with ExtendSim and their use is reserved.

Never use the SysGlobal variables for your own purposes; always use the general use globals (Global, GlobalInit, and GlobalStr) instead.

General use global variables

The following global variables are available for your use in equations or when creating blocks.

Name	Type	Use
Global0 through Global19	Real	General
GlobalInt0 through GlobalInt99	Integer	General
GlobalStr0 through GlobalStr9	String	General

Reserved global variables

The system globals, which start with “Sys”, are reserved by ANDRITZ Inc. for our internal purposes.

You can use the SysGlobal variables (for example, when creating discrete event or discrete rate blocks), but you should only use them in the same way they are used by the ExtendSim application.

☞ See “Globals in discrete event blocks” on page 160, “Globals in discrete rate blocks” on page 165, and “Globals for ARM (Advanced Resource Management)” on page 167 for tables that describe how ExtendSim uses the reserved globals during the Simulate, CheckData, and InitSim messages.

Name	Type	Use
SysGlobal0 through SysGlobal39	Real	Reserved for ITI
SysGlobal*Int*0 through SysGlobal*Int*99	Integer	Reserved for ITI
SysGlobal*Str*0 through SysGlobal*Str*19	String	Reserved for ITI
Sys*Flow*Global0 through Sys*Flow*Global4	Real	Reserved for ITI
Sys*Flow*Global*Int*0 through Sys*Flow*Global*Int*29	Integer	Reserved for ITI
Sys*Flow*Global*Str*0 through Sys*Flow*Global*Str*4	String	Reserved for ITI
Sys*DBN*Global*Int*0 through Sys*DBN*Global*Int*19	Integer	Reserved for ITI
Sys*ARM*Global*Int*0 through Sys*ARM*Global*Int*19	Integer	Reserved for ITI

Variables, Messages, & Functions

Messages and Message Handlers

A detailed description of the ModL messages

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

Messages and message handlers were introduced on page 33 and discussed more in “Message handlers” on page 75 and “Using message handlers” on page 111.

The following table summarizes each category and type of ModL message. The three categories of messages (application, user interaction with the dialog, and block-to-block) are discussed on page 111. Note that while messages can originate either from the ExtendSim application or from blocks, it is always a block that is on the receiving end of a message.

Summary of messages

Category	Type	Purpose	Page
Application	Simulation	Sent to blocks in simulation order during a simulation run	194
Application	Model Status	Sent to all blocks in a model when the model changes state	196
Application	Block Status	Sent when clicking on a block or connector, when placing, deleting, moving, or pasting a block, or when a model is opened on a different computing platform.	197
Dialog activity	Dialog	Sent when the modeler interacts with a block’s dialog	199
Block to Block	Connector	Sent via the Connector Message functions, or by the system when a modeler manually changes the number of variable connectors, or when the modeler connects or disconnects a block	201
Block to Block	Block to block	System and user-defined messages sent from a block’s ModL code when communicating information to other blocks	202
Application	Dynamic Link	Sent to subscribed blocks when there is a change to the ExtendSim database or global array part that they are linked to	203
Application	OLE	Sent in response to OLE conditions	203

Simulation messages

These messages are sent to all blocks during the simulation run *in the following order*:

Message	When sent
ModifyRunParameter	Sent before the simulation run begins to allow changing the Simulation Setup parameters via the SetRunParameter() function (see page 299). Note that this is sent only once even for multiple simulation runs.
SimStart	Sent after ModifyRunParameter and before all other simulation message handlers. See the BlockSimStartPriority() function on page 291 for details.
PreCheckData	Sent to prepare blocks for CheckData, below. Most blocks can ignore this message.

Message	When sent
CheckData	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 ExtendSim 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* (any non-zero) value and unconnected ones have a *false* (zero) value. This makes it easy to check whether a block is connected properly before running a simulation.
StepSize	After all CheckData messages. If the code needs a particular StepSize, set it here. In continuous models, set the DeltaTime system variable to the maximum step size. ExtendSim queries all blocks and uses the smallest DeltaTime value returned. This message is also used to set up 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), 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. (Note that ContinueSim, below, is sent instead of InitSim when continuing a saved model's run.)
ContinueSim	This message is sent instead of InitSim if you are continuing a saved model's run. Set up any variables for continuing a saved model run.
PostInitSim	Sent to give blocks another chance to initialize variables that could not be initialized until after the InitSim was sent to all blocks. Most blocks can ignore this message.
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, ExtendSim 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 and discrete rate simulations*, the first Simulate message gets sent with CurrentTime = StartTime and CurrentStep = 0. For subsequent Simulate messages, the Executive block (Item library) controls how CurrentTime is advanced. See the GetSimulateMsgs() function for how to turn off simulate messages in a block under certain conditions.
AbortSim	Sent only if an Abort occurred during the Simulate messages. This notifies the blocks to clean up and lets them know that an abort occurred.
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.
FinalCalc2	Sent to all blocks after FinalCalc completes.

Message	When sent
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 category (see “Block categories” on page 55), ExtendSim cycles through each block category 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 the modeler or the block’s code aborts the simulation.
SimFinish	Sent after all other simulation message handlers. See BlockSimFinishPriority() function at page 291 for details.

Model Status messages

When the status of the model changes, these messages are sent to all blocks:

Message	When sent
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 2D Animation command changes state. This is useful to change a block's appearance according to the value of the AnimationOn variable.
CloseModel	Right before the model closes, this message is sent to all blocks in the model to let them know that the model is closing.
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.
OldFileUpdate	Before the openModel message, if the file version is older than the application version.
OpenModel	Sent when a model is opened, or to a new hierarchical block that has just been placed on the model worksheet. 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.
OpenModel2	Sent after all of the blocks have received the OpenModel message, or after the OpenModel message to a new hierarchical block that has just been placed on the model worksheet. 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.
PauseSimulation	Sent to ALL blocks when the modeler pauses the simulation. Contrast this to ResumeSim which is sent only to the block that the modeler has changed, and ResumeSimAllBlocks sent to all of the blocks in the model.

Message	When sent
ResumeSim	This message allows the ModL code to respond to changes before continuing the simulation. It is sent when the modeler: • Either clicks dialog buttons during the simulation • Or, edits a parameter and chooses Run > Resume (or clicks the Resume button in the simulation status bar). Note: This message is sent only to the blocks that have had dialog values edited.
ResumeSi-mAllBlocks	This message allows the ModL code to respond to changes before continuing the simulation. It is sent when the modeler: • Either clicks dialog buttons during the simulation • Or, edits a parameter and chooses Run > Resume (or clicks the Resume button in the simulation status bar). Note: This message is sent to all of the blocks in the model, as opposed to ResumeSim, above, that is sent only to the blocks edited.
SimOrder-Changed	Sent when the modeler changes connections, connects a new block, deletes connections or blocks, or does anything that changes the simulation order.
SimSetup	Sent when the modeler changes anything in the Simulation Setup dialog.

Block Status messages

These messages are sent to individual blocks:

- When the block is clicked
- When a block is placed, deleted moved, or pasted
- When a block needs to communicate information to the model

Message	When sent
BlockClick	Sent when the modeler clicks on a block. Use GetMouseX(), GetMouseY(), and GetBlockTypePosition() to find out what portion of the block was clicked. See the Mandelbrot model (Documents/ExtendSim/Examples/Continuous/Custom Block Models) for an example.
BlockIdentify	Reserved for use by ANDRITZ Inc.
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	Upon completion of a move, sent to all blocks that moved. (See “Scripting” on page 300.)
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 (Value library) for an example. Call only the GetFileReadVersion() and ResizeDTDuringRead() functions in this message handler.
BlockRight-Click	Sent when a block is right-clicked. Usually used to create a custom popup menu. See the Math block (Value library).

Message	When sent
BlockSelect	Sent when a block becomes selected. See also BlockUnselect.
BlockUndelete	Sent to a block when its deletion is undone using the Edit menu Undo command.
BlockUnselect	Sent to a block when it becomes unselected. See also BlockSelect.
Connection-Make	Sent to all newly connected blocks when the modeler makes a connection on the model worksheet. For hierarchical blocks, this is sent to all internal blocks. To prevent the connection from being completed, you can call the *Abort* statement.
CloneInit	When a clone is placed on the model, sent to the block that owns the clone so that it can be re-initialized. This is useful for static text that needs to be re-initialized when cloned.
EquationCom-pilePlatform	When an equation tries to execute and it is compiled on the wrong platform, this message handler should just recompile the equation to fix it.
ConnectorRi-ghtClick	Sent when the modeler right-clicks a connector. Used, for example, in the Item library to create a popup menu that is used to add History blocks to a connection when that connection is right-clicked.
Connector-ShowHide	Sent when the modeler hides or shows the connectors, either from the toolbar or the Model menu command.
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 modeler deletes a block from the model. For hierarchical blocks, this is sent to all internal blocks.
DeleteBlock2	Sent to blocks after their connection lines to other blocks have been deleted, right before the block deletion occurs.
DragCloneTo-Block	This message gets sent when a modeler 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 Optimizer, Statistics (Report library) and the Scenario Manager and Find & Replace blocks (Value library).
HBlockClose	When a hierarchical block's submodel or structure is closed. For hierarchical blocks, this is sent to all internal blocks.
HBlockFrom-Library	Sent to all enclosed blocks when the H-block from a library is placed on the model. The OpenModel and OpenModel2 messages are then sent to the H-block.

Message	When sent
HBlockHelp-Button	This message is sent to all the blocks inside a hierarchical block when the H-block Help button is clicked. This can be used to intercept the Help button message for the H-block, 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 H-block, and prevent the Help text from opening up.
HBlockMove	Sent to all enclosed blocks when the enclosing H-block is moved.
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.
HBlockSave-ToLibrary	Sent to all enclosed blocks when the H-block structure is saved to a library.
HBlockUpdate	Sent to all enclosed blocks when the H-block structure is edited and closed.
HelpButton	Sent when the modeler clicks the Help button in the block's dialog. Executing an Abort statement during this message handler will prevent the ExtendSim Help from opening.
IconView-Change	Sent when the modeler changes the icon view or when a ModL function call from a block changes the icon view. If this message is stopped with an *abort* statement, the change is not made to that view.
MakeSelec-tionHierarchi-cal	Sent after a selection of blocks is made into a hierarchical block using the menu command.
MailSlotRe-ceive	Sent repeatedly when there are mailslot messages waiting to be picked up. See the mailslot functions for more information.
PasteBlock	When the modeler pastes a block onto the model. For hierarchical blocks, this is sent to all blocks within the hierarchical block.
PasteBlock2	Sent after the PasteBlock message has been received by all the blocks pasted.
PasteBlock3	Sent after the PasteBlock2 message has been received by all the blocks pasted.
Plotter0Close, Plotter1Close, Plotter2Close, Plotter3Close	Sent when the modeler closes a plotter window.
PlotProperty-Change	Sent to the plotter block when a user makes a change to a plot's properties.
TimerTick	Sent by the Timer chore started by the StartTimer function. (See the scripting functions.)

Dialog messages

Dialog messages are the names of dialog items and are sent to the block whenever the dialog is used. When a button in a dialog is clicked or a parameter is unselected (for example, after it has

been changed), ExtendSim sends a message with the same name as the dialog item to the ModL code.

For example, assume a block has a “Count” button in its dialog. When that button is clicked, ExtendSim sends the “Count” message to the block. If the block has an “on Count” message handler, it will be executed; if not, nothing happens.

Names for all the named dialog items in a block are listed in the Variables pane at the lower left of the structure window. If a dialog item (such as static text) does not have a dialog item name, it will not be listed in the Variables pane.

Message	When sent
AbortDialog-Message	If the modeler 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.
Cancel	When the Cancel button is clicked. This restores the block to the state it was in the last time it was opened, discarding changes to the dialog To keep this behavior, do not change the name of the dialog item from “Cancel”.
CellAccept	Sent to a block when the modeler 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.
DataTable-Hover	Sent to the block when the cursor is hovering over a data table.
DataTableResize	Sent when the datatable resize button is clicked. Use this so the block can be alerted to the new size and inform the modeler if the new size is not acceptable. Use the function WhichDialogItemClicked to determine which datatable resize button has been clicked. Use an Abort statement to prevent the resize.
DataTable-Scrolled	Sent to the owning block when one of its data tables is scrolled.
DialogClick	Sent when the modeler 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 modeler.
DialogClose	When the OK or Cancel buttons or the close box are clicked.
DialogItemRefresh	Sent when a dialog item becomes visible, if it was set up by calling the SetVisibilityMonitoring() function.
DialogItem-ToolTip	Sent when the cursor hovers over a dialog item so that the block can customize the tool tip (e.g format a value in a special way) that the modeler sees.
DialogOpen	When the block’s dialog is opened. 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 also Plotter I/O block code.
OK	When the modeler clicks the OK button. No need to do any special handling in this section unless you want to check input data. Use an Abort statement to prevent the dialog from closing if data is not acceptable.

Message	When sent
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. Use an Abort statement to prevent the tab switch.
YourButton	If you have created a button, radio button, or check box named "YourButton," this message handler is activated when the button is clicked. Use an Abort statement to prevent the action if necessary.
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. Use an Abort statement to prevent the value from changing.

Connector messages

These messages are sent to individual blocks:

- Via the Connector Message functions
- By the system when a modeler manually changes the number of variable connectors
- When the modeler connects or disconnects a block

☞ The connector functions start on page 260.

Message	When sent
ConArray-Changed	Sent continuously when a modeler drags a variable connector to increase or decrease the number of connectors in its array. Call ConArrayChangedWhichCon() to return the name of the connector, and then call ConArrayGetNumCons() to find out how many connectors there are in the dragged connector. Abort this message handler to prevent the modeler from adding too many or too few connectors.
ConArray-ChangedComplete	Sent when the modeler finishes dragging a variable connector, to allow the block to see the final number of connectors chosen.
ConArrayCollapseChanged	Sent when the modeler collapses or expands the variable connectors.
Connection-Break	Sent to all connected blocks when a connection is deleted. Note that a block can receive multiple ConnectionBreak messages when blocks or right angle connections are deleted.
Connection-Click	When a connection line is clicked, sent to all blocks connected so that they can react (e.g. report a value). The function ConnectorToolTipWhich() returns the connector number for that connection.

Message	When sent
Connection-Make	Sent to all newly connected blocks when the modeler makes a connection on the model. For hierarchical blocks, this is sent to all internal blocks. The function ConnectorToolTipWhich() returns the connector number for that connection. To prevent the connection from being completed, you can call the Abort statement in this message handler.
Connector-Name	Connector message. When the connector on a connected block receives a message from another block using the connector message functions.
ConnectorRightClick	Sent when the modeler right-clicks a connector. Used, for example, in the Item library to create a popup menu that is used to add History blocks to a connection when that connection is right-clicked.
ConnectorTool-Tip	Sent when the cursor hovers over a connector so that the block can customize the tool tip (e.g format a value in a special way) that the modeler sees.

Block to block messages

Model type dependent and user-defined messages sent from a block's ModL code to communicate information to other blocks:

Message	When sent
AttribInfo	Sent by the Executive block when attribute information has been changed.
BlockReceive0 through 9	Used by the Discrete Event library as system messages.
ClearStatistics	When a block's statistical variables need to be reset. This message is typically sent by a Statistics block (Report Library). See the Activity block (Item library) for an example of receiving this message.
DEExecutive-ArrayResize	Sent by the Executive block to notify Item or Flow blocks that item arrays have been resized.
BlockTableInfo	Sent by some of the blocks to query data table size. Not sent by ExtendSim.
ProofAnimation	This message is used by the blocks that are interacting with Proof Animation to produce proof animation functionality.
QueueFunction	Sent by the QueueTools block (Utilities library) and the Queue blocks (Item library) to communicate with each other.
ShiftSchedule	Sent by a block to all the blocks in the model when a shift schedule has been changed.
UpdateStatistics	When a block's statistical variables need to be recalculated and updated. This message is typically sent by a Statistics block (Report library). See the Activity block (Item library) for an example of receiving this message.
UserMsg0 through 19	User-defined message that is not used by any ExtendSim libraries. Use the SendMsgToBlock function to send these messages from another block.

Message	When sent
FlowBlock-Receive0-FlowBlock-Receive19	Used by the Rate library to communicate information between blocks.

Dynamic Link messages

When the part they are linked to changes, these messages are sent to individual blocks that are subscribed to an ExtendSim Database or global array.

☞ Database functions start on page 318; global array functions start on page 342.

Message	When sent
LinkContent	If linked or subscribed to part of an ExtendSim Database or global array, this is sent when the data changes within that part. This facilitates recalculation only when the data changes.
LinkStructure	If linked or subscribed to part of an ExtendSim Database or global array, this is sent when the structure (e.g. number of fields, records, names, etc.) changes within that part. Use the DILinkUpdateInfo() and DILinkUpdateString() functions to find out what changed.

OLE messages

These messages that are sent to an individual block on particular events.

Message	When sent
AdviseReceive	Sent to the block when it receives updated data from an advise conversation (see the functions for “Interprocess Communication (IPC)” on page 229).
OLEAutomation	Sent to a block when the BlockMsg automation method is invoked. See “BlockMsg” on page 123.

Variables, Messages, & Functions

ModL Functions

A detailed description of the ModL 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 includes a complete list of the ModL functions. These functions can be called in the blocks that use equations (Equation, Equation(I), Queue Equation, and Optimizer) and when creating new blocks.

ModL function overview

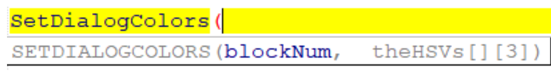
The rest of this chapter is a description of all the functions in ModL, listed by type. Within types, the functions are grouped by category. Within categories, the functions are listed alphabetically.

Function Type	Page	Categories
Math	207	Basic math, Trigonometry, Complex numbers, Statistical and random distributions, Financial, Integration, Matrices, Bit handling, Equations
I/O	222	File I/O (formatted), File I/O (unformatted), Internet Access, Interprocess Communication, OLE/COM, Mailslot, ODBC, Serial I/O, Other drivers, DLLs, Alerts and prompts, User inputs
Animation	250	2D Animation visible on the model worksheet.
Blocks and inter-block communication	256	Block numbers/labels/names/type/position, Block connectors and connection information, Variable connectors, Connector tool tips, Dialog items and dialog items from other blocks, Block data tables, Dynamic linking, Dynamic text items, Dialog item tool tips, Block dialogs (opening and closing), Messages to blocks (sending and receiving), Icon views. Also see Scripting, below
Models, notebooks, and libraries	293	Models, notebook, and library information, simulation parameters, DE Modeling Using Equation Blocks.
Scripting	300	Building and running a model remotely. Also see “Blocks and inter-block communication” and “Models, notebooks, and libraries” above
Reporting	308	Block reporting
Plotting/Charts	308	Functions for displaying graphs and data
Arrays, Queues, Delays, Linked lists, and String lookup tables	340	Dynamic arrays, Passing arrays, Global arrays, Queues, Delay lines, Linked list data structures, and String lookup tables
Database	318	Manipulating databases and database data with the ExtendSim database
Miscellaneous	358	Strings, Attributes, Calendar Date and time, Time units, Colors, EColors, Timer functions, Debugging, ADO, Help, Platforms and versions, and Application privileges.

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

Code completion

When you start to type a function or message handler name in the structure window or include file of a block, it will come up with a list of possibilities. Click the one you want.



Once the function has been placed in the script, type an open parenthesis "(" immediately following it. This causes the parenthesis to turn red and causes *call tips* to display the function's arguments as shown here. The first argument will be bolded. When you enter it, the parenthesis will turn black. As you enter each argument, subsequent arguments get bolded until all are entered.

Overriding

Functions can be overridden by being re-declared any number of times below the first declaration. This is useful in that include files can have basic forms of functions which can be re-declared and overridden in the main block code.

Type conversion of arguments

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

Static data limits

Each function has a limit of 32,560 bytes of data for locally defined static data. Dynamic arrays, global arrays, and database tables are not part of the count and are the more common and usually more useful method used to allocate large data structures.

Function returns

Except for void functions, which do not return values, all functions return values that are real, integer, or string. The type of value returned is indicated in the third column of the function tables as:

Return	Meaning
R	Real or Double (8 byte double)
I	Integer or Long (4 byte long integer)
S	String (up to 255 characters)
V	Void function (no value returned)

Math functions

Basic math

These functions perform numerical operations on their arguments.

Basic Math	Description	Return
Ceil(real x)	Nearest integral real with a value greater than or equal to *x*. For example: Ceil(1.3) returns 2.0, and Ceil(-1.3) returns -1.0	R
Erf(real x)	Returns the erf value of the variable x. Erf is the Error function, which is a special case of the incomplete gamma function. See Numerical recipes in C.	R
Exp(real x)	e^x	R
FFT(real array[n][2], inverse)	Replaces the array with the real and imaginary parts of the fast Fourier transform (see below).	V
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 or equal to *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
Log2(real x)	Returns the log base 2 value of the variable x.	R
Max2(real x, real y)	Maximum of the two arguments.	R
Min2(real x, real y)	Minimum of the two arguments.	R
NearlyEqual(real x, real y, integer precision)	Returns true if the x and y arguments are close enough to equal each other. The precision argument specifies the number of significant figures to compare the two numbers.	I
NearlyGreaterThan(real x, real y, integer precision, integer equal)	Returns true if x is greater than y. If *equal* is TRUE, NearlyEqual() is used, with the *precision* argument, to give an equal or greater result. If *equal* is FALSE, x has to be greater than y by the *precision* number of significant figures. See the NearlyEqual() function.	I
NearlyLessThan(real x, real y, integer precision, integer equal)	Returns true if x is less than y. If *equal* is TRUE, NearlyEqual() is used, with the *precision* argument, to give an equal or less than result. If *equal* is FALSE, x has to be less than y by the *precision* number of significant figures. See the NearlyEqual() function.	I
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	Return
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	Return
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	Return
AddC(result, U, Z)	result = *U*+*Z*	V
DivC(result, U, Z)	result = *U*/*Z*	V
MultC(result, U, Z)	result = *U***Z*	V
SubC(result, U, Z)	result = *U*-*Z*	V

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	Return
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
Mean(real array[], integer i)	Arithmetic mean of the first *i* members of the single-dimensional array.	R

Statistics/ distributions	Description	Return
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 Hyperexponential 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 ExtremeValue1A20 ExtremeValue1B 21 JohnsonSB 22 JohnsonSU23 Laplace24 Rayleigh25 InverseWeibull26 Logarithmic27 Hypergeometric28 Chisquared29 PowerFunction30 Cauchy31 Logistic32 InverseGaussian33 Pareto34	R

Statistics/ distributions	Description	Return
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 -8 Arg1 is negative or less than 1.0e-15 -9 Arg2 is negative or less than 1.0e-15 -10 Arg3 is negative or less than 1.0e-15 -11 Arg1 >= Arg2 -12 Arg1 > 1.0 -13 Arg2 > 1.0 -14 Arg3 > 1.0	I
RandomGetModel-SeedUsed()	The actual seed used for the model. If the Simulation 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 RANDOM-SEED global variable, which just shows the actual number entered in the Simulation 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.	V
SeedListClear()	Clears the list of the seed values.	V
SeedListRegister(real blockNumber, integer seed)	Keeps a list of the seed values. Returns a BLANK for success, meaning the item was entered in the list, or returns the blockNumber of the block that already posted the seed value. Note: Block number is a real so we can register DB cells that generate random numbers using a DB attribute. DB attributes will appear as negative numbers.	R
StdDevPop(real array[], integer i)	Population standard deviation of the first *i* members of the single-dimensional array.	R

Statistics/ distributions	Description	Return
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
UseRandomizedSeed()	Forces a randomized seed for the current simulation run, overriding any fixed seed entered into the Simulation Setup dialog. Must be called in CHECKDATA message handler.	V

See “Random numbers” in the main ExtendSim User Reference for information about how ExtendSim 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	Return
CalcFV(ratePer, nPer, pmt, pv, payAtBegin)	Future value	R
CalcNPER(ratePer, pmt, pv, fv, payAtBegin)	Number of periods	R

Financials	Description	Return
CalcPMT(ratePer, nPer, pv, fv, payAtBegin)	Payment	R
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	Return
IntegrateEuler(real array[4], real inputValue, real deltaTime)	Value of a 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.	V
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. Matrices are supported up to 1500x1500.

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, ma-
trixR
real vector[rows];                // real vector; also values
real matrixC[rows][columns][2];   // complex;
                                  //   also matrixAC, matrixBC, ma-
trixRC
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	Return
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.	V
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*.	V
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.	V
IdentityC(matrixRC, integer m)	Complex version of Identity.	V
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*.	V

Matrices	Description	Return
LUde-comp(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
LUde-compC(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.	V
MatAddC(matrixRC, matrixAC, matrixBC, integer m, integer n)	Complex version of MatAdd.	V
MatCopy(matrixR, matrixA, integer m, integer n)	Copies *matrixA* (of dimension m by n) into *matrixR*.	V
Mat-CopyC(matrixRC, matrixAC, integer m, integer n)	Complex version of MatCopy.	V
MatIn-vert(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
MatIn-vertC(matrixRC, matrixAC, integer m)	Complex version of MatInvert. If the matrix is singular, the function returns TRUE.	I
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*.	V
MatMat-ProdC(matrixRC, matrixAC, integer mA, integer nA, matrixBC, integer mB, integer nB)	Complex version of MatMatProd.	V

Matrices	Description	Return
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).	V
MatScalar-ProdC(matrixRC, matrixAC, integer m, integer n, BC)	Complex version of MatScalarProd.	V
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.	V
Mat-SubC(matrixRC, matrixAC, matrixBC, integer m, integer n)	Complex version of MatSub.	V
MatVectorProd(vectorR, matrixA, integer m, integer n, vectorB)	Creates an *m* length vector (*vectorR*) by multiplying *matrixA* (*m* by *n*) by *vectorB* (*n*).	V
MatVector-ProdC(vectorRC, matrixAC, integer m, integer n, vec-torBC)	Complex version of MatVectorProd.	V
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*.	V
OuterC(matrixRC, vectorAC, integer m, vectorBC, integer n)	Complex version of Outer.	V
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

Matrices	Description	Return
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*.	V
TransposeC(matrixRC, matrixAC, integer m, integer n)	Complex version of Transpose.	V

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	Return
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
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 the user can enter equations built on ModL code and the ExtendSim built-in functions. (Note that user-defined functions must be defined in include files in order to be used in an equation block.) See "Dynamic text items" on page 283 if you want to implement much larger (text edit boxes are initially limited to 255 characters) user-entered equations (up to 32000 characters). See the Equation block (Value library) for an example of using these and the dynamic text item functions in building your own equation block.

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

☞ 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 ExtendSim 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 (Value library).

Equations	Description	Return
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. Also see "Dynamic text items" on page 283 for larger equations than 255 characters. If you need unlimited input and output variables, see EquationCalculateDynamicVariables, below.	R
EquationCalculate20 (real input1, real input2, ... real input19, real input20, integer dynArrayName[])	This works the same as the EquationCalculate function, except it allows 20 inputs. Also see "Dynamic text items" on page 283 for equations larger than 255 characters. If you need unlimited input and output variables, see EquationCalculateDynamicVariables, below.	R
EquationCalculateDynamic(real arrayValues[], integer equationArray[])	Calculates a dynamic text equation using the input arguments from the array arrayValues. Up to 20 input arguments are allowed.	I
EquationCalculateDynamicVariables(real inputDynArray, real outputDynArray, integer codeDynArray)	This function allows unlimited input and output variables. InputDynArray values and outputDynArray real array are used in the equation like this example: outputsName[i] = inputsName[i]; Make sure that you have enough elements allocated in the input and output dynamic arrays. See the Equation block in the Values library to see how to use this function.	I

Equations	Description	Return
EquationCompile (string inputVar-Name1, ..., string inputVarName10, string output VarName, string equation, integer tabOrder, integer dynArrayName[])	Compiles an equation that is entered 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. 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 dia-log 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. Also see "Dynamic text items" on page 283 for larger equations than 255 characters. If you need unlimited input and output variables, see EquationCompileDy-namicVariables, below.	I
EquationCom-pile20 (string inputVarName1, ..., string inputVar-Name20, 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. Also see "Dynamic text items" on page 283 for larger equations than 255 characters. If you need unlimited input and output variables, see Equation-CompileDynamicVariables, below.	I
EquationCompile-Dynamic (str31 varNames[], string dynamicTextAr-ray, integer out-putEquation[], string labelOutput, integer tabOrder)	Compiles a dynamic text equation. See EquationCompile20(), as this is similar except that the input variable names are supplied in the array VarNames. VarNames can be up to 20 arguments.	I
EquationCompile-DynamicVariables (string inputsName, string out-putsName, string equationDynArray, integer codeDy-nArray, integer tabOrder)	This function allows unlimited input and output variables. InputsName and outputsName are used in the equation like this example: outputsName[i] = inputsName[i]; where i will be the index of that input or output variable. TabOrder is used to select the correct text item when there is an error in the equation. You can do a string substitution in the user's raw equation to put it in this indexed form. See the Equation block in the Values library to see how to use this function. See also EquationCompileDynamic-VariablesSilent.	I

Equations	Description	Return
EquationCompile-DynamicVariables-Silent (string inputsName, string outputsName, string equationDy-nArray, integer codeDynArray, integer tabOrder)	Similar to the EquationCompileDynamicVariables function, except this function doesn't show the lines compiled progress dia-log	I
EquationCompile-SetStaticArray (integer dynArray)	See the Equation block for use. The dynArray argument can be any type as the equation compile functions set up the array to hold any declared static variables.	V
EquationDebug-Calculate(integer debugEquationIn-dex, real inputVar-ValuesArray[], real outputVarValue-sArray[])	Calculates the equation using specified input values. The output values for the result will be put into *outputVarValuesArray*. Returns a TRUE value if an error occurs.	I
EquationDebug-Compile(integer debugEquationIn-dex, string equa-tionCodeArray[], string varsInputAr-ray[], string var-sOutputArray[], integer tabOrder)	Converts the equation code and variable names into block form so it can be debugged. *DebugEquationIndex* is the previously returned value from a previous call to this function for this equa-tion, or -1 (minus one) when this function is called for the first time with this equation. Returns a debug equation index to be used in the other equation debugging functions. *TabOrder* is the tab order of the text item with the equation. See the equation-based blocks for how to use this function. NOTE: *DebugEquationIndex should be set to -1 if EquationDe-bugDispose() (below) is called for that index OR this function is called for the first time for this block.*	I
EquationDebug-Dispose(integer debugEquationIn-dex)	Disposes and releases the memory used by the hidden block spec-ified by *debugEquationIndex* used to debug a particular equation. This does not affect the visible equation block. NOTE: The variable used for *DebugEquationIndex should be set to* *-1 (minus one) after this function is called so it works correctly if Eq*uationDebugCompile(), above, is called after this call.	V
EquationDebugSet-Breakpoints(inte-ger debugEquationIn-dex)	Opens a “Set Breakpoints” window so the user can click to create debugger breakpoints. Returns a True value if an error occurs.	I

Equations	Description	Return
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 (Value library). A shorter name for this function is EqGet(). 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
EquationInclude-Set(string theIn-cludeName)	Called right before one of the equationCompile functions, this puts the contents of the specified include file into the compiled equation. Call this for each include desired.	V
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. A shorter name for this function is EqSet(). 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 EquationGet-Static(index), above, to use that value in their equation.	V
IncludeFileEdi-tor(string include-FileName, integer blockNumber)	Tags the specified include file to do the following: If you click the close box, it will not close, but instead will send a message to the block specified by blockNumber. Equation-based blocks use this for their external code editor functionality; see the equation blocks for examples. Returns 0 for success or a negative value to indicate failure.	I
ShowFunction-Help(integer alpha)	Brings up ExtendSim's Help with a list of the functions and argu-ments 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.	V

I/O functions

File I/O, formatted

Use these functions to manipulate formatted text files. 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 (""), ExtendSim 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 "driveLetter:\direc-tory\directory\fileName" with each level separated by backslashes. If the volumeName and folder names are left off of the pathname, the file name will come from the current folder. Note that names of files can only be 31 characters long.

The Import and Export functions let you define the column delimiter (separator) character in the file to be read or written. In ExtendSim, Excel, Word, and most other tabular data applications, tabs normally delimit columns, and CRLFs (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	Return
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

File I/O (formatted)	Description	Return
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	Return
Create-Folder(string path-Name)	Creates a new folder from the *pathName*. The *pathName* must use backslashes (\) to separate folder names, "myDrive:\Extend-SimX\myNewFolder" where "X" is the ExtendSim version. Returns FALSE if successful, TRUE if there was an error.	I
DirPathFromPath-Name(string path-Name)	Returns the folder pathname part of a complete pathname. Also see FileNameFromPathName(), below. For example: "C:\myfolder\"	S
FileChoose(string defaultFilename, string Prompt)	Pops up the standard file selection dialog, with the prompt and the default file name specified, and returns the file name of the file the user selects. This differs from the fileOpen function, which it otherwise resembles, in that it just returns the file name/path name of the selected file without opening it. This allows the developer to use that name however she chooses.	S
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.	V
FileDelete(string pathname)	Deletes the file. Use this with caution because deleted files are not recoverable.	V
FileEnd-OfFile(integer file-Number)	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

File I/O (unformatted)	Description	Return
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 CRLF (carriage line feed) row delimiter was found. Call this immediately after FileRead to find out whether a column delimiter or CRLF followed the data.	I
FileGetPathName (integer fileNumber)	Returns the file's path name.	S
FileInfo(string filePathName, integer which)	Returns information about the file specified in the filePathName argument. The *which* argument specifies what information will be returned: 1: created date 2: modified date Dates are returned as ExtendSim date values.	R
FileIsOpen(string pathName)	Returns TRUE if the file described by the *pathName* is open.	I
FileNameFromPathName (string pathName)	Returns the filename part of a complete pathname.Also see DirPathFromPathName(), above, to get the path name part of a complete path name.	S
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 (""), ExtendSim 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 FileOpen, the new data is appended to the end of the file. Call FileOpen in the InitSim message handler to open files at the beginning of a simulation. NOTES: 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. Note that if *pathname* is an empty string (""), the user will be prompted for a filename at run time.	I

Functions

File I/O (unformatted)	Description	Return
FileRead(integer fileNumber, string colDelim)	Reads and returns a string read from the file, up to *colDelim* (a column delimiter character) or to a CRLF (carriage line feed) 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.	V
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 CRLF (carriage line feed) delimiter is written after s. If the column delimiter is a plus sign ("+"), no delimiter is written between the strings.	V
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 folder. The first array will contain the names of all the files/directories, and the second will contain an integer value that will be zero for a file, and one for a folder. It returns the number of row entries in the array. The pathname separator on a mac is a ":", on windows a "/".	I
GetFileReadMachineType()	Returns the type of machine the currently active model was saved on. (This is only useful if models have been moved from one platform to another. Otherwise, it will be the same as the machine the model is running on.) Type 2 is Windows, 1 is models built on 68k Macs, and 4 is models built on PPC Macs.	I
StripLFs (integer strip)	Sets a flag in ExtendSim that determines if the fileread functions will strip LF characters. This flag defaults to TRUE, so you should call StripLFs(FALSE) if you find that meaningful LF characters are missing from your data.	V
StripPathIfLocal (string pathName)	Strips off the pathname if the file is in the same folder 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

Internet access

These functions allow data and file access and manipulation using Internet protocols. The *FTP* block (Libraries/Example Libraries/ModL Tips) is an example of the use of these functions in FTP access.

The connection type handles used in these functions are:

SessionHandle	The handle to this entire Internet communication session.
FTPHandle	The handle to a specific FTP session. **You must use this handle to access and manipulate files on a Server.**

SearchHandle	The handle to a specific search operation.
InetHandle	Any one of the above handle types.

Internet access	Description	Return
INetCloseHandle(Integer inetHandle)	Closes an InetHandle. This function will close a handle created by InetOpenSession(), InetFTPFileFirstFile(), or InetConnect().	V
INetConnect(integer sessionHandle, string serverName, string userName, string passWord, integer connectionType)	Creates an Internet connection type. The *connectionType* integer can be 1:FTP (**The only connectionType that allows you to access or manipulate files.**) 2:HTTP (Allows you to open a connection to a website location. Cannot be used for accessing or manipulating of files.) 3:Secure HTTP Returns a connection handle that needs to be closed with INetClosehandle when the connection is complete.	I
INetFileImportText(integer hFile, string format, stringArray array)	Given an INet handle (most likely created by INetOpenURL) this function will import data from the file represented by that handle into the specified array.	I
INetFindNextFile(integer searchHandle)	Continues an Internet file search by moving the searchHandle on to the next file. Returns a searchHandle.	I
INetFTPCreateDirectory(integer FTPHandle, string targetName)	Creates a folder named targetName in the current folder.	I
INetFTPDeleteFile(integer FTPHandle, string targetName)	Deletes the path\file targetName if it is found.	I
INetFTPExport(integer FTPHandle, string fileName, string delim, real array[][], integer rows, integer cols)	Operates much like the Export function (see page 222) with the exception that it writes to a path\file accessed via FTP.	I
INetFTPExportGA(integer FTPHandle, string fileName, string delim, integer GAIndex, integer rows, integer cols)	Writes out the contents of a Global Array to the FTP path\file.	I

Internet access	Description	Return
INetFTPExportText(integer FTPHandle, string fileName, string delim, string array[][], integer rows, integer cols)	Operates much like the ExportText function (see page 222) with the exception that it writes to a path\file accessed via FTP.	I
INetFTPFindFirstFile(integer FTPHandle, string searchFile, integerflags)	Starts a search for files in the default folder. An empty string will find the first file in the current folder with any filename. Returns a search handle that needs to be closed with INetClosehandle when the search is complete. Currently, flags should be set to zero.	I
INetFTPGetCurrentDirectory(integer FTPHandle)	Returns the path to the current FTP directory as a string.	S
INetFTPGetFile(integer FTPHandle, string targetName, string fileName, integer failIfExists, integer fileAttributes, integer flags)	Copies a file from the remote site to the local machine. The path\targetname specifies the name of the file to be retrieved. The path\filename specifies the local name where the file should be put. FailIfExists determines what happens if the local file already exists. Currently, flags and file attributes should be set to zero.	I
INetFTPImport(integer FTPHandle, string fileName, string delim, real array[][])	This operates much like the Import function (see page 222) with the exception that it reads a path\file accessed via FTP.	I
INetFTPImportGA(integer FTPHandle, string targetName, integer format, integer GAIndex)	Reads the contents of an FTP path\file into the Global Array.	I
INetFTPImportText(integer FTPHandle, string fileName, string delim, string array[][])	This operates much like the ImportText function (see page 222) with the exception that it reads a path\file accessed via FTP.	I
INetFTPPutFile(integer FTPHandle, string fileName, string targetName, integer flags)	Copies a local file to the Internet. Path\fileName specifies the name of the local file. Path\targetName specifies the desired name of the file on the Internet. Currently, flags should be set to zero.	I

Internet access	Description	Return
INetFTPRemove-Directory(integer FTPHandle, string targetName)	Deletes the specified directory path\targetName from the site.	I
INetFTPRename-File(integer FTPHandle, string targetName, string newName)	Renames the file path\targetName to newName.	I
INetFTPSetCur-rentDirectory(inte-ger FTPHandle, string targetName)	Sets the current directory to path\targetName.	I
INetGetFindFile-Info(which)	Returns info about the current file in the FTP search. The only allowed input is connectionType *which*=1 (FTP); it returns TRUE if a directory, FALSE if a file.	I
INetGetFindFile-Name()	Returns the name of the current file in the FTP search.	S
INetOpenSession()	Starts an Internet session. Returns a session handle that needs to be closed with INetClosehandle when the session is complete.	
INetOpenURL (integer session, string url)	Given a session handle (created by the INetOpenSession function) and a URL, this function will open a connection to the site corre-sponding to that URL	I

Interprocess Communication (IPC)

Interprocess communication (IPC) provides a standard way in which one application can directly communicate with another. These functions allow ExtendSim to act as a client application by connecting to and requesting data and services from a server application. The server application must support dynamic data exchange (DDE)). The IPC functions start with “IPC”.

DDE is no longer supported as of release 10 of ExtendSim. Use OLE/COM instead. The functions that are no longer available in ExtendSim10 are noted below.

IPC & Publish/ Subscribe	Description	Return
IPCAdvise(integer conversation, string item, integer block-Number, string dia-logItem, integer rowStart, integer colStart, integer rowEnd, integer colEnd)	OBSOLETE AS OF ES10 (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 IPCS-topAdvise Function when the advise loop is to be terminated.	I

Functions

IPC & Publish/ Subscribe	Description	Return
IPCCheckConversation(integer conversation)	OBSOLETE AS OF ES10 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
IPCConnect(string serverName, string topic)	OBSOLETE AS OF ES10 Initiates an IPC conversation between ExtendSim and a server. The *serverName* 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 Windows expects “Excel”, on the Mac OS 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)	OBSOLETE AS OF ES10 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 disconnection was successful.	I
IPCExecute(integer conversation, string executeData, string item)	OBSOLETE AS OF ES10 Sends a command to be executed by the server in the specified *conversation*. 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 connection, 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	Return
IPCOpenFile (string fileName)	Opens the file (for example, a spreadsheet) in the Finder using DDE (Windows) or AppleEvents (Mac OS). 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 ExtendSim'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)	OBSOLETE AS OF ES10 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(integer conversation, string item, string delim, string array data)	OBSOLETE AS OF ES10 (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)	OBSOLETE AS OF ES10 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 (integer conversation, string item, string delim, string array data)	OBSOLETE AS OF ES10 (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 returned.	I
IPCSendCalcReceive(integer product, real sendValue, integer sendRow, integer sendCol, string funcName, integer receiveRow, integer receiveCol)	OBSOLETE AS OF ES10 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

IPC & Publish/ Subscribe	Description	Return
IPCSetTimeOut(integer timeOut)	OBSOLETE AS OF ES10 Sets the Timeout value for the various IPC functions. This determines how long ExtendSim 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.	V
IPCServerAsync (integer async)	If *async* is TRUE, sets a flag in ExtendSim so that ExtendSim will return from further Execute messages immediately instead of waiting for the Execute messages to complete. Useful when used in an Execute message from another application so that the other application can continue to do other tasks while ExtendSim calculates.	V
IPCSpreadSheet-Name(string spreadSheetName)	OBSOLETE AS OF ES10 Used to establish a default *spreadSheetName*, and is only used in conjunction with the IPCSendCalcReceive function. The IPCSendCalcReceive function does not take a spreadsheet name as an argument, and Lotus 1-2-3 requires a specific spreadsheet name for DDE communication on Windows.	V
IPCStopAdvise (integer conversation, integer adviseLoopID, integer block)	OBSOLETE AS OF ES10 (Windows only) Stops the specified advise loop within the specified block.	I
ServerOpen-Port(integer port)	Initializes ExtendSim to listen on that port. If the port is the same as previously set, the function just returns. If the port is 0 or negative, ExtendSim stops listening on the previously set port. Returns zero in all cases.	I
UpdatePublishers()	OBSOLETE AS OF ES10 (Mac OS only) 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.	V

OLE/COM (Windows only)

Embedded objects are no longer supported as of ExtendSim release 10. The functions that supported embedding in prior releases are noted below as being "obsolete".

The ModL functions below allow you to access, communicate with, and control other applications via OLE Automation.

OLE functions support SafeArray functionality where appropriate.

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.

☞ Examples of using OLE in code include the Object Mapper block (Utilities library), the Data Import Export block (Value library), and the Read and Write blocks in the Value and Item libraries.

OLE (Windows only)	Description	Return
OLEGetHelpContext (integer blockNumber, string dialogItem, integer dispID)	OBSOLETE AS OF ES10. Use OLEDispatchGetHelpContext. Returns the Help context value of the specified dispID.	I
OLEActivate(integer blockNumber, string dialogItem)	OBSOLETE AS OF ES10 Activates the specified embedded object. *DialogItem* is the dialog variable name in quotes. Returns FALSE if successful.	I
OLEAddRef(integer interfacePtr)	OBSOLETE AS OF ES10 Addrefs the interface specified. Returns the refcount.	I
OLEArrayParam (array data, integer variants)	Puts the data in the array 'data' into a safeArray, packaged as a parameter for an Invoke call. The variants argument determines whether each data element will be packaged in a separate variant or not.	I
OLEArrayParamVariableColumns(short hNum, array datatableName, integer numCols, integer variants)	Specifies that the datatable array passed in will be used as an argument for the next OLEInvoke call. The numCols argument will define how many columns the function defines the data as containing.	I
OLEArrayResult (array data)	Retrieves the SafeArrray result data from an Invoke call, putting the data into the array 'data'.	I
OLEArrayResultVariableColumns(array datatableName, integer numCols)	Specifies that the datatable array passed in will be filled with the result of the last OLEInvoke call. The numCols argument will define how many columns the function defines the data as containing.	I
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

OLE (Windows only)	Description	Return
OLEDB-Param(integer DBIndex, integer tableIndex, integer variants)	Passes the contents of the specified DB table as a safe array parameter.	I
OLEDBResult(integer DBIndex, integer tableIndex)	Fills the specified database table with the results of an Invoke or ParameterGet call that was just made.	I
OLEDeactivate (integer blockNumber, string dialogItem)	OBSOLETE AS OF ES10 Deactivates the specified embedded object. *DialogItem* is the dialog variable name in quotes. Returns FALSE if successful.	I
OLEDispatchGet-CLSID(integer dispHandle, integer progID)	Returns the CLSID as a string. ProgID is FALSE to return the CLSID, or TRUE to return the program ID.	S
OLEDispatchGet-DispatchName (integer dispHandle, integer which)	Same as the function OLEGetDispatchName below (which is obsolete as of ExtendSim 10), except it takes a dispatchHandle instead of a blocknumber and a dialog item name.	S
OLEDispatchGet-DispID(integer dispHandle, string theName)	Given a function/variable name, returns the DispID. Same as the function OLEGetDispID below (which is obsolete as of ExtendSim 10), except it is expecting a dispatchHandle instead of a block number and dialog item name.	I
OLEDispatchGet-Doc (integer IDispatchHandle, string returnDoc[], integer dispID, integer which)	This function is a Dispatch handle version of the OLEGetDoc() function (which is obsolete as of ExtendSim 10). For a description of the IDispatchHandle argument, see the OLEDispatch-GetHelpContext() function.	I
OLEDispatchGet-FuncIndex(integer dispHandle, integer dispID)	Same as the function OLEGetFuncIndex (which is obsolete as of ExtendSim 10), except it takes a dispatchHandle instead of a blocknumber and a dialog item name.	I
OLEDispatchGet-FuncInfo(integer dispHandle, integer funcIndex, integer which)	Same as the function OLEGetFuncInfo (which is obsolete as of ExtendSim 10), except it takes a dispatchHandle instead of a blocknumber and a dialog item name.	I

OLE (Windows only)	Description	Return
OLEDispatch-GetHelpContext (integer IDispatch-Handle, integer 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 (obsolete as of ExtendSim 10), or in the block dialog, or has been created via OLE Automation in an remote application. See OLEDispatchResult() and OLECreateObject().	I
OLEDispatchGet-Names (integer IDispatchHandle, Str31 names[], integer dispID)	See OLEDispatchGetHelpContext(). Same as OLEGetNames(), which is obsolete as of ExtendSim 10, except uses a handle.	I
OLEDispatchIn-voke (integer IdispatchHandle, integer dispID)	See OLEDispatchGetHelpContext(). Same as OLEInvoke(), which is obsolete as of ExtendSim 10, except uses a handle.	I
OLEDispatch-Param (integer dispatchHandle)	OBSOLETE AS OF ES10 Adds a DispatchHandle to the argument list for the next Invoke call. Note: arguments are listed in back to front order. Returns FALSE if successful.	I
OLEDispatchProp-ertyGet (integer IdispatchHandle, integer dispID)	See OLEDispatchGetHelpContext(). Use this function instead of OLEPropertyGet(), which is obsolete as of ExtendSim 10.	I
OLEDispatchProp-ertyPut (integer IdispatchHandle, integer dispID)	See OLEDispatchGetHelpContext(). Use this function instead of OLEPropertyPut(), which is obsolete as of ExtendSim 10.	I
OLEDispatchRe-sult()	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 (obsolete as of ExtendSim 10), or in the block dialog.	I
OLEGAParam (integer arrayIndex, integer variants)	Copies the data in the global array defined by arrayindex into a SafeArray, packaged as a paramaeter for an Invoke call. The variants argument determines whether each data element will be packaged in a separate variant or not.	I
OLEGAResult (arrayIndex)	Retrieves the SafeArrray result data from an Invoke call, putting the data into the global array referred to be arrayIndex.	
OLEGetCLSID (integer blockNum-ber, string dialog-Item, integer progID)	OBSOLETE AS OF ES10. Use OLEDispatchGetCLSID. Returns the CLSID as a string. ProgID is false to return the CLSID, or TRUE to return the program ID.	

Functions

OLE (Windows only)	Description	Return
OLEGetDispatchName (integer blockNumber, string dialogItem, integer dispID)	OBSOLETE AS OF ES10. Use OLEDispatchGetDispatchName *DialogItem* is the dialog variable name in quotes. Returns the name associated with the specified dispID.	S
OLEGetDispID (integer blockNumber, string dialogItem, string theName)	OBSOLETE AS OF ES10. Use OLEDispatchGetDispID *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)	OBSOLETE AS OF ES10. Use OLEDispatchGetDoc *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 than a single string. This function accesses text that is in the objects Type Library. What information is available, and whether any is available, 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
OLEGetFuncIndex (integer blockNumber, string dialogItem, integer dispID)	OBSOLETE AS OF ES10. Use OLEDispatchGetFuncIndex Returns the function Index used in OLEGetFuncInfo that corresponds to the dispID.	I

OLE (Windows only)	Description	Return
OLEGetFuncInfo (integer blockNumber, string dialogItem, integer index, integer which)	OBSOLETE AS OF ES10. Use OLEDispatchGetFuncInfo *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 : returns DispID (sometimes called memberID). 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()	OBSOLETE AS OF ES10 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 ExtendSim.	S
OLEGetInterface (integer blockNumber, string dialogItem, integer whichInterface)	OBSOLETE AS OF ES10 *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
OLEGetNames (integer blockNumber, string dialogItem, Str31 names[], integer dispID)	OBSOLETE AS OF ES10 *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
OLEGetRefCount (integer interfacePtr)	OBSOLETE AS OF ES10 Returns the RefCount on the Specified interfacePtr. Note that this does nothing more than an AddRef and a Release, so there is no reason to call this routine if you are already using addref and release.	

Functions

OLE (Windows only)	Description	Return
OLEInsertLicensedObject (integer blockNumber, string dialogItem, strng guid, integer xPixel, integer yPixel, string licString)	OBSOLETE AS OF ES10 Same as the OLEInsertObject function, below, with the exception of the last argument, which allows you to pass a license string to the object to be inserted. This is used for activeX objects that allow licensed execution as runtimes. See the function OLERequestLicKey for additional information.	I
OLEInsertObject (integer blockNumber, string dialogItem, string guid, integer xPos, integer yPos)	OBSOLETE AS OF ES10 *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
OLEInsertObjectFromFile(integer blockNumber, string dialogItem, string guid, integer xPixel, integer yPixel, string licString, string filePath)	OBSOLETE AS OF ES10 Inserts an Object into an embedded object dialog item, or onto the worksheet, like OLEInsertObject. It takes the additional Argument licString, the usage of which is defined in OLEInsertLicensedObject. It also takes a filepath Argument, which allows the coder to define an object based on a file, as is done in the standard Object insertion dialog.	I
OLEInvoke(integer blockNumber, string dialogItem, integer dispID)	OBSOLETE AS OF ES10. Instead see OLEDispatchGetHelpContext() *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 an 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
OLEObjectIsRegistered(string clsid)	Checks to see if a specific CLSID is already registered. Returns TRUE if the CLSID is already in the registry and FALSE if it is not.	

OLE (Windows only)	Description	Return
OLEProperty-Get(integer block-Number, string dialogItem, integer dispID)	OBSOLETE AS OF ES10. See instead OLEDispatchProperty-Get. DialogItem is the dialog variable name in quotes. Gets the prop-erty specified by dispID. You must call Result functions to retrieve the value.	I
OLEProperty-Put(integer block-Number, string dialogItem, integer dispID)	OBSOLETE AS OF ES10. See instead OLEDispatchPropertyPut. DialogItem is the dialog variable name in quotes. Sets the prop-erty 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
OLERealResult()	Returns a real value from the last Invoke call, if a return value is available.	R
OLERelease(inte-ger interfacePtr)	Releases the interface pointer specified. Returns the refCount.	I
OLEReleaseInter-face (integer inter-facePtr, integer whichInterface)	OBSOLETE AS OF ES10 Releases the interface pointer returned by OLEGetInterface. Returns FALSE if successful.	I
OLERemoveOb-ject(integer block-Number, string dialogItem)	OBSOLETE AS OF ES10 Removes the OLE object from the dialog item.	I
OLEReq-uestLicKey(string guid)	OBSOLETE AS OF ES10 This function requests a license key from a licensed activeX con-trol on your machine. This can be used with OLEInsertLicensed-Object, above, to allow the user to retrieve the license key to be used when inserting a licensed runtime activeX Control.	S
OLESetNamed-Param (integer par-amID)	Specifies that the first named parameter is the parameter Para-mID. If the Idispatch interface you are using specifies named parameters, then they are the first parameters by definition. All this function does is specify which named parameters you are using. The first time you call it, it specifies what the paramID of the first named parameter is, the second time the second, and so on.	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

OLE (Windows only)	Description	Return
OLESupressInvokeErrors(integer supressErrors)	Used to stop those pesky error messages from appearing during an invoke of an OLE object.	V
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(integer 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
WinRegSvr32(integer registerObject, string fileName, string dir)	Runs the RegSvr32 command line tool used to register .dll files as components in the windows registry. The registerObject integer is a flag that determines if you want to register (true) or unregister (false) the object. FileName is the name of the dll file. Dir is the path name to the directory containing the dll. NOTE: also works correctly for 64bit.	I

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.

☞ Because mailslot technology is a “polling” system, there will be a time delay between when a message is sent to when a message is actually received.

☞ 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.

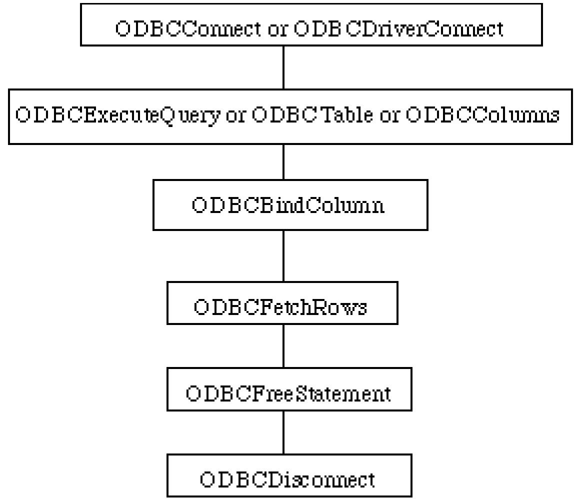
See the Windows API documentation for more information about mailslots.

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

Mailslot (Windows only)	Description	Return
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.



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.

☞ There is a sample block in the ModL Tips library) in the Input/Output category that shows the syntax of these commands. The block is called ODBC, and can be used as an 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 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
ODBCColumns2(integer hdbc, string catalog, string schema, string table, string column)	This function works the same as the ODBCColumns function except that it adds four string arguments. See your ODBC documentation (SQLColumns) for additional information about the meaning and use of these arguments. Please note that the arguments are strings, so you cannot pass in the NULL values that are defined in the SQLColumns specification. For this function, we've defined an empty string "" to be interpreted as a NULL, so just use empty strings for unused arguments where NULL is normally used.	I
ODBCConfigDataSource(integer fRequest, string szDriver, string szAttributes)	Calls the Windows API SQLConfigDataSource() function with the entered arguments	
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 ExtendSim, 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 database 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 important to disconnect this connection before quitting ExtendSim, 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
ODBCExecuteArray(integer hdbc, string array[])	See the description for ODBCExecuteQuery() below. This function allows a query that contains more than 255 characters by allowing you to pass in an array of strings instead of just one string.	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
ODBCKeyword(string word)	Returns a TRUE value if the string word is an ODBC reserved keyword. Otherwise returns a FALSE value.	I

ODBC	Description	Return
ODBCNumResult-Cols (integer statement)	Returns the number of resulting columns in the specified dataset. This function will only return useful information after an ODBCExecuteQuery, ODBCTables, or ODBCColumns call has established a dataset.	I
ODBCSetRows (integer connection, string tableName, 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. That is, the variable named in the IDName field will be compared to the values in the IDArray array, and each row of the columnName variable will be updated based on the database row selected by the values in the IDArray array. Returns FALSE (0) if unsuccessful.	I
ODBCSetRows-Type(linteger hdbc, string tableName, string colName, string IDName, string y[], string x[], integer varTypeX, integer varTypeY)	See the description for the ODBCSetRows function. Whereas the SetRows function defaults the types of the values of both string arrays to SQL_CHAR, the SetRowsType function allows you to specify the types of the variables in the y and x arrays. Type values are: SQL_UNKNOWN_TYPE 0, SQL_CHAR 1, SQL_NUMERIC 2, SQL_DECIMAL 3, SQL_INTEGER 4, SQL_SMALLINT 5, SQL_FLOAT 6, SQL_REAL 7, SQL_DOUBLE 8, SQL_DATETIME 9, SQL_VARCHAR 12, SQL_TYPE_DATE 91, SQL_-TYPE_TIME 92, SQL_TYPE_TIMESTAMP 93	I
ODBCSuccessInfo (integer ShowSuccessInfo)	Sets a flag that determines if warning error messages are shown, or not.	V
ODBCTables (integer 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 disconnecting 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 columns of the resulting dataset.	I

Serial I/O

These functions allow ExtendSim to interact with serial ports. For Windows the port arguments are from 1 through 4 for COM1 through COM4, respectively.

You can, for example, set up a simulation 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 ExtendSim simulation can be controlled by a modem from a remote location.

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	Return
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 Serial-Read 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.	V
SerialWrite(integer port, string s)	Writes the string *s* to the serial port buffer.	V

DLLs

Working with DLLs

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

- DLLs called by ExtendSim must be built for 64-bit execution.
- The DLLs are stored in the ExtendSim/Extensions/DLLs folder.
- Unless the variables are strings or arrays, variables passed from the ModL code to a DLL are passed by value:
    - Reals are 8-byte double precision
    - Integers are 4-byte long integers (converted to 8-byte for the DLL)
    - Pointertype (64 bit) variables are 8-byte integers
- Strings and arrays are passed to DLLs as 8-byte pointers to data that has been allocated by ExtendSim. Modifications to that data will affect the original information in ExtendSim.
    - Arrays that are passed to a DLL come through as pointers to the original data in ExtendSim. Accessing and modifying the data is fine, but you should not try to resize the pointer. If you do, ExtendSim will not be able to access the data and will probably crash..
    - Strings are passed to DLLs 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. To convert strings to C strings and back again, see the DLLCtoPString and DLLPtoCString functions.
- The most common problem associated with building and using a DLL is making sure that the names of the routines that you want to call are exported and that they are exported *without* Name Mangling or Name Decoration. Name Mangling is an option for how names are exported from a DLL; it adds information about the arguments to the exported name.

⚠ When building a DLL for use with ExtendSim, the Name Mangling option should be off.

☞ See "DLLs" on page 86 for some DLL coding examples.

About the functions

These function calls are called with variable argument lists. The first argument in each case is the Proc-Address for the procedure, which is a 4-byte integer (**not** a 64-bit Pointertype). 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 than call it each time you call the specific function.

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.

ProcAddress is not a 64-bit Pointertype variable. Instead, it is a 32 bit (4-byte) integer index that points to a pointer internal to ExtendSim. This makes the 32 bit block code used in previous releases compatible with the 64 bit code of ExtendSim 10.

All of the DLL calls below are translated into 64-bit fastcalls, which is the only call allowed in 64-bit systems.

DLL/Shared Libraries	Description	Return
DLLBoolCFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a Boolean (translated into an integer by ExtendSim). 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 an integer by ExtendSim). 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 an integer by ExtendSim). 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 V string that is usable by ModL, as ExtendSim/ModL internally can use only V (Pascal) strings. In some cases pre-established DLLs 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.	V
DLLDoubleCFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a real. Accepts a variable argument list. Assumes a C calling convention.	R
DLLDoublePascalFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a real. Accepts a variable argument list. Assumes a Pascal calling convention.	R
DLLDoubleStdcallFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns a real. This function assumes a StdCall calling convention.	R

DLL/Shared Libraries	Description	Return
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. See also DLLUnloadLibrary.	I
DLLLongCFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress. Returns a 64 bit integer that can be saved as a ModL integer (32 bit) or, if needed, as a pointertype (64 bit integer/pointer). Accepts a variable argument list. Assumes a C calling convention.	I
DLLLongPascalFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress. Returns a 64 bit integer that can be saved as a ModL integer (32 bit) or, if needed, as a pointertype (64 bit integer/pointer). Accepts a variable argument list. Assumes a Pascal calling convention.	I
DLLLongStdCallFunction (integer procAddress, ...)	Calls a DLL routine referenced by procAddress. Returns a 64 bit integer that can be saved as a ModL integer (32 bit) or, if needed, as a pointertype (64 bit integer/pointer). 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
DLLMakeProcInstanceLibrary (string libraryName, string procName)	Similar to DLLMakeProcInstance except it has a libraryName argument so you can specify which library should be opened and searched for the procedure.	I
DLLPtoCString (string theString)	Converts a V string to a C string that is usable by a DLL, as ExtendSim/ModL internally can use only V (Pascal) strings. In some cases pre-established DLLs 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 V strings.	V
DLLUnloadLibrary (string name)	Unloads the DLL from memory. See also DLLLoadLibrary.	I
DLLVoidCFunction(integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns nothing. Accepts a variable argument list. Assumes a Pascal calling convention.	V
DLLVoidPascalFunction (integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns nothing. Accepts a variable argument list. Assumes a Pascal calling convention.	V
DLLVoidStdcallFunction (integer procAddress, ...)	Calls a DLL routine referenced by procAddress and returns nothing. This function assumes a StdCall calling convention.	V

Alerts and prompts

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

Alerts & Prompts	Description	Return
Beep()	Causes the computer to beep using the beep sound selected in the Control Panel.	V
IntegerParameter (label, default)	Similar to NumericParameter, except it returns an integer value and displays the parameter to the user as an integer.	I
IntegerParameter2 (label1, default1, label2, default2, resultArray)	Similar to NumericParameter2, except it returns integer values and displays the parameters to the user as integers.	I
NumericParameter(string message, real default)	Similar to the userParameter function, except that this returns a real value. This function will return a NOVALUE if the user clicks on the cancel button. See also userParameter.	R
NumericParameter2 (string message1, real default, strng message2, real default2, real array[])	Similar to the numericParameter function, except that this function returns two values in a real array declared as real array[2]; The return value of the function itself will be a zero if the user successfully input one or more numbers, or a –1 if they canceled. See also numericParameter2 and userParameter.	R
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 89 for important details.	I
Speak(string s)	(Mac OS only) Speaks the string if the speech manager is present.	V
UserError(string s)	Opens a dialog with an OK button displaying the string *s*.	V
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, or the empty string (““) if Cancel was clicked. 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. See also numericParameter.	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
userPromptCustomButtons(string str, string button1, string button2)	User prompt with the ability to customize the text of the buttons. Returns a 1 (one) if the first button is clicked or 2 (two) if the second 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 367.

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	Return
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 (Mac OS) 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
GetMouseXActive-Window()	Returns the mouse X position in pixels relative to the active window, for example, a hierarchical submodel window. Use the GetBlockTypePosition() function to get the coordinates of a block in that window. 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
GetMouseYActive-Window()	Returns the mouse Y position in pixels relative to the active window, for example, a hierarchical submodel window. Use the GetBlockTypePosition() function to get the coordinates of a block in that window. Can be used in the on blockClick message handler.	I
HBlockClicked()	This function returns the block number of the Hblock that was last double clicked. This is intended to be called during a On HblockOpen message handler, and will always return a negative one at other times.	V
isKeyDown(integer keyCode)	Returns a True/False value for whether or not the specified key is pressed. The constants for *keyCode* are the constants for the API call GetKeyState (Windows) or the GetKeys functionality (Macintosh).	I

User inputs	Description	Return
Lastkeypressed()	The value returned will be the ASCII value of the character 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 monitoring what keys the user hits on the keyboard. Used in conjunction with startTimer (See "Timer functions" on page 364) 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
WhichDTCell-Clicked(integer rowCol)	Returns the row or column of the cell in the data table that was clicked on. This function should only be used in the on dialog-Click message handler, and usually after the whichDialogItemC-licked function has determined that a specific data table was clicked on. *rowCol* 0 = row *rowCol* 1 = col	I

Animation

2D Animation

Use these functions to implement the ExtendSim 2D animation. Examples of using these functions are shown in "2D animation" on page 128.

The Show 2D Animation command (Run menu) only controls whether animation is shown *during the simulation run*. At all other times, the block will still show animation if the block creator has coded it to do that. For instance, animation is available when the modeler makes any changes in a block's dialog, whether Show 2D Animation is selected or not and regardless of whether the simulation is running. When the command Show 2D Animation 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 131, 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 ExtendSim doesn't find that object in the current enclosing H-block, ExtendSim will search the H-block enclosing that one and so on, until it finds the object or reaches the top level.
- The outsideIcon value is a value 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.
- Some functions have the option of selecting a pattern in addition to a color. Starting with ExtendSim 10, patterns remain as arguments for those functions but are no longer supported.

- AnimationEColor specifies the color of animated objects. See the “Select Color window” on page 366.

There is a limit of 256 animation objects that can be *manually* placed on a block’s icon when the block is created. However, the AnimationObjectCreate function supports thousands of animation objects. This function can be used to create new animation objects during the run; use AnimationObjectDelete() to delete the objects during the run.

Animation	Description	Return
AnimationAntialias (integer objNum, integer antialias)	Determines whether or not the specified animation object uses anti-aliasing when it draws. Defaults to on (True) for most animation object types. Call this function with a false (0) if you don't want the specified animation object to be drawn with anti-aliasing.	V
AnimationBlockToBlock (integer animationObjNum, integer blockNumFrom, integer conNumFrom, integer blockNumTo, 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.	V
AnimationBorder (integer obj, integer pixels)	Changes the border size of an animation object. Pixels can be 0, for no border, and up to 20 for a 20 pixel border.	V
AnimationBorderColor (integer objNum, integer hue, integer saturation, integer value)	Allows you to specify the color of the border on an animation object. Only effects objects that have a visible border around them. As of ExtendSim 10, see instead AnimationBorderEColor	V
AnimationBorderEColor (integer objectNum, integer EColorValue	Allows you to specify the EColor of the border on an animation object. Only effects objects that have a visible border around them. See “EColors” on page 365 for more information.	V
AnimationColor (integer obj, integer hue, integer sat, integer bright, integer pattern)	Sets the color of the object using the pattern number, hue, saturation, and brightness. Patterns are not supported as of release 10, so use 1 for a solid pattern. Also, for ExtendSim 10 or later, use the AnimationEColor function instead.	V
AnimationEColor (long objNum, long EColorValue)	Set the color of an animation object using an EColor value. See “EColors” on page 365 for more information.	V

Animation	Description	Return
AnimationGetHeight (integer obj, integer originalValue)	Returns the offset of the height of the object. If *originalValue* is true, gets the original height. See also AnimationGetHeightR, below.	I
AnimationGetHeightR (integer obj, integer getOrig)	The same as the animationGetHeight function above, except it returns a real number for the height instead of an integer.	R
AnimationGetLeft (integer obj)	Returns the offset of the left side of the object relative to its original position in the icon.	I
AnimationGetLeftRelative (integer blockNumber, integer objNum, integer original)	Gets the left of the animation object relative to the upper left hand corner of the block icon. See AnimationGetTopRelative and AnimationGetTopRelativeR for more info.	I
AnimationGetLeftRelativeR (integer blockNumber, integer objNum, integer original)	Exactly the same as AnimationGetLeftRelative except this function returns a real for the left number.	R
AnimationGetSpeed()	Returns the speed setting, with 1 being slowest and 5 being fastest. See AnimationSetSpeed(), below.	I
AnimationGetTop (integer obj)	Returns the offset of the top side of the object relative to its original position in the icon.	I
AnimationGetTopRelative(integer blockNumber, integer objNum, integer original)	Gets the top of the animation object relative to the upper left hand corner of the block icon. Compare to AnimationGetTop, which returns the location relative to just the Animation Object location, and will always return zero unless the animation object has been moved with an AnimationMove call. Original specifies if the moved rect (0), or the original unmoved rect (1) should be used.	I
AnimationGetTopRelativeR(integer blockNumber, integer objNum, integer original)	Exactly the same as AnimationGetTopRelative except this function returns a real for the top.	R
AnimationGetWidth(integer obj, integer getOrig)	Returns the width of the object. If *getOrig* is true, gets the original width before any stretching. See also AnimationGetWidthR, below.	I
AnimationGetWidthR(integer obj, integer getOrig)	Same as AnimationGetWidth, above, except it returns a real number for the width instead of an integer.	R
AnimationHide (integer obj, integer outsideIcon)	Immediately hides the object. NOTE: as of ExtendSim 10, the outsideIcon argument is unused.	V

Animation	Description	Return
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).	V
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 animation object.	V
AnimationMovie integer obj, string movieName, real speed, integer playSoundTrack, integer async)	(Mac OS only) Defines the object as a QuickTime movie, played at the specified speed (relative to 1.0, the normal speed). If *playSoundTrack* 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 ExtendSim\Extensions folder. Unlike other resources, the "movieName" is its file name, not a resource name.	V
AnimationMovieFinish(integer obj)	(Mac OS 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
AnimationObjectCopyData(integer objNum, integer objNum2)	This function copies the animation object data, (like type of object, poly points, color, size, etc.) from one object to another one.	I
AnimationObjectCreate(integer enclosingHblockIfTRUE)	Creates an animation object on the fly. The animation object number for the created object will be returned. If the enclosingHblockIfTRUE argument is set to one, or TRUE, the function will create the animation object in the enclosing Hblock of the current block. In this case, the returned value will be the negative value of the animation object number. Animation objects create on the fly with this function are not part of the block structure, and will need to be recreated in openModel, or InitSim, or whenever they are used. They will persist until the model window (Including Hblock model windows,) is closed, or AnimationObjectDelete is called.	I
AnimationObjectDelete(integer objNum)	This function will delete an animation object. This will not affect animation objects that are defined in the block structure, just those created on the fly with AnimationObjectCreate. Use negative objNum if in enclosing Hblock.	I
AnimationObjectExists(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

Animation	Description	Return
AnimationObject-Exists2(integer blockNumber, integer objNum)	Similar to the AnimationObjectExists function, except that this function includes a block number argument, allowing you to check for the existence of an animation object in a remote block.	I
AnimationOval (integer obj)	Defines the object as an oval.	V
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 90 for important information about pictures.	V
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. ExtendSim normalizes the size of the pixels to conform to the animation object size. *IntArray* is declared as a dynamic array: `integer intArray[];`	V
AnimationPixel-RectEColor-Init(integer objNum, integer EColorValue)	Initializes or sets the color value of all the pixels in an Animation Pixel Rectangle to the specified color. (This is quicker then looping through each pixel.)	V
AnimationPixelSet (integer objNum, 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. Note: As of ExtendSim 10, see instead AnimationPixelRectEColorInit	V
AnimationPixelSe-tEColor (integer objNum, integer row, integer col, integer EColor, integer drawNow)	Same as AnimationPixelSet, above, except it takes an EColor value for the color instead of H/S/V values.	V
Animation-Poly(integer objNum, integer pointCount, integer pointArray[])	Animates a polygon. PointCount is the number of points to be drawn, point array is a two dimensional array of points that contains the points to be animated. The points are defined in pixels from the upper left hand corner of the animation object location.	V
AnimationRectangle(integer obj)	Defines the object as a rectangle.	V
AnimationRn-dRectangle(integer obj)	Defines the object as a rounded rectangle.	V

Animation	Description	Return
AnimationSetDelayMode(integer trueIsDelay)	If trueIsDelay is TRUE (default), then ExtendSim uses its normal delay, corresponding to the animation speed that the user chose. FALSE removes any animation delay, no matter what speed the user chooses. Use this function to set your own delay for animation and have ExtendSim ignore the animation speed set by the user. See AnimationGetSpeed(), above, to see what speed the user chose.	V
AnimationSetSpeed(integer newSpeed)	Sets the animation speed, with 1 being slowest and 5 being fastest. See AnimationGetSpeed(), above.	V
AnimationShow (integer obj)	Shows a hidden object.	V
AnimationStretchTo(integer obj, integer leftOffset, integer topOffset, integer width, integer height, integer outsideIcon)	Stretches the object relative to its original position in the icon.	V
AnimationText (integer obj, string msg)	Defines the object as text with the string *msg*. Also see the TextWidth function in *Strings*, below, that is useful in applying the AnimationStretchTo function to the object so that the text width is accommodated.	V
AnimationTextAlign (integer objNum, integer justification)	This function specifies the text alignment for the specified animation object. *Justification* takes the following values: Left: 0 Right: -1 Center: 1	V
AnimationTextSize(integer objNum, integer size)	Allows you to modify the size of animation text. The size argument is a font point size.	V
AnimationTextTransparent(integer objNum, string text)	Exactly the same as the animationText function, except the background behind the text will be transparent. It's normally white for the regular function.	V
AnimationZOrderGet(integer objNum)	Gets the zOrder value of the specified animation object. Note: zOrders for animation objects only control the zOrder relative to the other animation objects from the block that controls the animation. Animation objects are children of the block that they are associated with, and are all ordered relative to that block's zOrder.	R
AnimationZOrderSet(integer objNum, real zOrder)	Sets the zOrder value of the specified animation object. See note about zOrders above.	V

Animation	Description	Return
DialogPicture (string variableName, string pictureName, 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
PictureList (String15 arrayName[], integer type)	Resizes, and fills the dynamic array *arrayname* with the names of the pictures from the extension folder or the application to be used with the AnimationBlockToBlock() function. Return value is the number of pictures. *type*: 0 - All pictures in the Extensions folder. 1 - Only AnimationBlockToBlock pictures in the application. 2 - Only AnimationBlockToBlock pictures in the Extensions folder. Pictures with names that start with a "@" character are not AnimationBlockToBlock pictures and are not returned. The @ character prevents pictures from showing up in the block's animation tab popup menu. 3 - All non-AnimationBlockToBlock pictures (pictures that start with an @ character.	I
ProofEncode (string commandLine)	The version of the Proof Animation DLL that works with ExtendSim requires a numeric value calculated by this function to be passed to it periodically. See the Proof Animation blocks in the Animation library to see how to use this function.	I
ProofEncodeReset()	Resets counters for Proof copy protection.	V

Blocks and inter-block communications

Block numbers, labels, names, categories, position

As seen in their Properties, all objects in a model (blocks, text, shapes, connection lines, and so forth) are numbered uniquely and sequentially from 0 to n-1 as they are added to the model worksheet. In most cases, the numbers assigned to each object do not change. However, if a block or text is deleted, its number becomes an "unused slot" which is available when another object is added to the model.

Numbers

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.

Names and labels

- *Block names* are the names you give a block when you create it. For example, "Data Import Export" is the name of a block in the Value 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 at the bottom of a block's dialog. Labels appear at the bottom of the block's icon.

Categories and types

- *Categories* are used to group blocks in the library menu based on some commonality. Examples of categories include *Math* or *Holding* from the Value library. Categories are set using the Develop > Set Block Category menu command.
- Block types are a way to delineate Item library blocks as residence, passing, or decision.

Block numbers, labels, etc.	Description	Return
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
BlockRect (integer blockNumber, integer or real array[4], integer useAnimation)	Virtually the same as the getBlockTypePosition function, except for the useAnimation argument. This argument specifies (with a true/false, 1/0 value) whether or not the animation objects are to be included in the returned rectangle. If useAnimation is true, the rectangle will include the positions of the animation objects that are off the icon in the rectangle, otherwise it will not.	I
FindInHierarchy (string FindBlockName, string FindBlockLabel, string FindDialogName, integer FindDialog)	This function is used by several blocks, including the Catch Item and Throw Item blocks in the Item library, to locate the corresponding block associated with the current block. (For example, to find a Catch corresponding to 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; it returns a –1 if no matching block is found. FindBlockName is the name of the block to be found (e.g. 'Catch', or 'Throw'). FindBlockLabel is the block label of the block to be found. FindDialogName is the name of the dialog item you wish to search for. The FindDialogName field should be filled with the dialog item name, a colon, and then the value of the dialog item that you wish to search for. For example Item:54 will search for the presence of a dialog item with the name "item", and the value "54". FindDialog takes the following values: 0: just check the block name, and the block label. 1: just check the block name and the dialog item name and value. 2: check the name, label, and dialog item.	I
FindInHierarchy2(integer blockNumber, string findBlockName, string findBlockLabel, integer findDialog)	The same as the FindInHierarchy function except it takes a block number argument.	I

Block numbers, labels, etc.	Description	Return
GetBlockLabel(integer i)	Returns the label string for the global block *i*.	S
GetBlockMemSize(blockNumber)	Returns the number of bytes of memory used by the block.	I
GetBlockType(integer i)	Returns the string that represents the category (e.g. *Math* or *Holding*) for the global block *i*. Block categories are set in the Develop > Set Block Category menu.	S
GetBlockTypeNumeric(objectID)	Like GetBlockTypePosition() this function returns the block type but since it doesn't have to calculate the block position, it is faster.	I
GetBlockTypePosition(integer i, integer or real Array[4])	In versions prior to 10, many objects (anchor points, blocks, text, etc.) were referred to as "blocks". This function returns a specified Type for the global block *i*. Types are as follows: 0: empty slot 1: anchor point 2: text 3: block 4: hierarchical block 5: embedded object (obsolete as of ExtendSim 10) 6: Slider, Switch, or Meter (from the Model > Controls command prior to ExtendSim 10) The function puts an integer or a real in the array depending on whether the array is defined as integer or real. On return, array[0] will contain the top, array[1] the left, array[2] the bottom, and array[3] the right position's pixel values. To include animation objects, see the blockRect function. ☞ See also the GetBlockTypeNumeric and ObjectIDNext functions, which are faster for certain situations.	I
GetEnclosingHblockNum()	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
GetStaticNames (integer blockNumber, array names, array types)	Fills the names and types arrays with the names and types of all the static variables defined in the block structure. Names should be an array of strings, and types should be an array of longs. The value of the cells of the types array will be from the following list: 3: LONGTYPE 6: DOUBLETYPE 7: STRING255TYPE 8: STRING15TYPE 9: STRING31TYPE 10: STRING63TYPE 11: STRING127TYPE	I

Block numbers, labels, etc.	Description	Return
GlobalToLocal (integer blockNumber)	Returns the local block number for the specified block if it is contained in an H-block. Returns a -1 if the block is not contained in an H-block.	I
IconBody (blockNum, real array[4], useAnimation, useConnectors)	Similar to the BlockRect function. Returns the rect for the body of the icon, allowing (via the useAnimation and useConnectors arguments) control over whether the animation objects and/or connectors are included in the rect. Returns a zero for success or a non zero value for an error.	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
LocalToGlobal (integer i)	Converts a local number in the hierarchical block from which this function is called to a global number.	I
LocalToGlobal2 (integer HblockNum, integer localBlockNum)	Similar to LocalToGlobal, this function will return the global blocknumber 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
MakeOptimizerBlock (integer trueFalse)	This function tags the block as an optimizer block so that the Run > 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 Optimizer block (Value library).	V
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
MyLocalBlockNumber()	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

Block numbers, labels, etc.	Description	Return

ObjectIDNext(Integer fromBlock, integer which)	Returns the objectID of the next object of the type requested, as controlled by the *which* variable. You can step through all the blocks in the model quite a bit faster than in V9 since, depending on the *which* value, you can just jump to the next block.
	
	FromBlock specifies where in the list to begin, so it will normally be started at -1. *Which* currently takes the arguments:
	0: blocks
	1: Hblocks
	2: both regular and Hblocks
	
	The pseudo code for using the function:
	objectID = objectIDNext(-1, 0);
	while (objectID > -1)
	{
	objectID = objectIDNext(objectID, 0);
	}
	
	This will loop through all the blocks in the model without needing to call numblocks or getBlockTypePosition.
		I

SetBlockLabel (integer i, string str)	Sets the label for the global block *i* to *str*. Blocks are numbered from 0 to NumBlocks()-1.	V
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*.	V

Block connectors and connection information

The following functions give you information about connectors and connected blocks in a model. This helps you manipulate connectors and 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, categories, position" on page 256), "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.

Also see "Variable connectors" on page 264, below, to see their use and to see the connector labeling functions.

Block connection	Description	Return
AlignConnection (long blockFrom, long conFrom, long blockTo, long conTo, long vertical)	Adjusts the position of the second block (blockTo) to make the connection line between the two connectors specified straight.	I

Block connection	Description	Return
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
GetConHblocks (integer blockNumber, integer Conn, integer intArray [][2])	Similar to the GetConBlocks() function, above, but returns a list of connected Hblocks. You can also start from a textblock or anchor point if you know the block number (connector numbers are zero for textblocks or anchor points).	I
GetConName (integer Block, integer Conn)	Name of the connector.	S
GetConnectedTextBlock (integer Block, integer Conn)	If *block* is positive, returns the block number of the named connection (bypasses connector text blocks inside an Hblock and goes out Hblock connector) connected to the connector *conn* of block number *block*. If *block* is negative, stay inside the Hblock (can return the connector text block inside of an Hblock). Use the BlockName() 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 ExtendSim constants: 13 Value (passes values) 14 Item (passes discrete event items) 15 Universal (other types of connectors can connect to it) 16 User-defined (can be programmed with custom behavior)) 25 Flow (passes the effective rate) 308 Reliability (passes reliability information) -1 Array (passes arrays of data) 309 Box (other purpose)	I
GetConnectedType2(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
GetConnectionColor (integer blockFrom, integer conFrom, integer blockTo, integer conTo, integer colorArray[])	Fills the *color* array (three integer element) with the hue, saturation, and brightness (HSV) values for the color of the connection line. The color selector is on page 366. Returns 0 for success or a negative value to indicate failure. Note: As of ExtendSim 10, see instead GetConnectionEColor.	I

Functions

Block connection	Description	Return
GetConnectionEColor (integer blockFrom, integer conFrom, integer blockTo, integer conTo)	Returns the value for the EColor of the specified connection line. The EColor selector is on page 366.	I
GetConnectorPosition(integer blockNumber, integer Conn, integer or real Array[4])	Returns the position of the specified connector in pixels. Array should be defined as a four-row integer or real array (`integer array[4];` or `real array[4];`), which on return from the function call will be filled with the four values: array[0] = top array[1] = left array[2] = bottom array[3] = right	I
GetConnectorType(integer blockNumber, integer Conn)	Returns the connector type of the specified connector. 13 Value (passes values) 14 Item (passes discrete event items) 15 Universal (other types of connectors can connect to it) 16 User-defined (can be programmed with custom behavior)) 25 Flow (passes the effective rate) 308 Reliability (passes reliability information) -1 Array (passes arrays of data) 309 Box (other purpose)	I
GetConNumber (integer blockNumber, string conNameStr)	Returns the Connector number of the connector of the specified name.	I
GetEnclosingHblockCon(integer blockNumber, integer conNum)	Used to find the connector index of the outer connector, given a blockNumber and connector index inside the Hblock that is connected to the Hblock's internal connector text. Returns connector number of the enclosing Hblock's outer conector or -1 if not an Hblock.	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
GetIntermediateBlocks(integer blockNum1, integer conn1, integer blockNum2, integer conn2, array)	Returns number of connected dot blocks, text blocks, and connectors between two blocks. Fills the array with information about the blocks. See GetConBlocks function for definition of the array. If blockNum2 is negative, returns above information between a block (blockNum1) and the text block or connector text block specified by blockNum2 (can be inside a hierarchical block).	I

Block connection	Description	Return
GetNumCons(integer Block)	Number of connectors on the block.	I
GetRightClicked-Con()	This function returns the conn number of the last connector that was right clicked on. This function should be called in the CONNECTORRIGHTCLICK message.	I
IsConVisible(integer blockNumber, integer Conn)	Tests the specified connector for visibility. Returns TRUE if the connector is visible, FALSE otherwise	I
MakeFeedback-Block(feedBack)	If *feedBack* is TRUE, this function causes ExtendSim 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 (Utilities library).	V
NodeGetCurrent-Value(nodeIDIndex)	Returns the currently set value of the connected block connectors. See NodeGetIDIndex(), below.	R
NodeGetIDIndex(blockNumber, conNum)	Returns the nodeID for a connected network of block connectors. Each connected network has a unique nodeID, which can change when additional block connectors are connected or the model is reopened. The nodeID is equivalent to an index of a real value that holds the set value of the connected connectors.	I
SetConnection-Color (integer blockFrom, integer conFrom, integer blockTo, integer conTo, integer hue, integer saturation, integer value)	Sets the color value of the specified connection line. See “Select Color window” on page 366 for a description of the hue, saturation, and value arguments. This function returns a TRUE if it succeeds and a FALSE if it fails. Note: as of ExtendSim 10, see instead SetConnectionEColor	I
SetConnectionE-Color (integer blockFrom, integer conFrom, integer blockTo, integer conTo, integer EColorValue)	Sets the EColor value of the specified connection line. See “Select Color window” on page 366. This function returns a TRUE if it succeeds and a FALSE if it fails.	I
SetConnection-Thickness (long blockFrom, long conFrom, long blockTo, long conTo, long value)	Sets the connection line thickness of the specified connection line. Returns a zero for success or a non zero value for an error.	I

Block connection	Description	Return
SetConVisibility(integer blockNumber, integer conn, integer visible)	Sets the visibility flag on an individual connector. This function can be used by block developers to hide and show connectors based on choices the user has made in the dialog.	V
SetIndexedConValue(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 connectors pane.	V
SetIndexedConValue2(integer block, integer conn, real value)	This is same as SetIndexedConValue(), except it refers to a global block number.	V
SetSelectedConnectionColor (integer hue, integer saturation, integer value)	Sets the color value of all selected connections. See "Select Color window" on page 366 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
SetSelectedConnectionEColor (integer EColorValue)	Sets the color value of all selected connections. See "Select Color window" on page 366. This function returns a TRUE if it succeeds and a FALSE if it fails.	I

Variable connectors

These functions are used with a block's variable connectors, and to interface them with the standard connector functions. Note that all of the single connector functions, above, also work. Where *conn* is specified in the argument list, use ConArrayGetConNumber() to find the actual connector number on the block from the array owner and nth connector in the array. For example, to send a message out of a variable connector, call ConArrayGetConNumber() to get the actual number of the connector (its index). Then call SendMsgTo…() and instead of using the actual connector name (e.g. FlowIn), use V7's new feature and use the connector number you got from ConArrayGetConNumber().

These connector functions use a block number and the connector name as a string so you can use them to control other blocks.

Variable connectors	Description	Return
ConArrayChangedWhichCon()	When the user drags to get more or less variable connectors, the block gets a CONARRAYCHANGED message. This function returns the owning connector name, as a string, that is being dragged. Use this string in the ConArrayGetNumCons() function to get the actual number of connectors during the dragging operation.	

Variable connectors	Description	Return
ConArrayGetCollapsed(integer blockNumber, string origConName)	Returns True if the specified connector is collapsed, or false if it is not.	I
ConArrayGetConNumber(blockNumber, origConNameStr, nthConn)	Returns the connector number of the nth connector in this array. OrigConNameStr is the owning connector name as a string. NthConn is the index of the array connector, starting from 0 for the owning connector, to the number of connectors in this array minus 1 for the last connector. The returned connector number can be used in the other connector functions.	I
ConArrayGetDirection(blockNumber, origConNameStr)	Returns the array direction as: 0 top, 1 right, 2 bottom, 3 left. If not an array connector or an error, returns -1.	I
ConArrayGetNthCon(conn)	Given a block connector number, returns the member index of its connector array (e.g. Given the block connector number of ConIn[3] (could be 253), returns 3 meaning ConIn[3]). A -1 returned is an error.	I
ConArrayGetNthCon2(blocknum, conn)	Same as ConArrayGetNthCon except it has an additional argument to specify the connector on a different block.	I
ConArrayGetNumCons(blockNumber, origConNameStr)	Returns the number of connectors in this array. If there are no added array connectors, returns 1 for the original connector. OrigConNameStr is the owning connector name as a string.	I
ConArrayGetOwnerCon(conn)	Given a block connector number, returns the block connector number of the owning (originating) connector of a connector array (e.g. Given the block connector number of ConIn[3] (could be 253), returns the block connector number of ConIn[0]). A -1 returned is an error.	I
ConArrayGetTotalCons(blockNumber)	Returns the total number of connectors in this block, including array connectors. Used to prevent a dragged connector array from adding too many connectors, causing the block to have more than its limit of 255 connectors.	I
ConArrayGetValue(ConName, nthConn)	Returns the value of the nth connector of the array owned by ConName. ConName can be the name of the connector without quotes or, new for V7, the index from 0 to numCons-1 as a constant or variable (If the compiler detects a connector name, it turns it into an index rather than evaluating its value).	R
ConArrayMsgFromCon()	Returns the index of the connector that received the message in an array. For example, if the Con1In message handler received a message, calling this function at the beginning of the message handler would return which connector in this array (from 0 to num-1) received the message.	I

Variable connectors	Description	Return
ConArraySendMsgToAllCons(integer origConName, integer nthConn)	Sends a connector message to all connectors connected to the specified Connector. See SendMsgToAllCons.	V
ConArraySendMsgToInputs(integer origConName, integer nthConn)	Sends a connector message to input connectors connected to the specified Connector. See SendMsgToInputs.	V
ConArraySendMsgToOutputs(integer origConName, integer nthConn)	Sends a connector message to output connectors connected to the specified Connector. See SendMsgToOutputs.	V
ConArraySetCollapsed(integer blockNumber, string origConName, integer trueOrFalse)	Sets the collapsed state on the specified connector. True will collapse the connector, and False will uncollapse it.	V
ConArraySetNumCons(blockNumber, origConNameStr, newNumCons, trueToIgnoreConnections)	Call this to set the number of connectors in a variable connector (a 1 for NewNumCons means to only keep the original connector). If trueToIgnoreConnections is TRUE, the user can delete connectors that have connections on them. If trueToIgnoreConnections is FALSE, connectors that have connections on them will not be deleted. This means that connections may prevent all of the desired connectors from being deleted.	V
ConArraySetValue(ConName, nthConn, value)	Sets the value of the nth connector of the array owned by ConName. ConName can be the name of the connector without quotes or, new for V7, the index from 0 to numCons-1 as a constant or variable (If the compiler detects a connector name (not a string), it turns it into an index rather than evaluating its value).	V
ConnectorLabelsGet(blockNumber, origConNum, labelsStrArray[])	Call this to get the connector labels for single connectors or connector arrays. labelsStrArray is any kind of string array (local, static, or dynamic) that will contains all the labels used, one per array element, which will allow up to n labels to be retrieved. Each label is the ith connector in that connector array (0th for single connectors). Returns the number of labels retrieved.	I
ConnectorLabelsSet(blockNumber, origConNum, numLabels, labelsStrArray[], position, hue, saturation, value)	Call this to set the connector labels for single connectors or connector arrays. labelsStrArray is any kind of string array (local, static, or dynamic) containing all the labels needed, one per array element. The labels can contain combinations of style information such as <biur> for bold, italics, underlined, right adjusted. Positions are: 0 top, 1 right, 2 bottom, 3 left.	V

Connector tool tips

The System message ConnectorToolTip is sent to the block when the cursor hovers over a connector. In that message handler, the connector tool tip functions (below) will allow you to customize and access the tooltip that will appear.

Connector tool tips	Description	Return
ConnectorToolTipGet(integer blockNumber, integer connectorIndex)	Gets the current string value of a connector tool tip from a block and connector. This is useful if you want to show what block your block is connected to, like the batch and select blocks.	S
ConnectorToolTipSet(stringstring, integer replace)	Allows you to set the string that will be displayed. If the replace flag is true, the default string generated by ExtendSim will be replaced, if it is false, this string will be appended to the default string.	V
ConnectorToolTipWhich()	When the ConnectorToolTip message is received, this function returns the connector index of the connector the cursor is over. This function is also used in the ConnectionMake message to return the connector being connected.	I

Dialog items

These functions work with a block's dialog items and can be used to get and set values from dialog items in other blocks, as well as change dialog item colors, create popup menus, hide and show items, etc.

The functions in this list that include a global block number in their argument list can affect any block in the model, including the calling block if its own block number (from the MyBlockNumber() function) is used.

☞ Block controls (switch, slider, and meter) can be set or changed via the Get/SetDialogVariable functions or poke/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 therefore 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 controls 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.

☞ For additional formatting options, see "Formatting/interactivity using column and parameter tags" on page 284. These facilitate both specialized formatting and mouse-interactivity with parameters and data tables. Also see "Block data tables" on page 274, "Dynamic text items" on page 283, and "Dynamic linking" on page 280.

Block dialog items	Description	Return
AppendPopupLabels(string variableNameStr, string theLabels)	This function appends the specified string onto the labels associated with the named popup menu. Popup menu labels can total up to 5100 characters. This function is the method for adding menu labels. *VariableNameStr* is the dialog item name as a string or in quotes. See SetPopupLabels() below.	I

Block dialog items	Description	Return
CreatePopupMenu (string string1, string string2, integer initialSelection)	Creates a Popup at the last location the user clicked. This function can be used in conjunction with the on dialogClick message handler, the whichDialogItemClicked function, and the whichDTCellClicked function to create a popup menu that appears on a dialog item, or data table cell in response to a user's click. See the code of the on dialogClick messagehandler of the Activity block (Item library) for an example of how to use this function. Also see PopupMenuArray(), below.	I
DialogHasEmbeddedObject (integer blockNumber)	OBSOLETE AS OF ES10 Returns the dialog item name if the specified block contains any embedded object dialog items in its dialog. This function will return an empty string "" if there are no dialog items of this type in the dialog.	S
DialogItemVisible (integer blockNumber, string variableNameStr, integer 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
DIGetID(integer blockNumber, string name)	Returns a dialog item ID number. This is used in the LINKCONTENT and LINKSTRUCTURE message handlers to help identify which dialog item is getting the message.	I
DIGetName(integer blockNum, integer dialogID)	Returns the dialog item name from the item's dialogID	S
DIMoveBy(integer blockNumber, string name, integer y, integer x)	Offsets the dialog item by the values of the y and x parameters.	I
DIMoveTo(integer blockNumber, string name, integer top, integer left)	Moves the specified dialog item to the y and x location specified by the top and left variables.	I
DIMsgNumber(integer blockNumber, string name)	Returns the message number associated with a dialog item. Used to send a dialog item message to a block.	I
DIPopupButton(integer blockNumber, string name, integer behavesAsButton)	Changes the behavior of the specified popup menu so that it will not show the typical popup behavior when it is clicked, but will instead behave like a button (Sending a message to the MODL code, but not displaying a popup menu). This is used when the block developer wants something that looks like a popup menu, but has a button's behavior.	I

Block dialog items	Description	Return
DIPosition-Get(integer block-Number, string name, integer array position)	Gets the position of a dialog item specified by blockNumber, and dialogItemName. The coordinates of the dialog items location will be placed into the integer array position. Position needs to be declared as: integer array[4];	I
DIPosition-Home(integer blockNumber, string name)	Resets the location of the dialog item back to the position and size defined in the structure.	I
DIPositionSet(integer blockNumber, string name, integer top, integer left, integer bottom, integer right)	Sets the position of the specified dialog item. This function also includes the bottom and right arguments so you can change the displayed size of the dialog item as well.	I
DisableDialogItem (string variable-NameStr, Integer TRUEFALSE)	Disables, or enables the specified dialog item.	V
DisableDialog-Item2(integer blockNumber, string name, integer disableEnable)	Disables the specified Dialog Item. This function differs from the DisableDialogItem() function in that it takes a block number argument so you can disable a dialog item from a remote block.	I
DISetFocus(integer blockNumber, string dialogItem-Name)	Sets the focus on the specified dialog item. Returns 0 for success or a negative value to indicate failure.	I
DISetParent (blockNum, dialog-ItemName, dialog-ItemName)		
DITitleGet(integer blockNumber, string dialogItem-Name)	Returns the string title of the dialog item. For dialog items like radio buttons and check boxes, the title is the text that appears on the dialog. Use this function to retrieve the string if it has been set by the function DITitleSet.	S
DITitleSet(integer blockNumber, string name, string title)	Sets the title of a dialog item. Useful for Dialog items like radio buttons and checkboxes where there is no other way to change the title of the dialog item on the fly. ⚠ If you use the ampersand character (&) in the label of a Radio Button, Checkbox, or Frame dialog object you will need to enter it twice (&&). Otherwise, the character will not show on the label.	I

Functions

Block dialog items	Description	Return
GetDialogColors (integer blockNumber, integer HSVColorArray[][3])	Copies all the color information from the dialog colors into the color array. Used to save all the colors in a dialog in on dialog close. The HSVColorArray is declared as a dynamic array: integer HSVColorArray[][3];	I
GetDialogItemColor (integer blockNumber, string variableNameStr, integer HSVColorArray[3])	Gets a color value associated with the dialog item. For ExtendSim 10 or later, use the GetDialogItemEColor function.	I
GetDialogItemEColor (long blockName, string dialogItemName)	Returns the EColor value of the color of the dialog item. See "EColors" on page 365 for more information.	I
GetDialogItemInfo(integer blockNumber, string variableNameStr, integer which)	Returns TRUE if *which* qualities are true for the dialog item: 0: exist 1: hidden 2: enabled 3: display only 4: dialog item type (see below) 5: rows 6: columns 7: width 8: height 9: left 10: top 11: number of the dialog tab Dialog item types (which = 4) are: 1: button, 2: checkbox, 3: radiobutton, 4: meter, 5: parameter, 6: slider, 7: datatable, 8: edittext, 9: stattext, 12: switch, 13: stringtable, 14: plotpane, 16: popupmenu, 17: embedobject (obsolete as of ES 10), 18: dynamic text, 19: textframe, 20: calendar, 21: edittext31	I
GetDialogItemLabel (integer blockNumber, string variableNameStr, integer n)	Returns the nth item label. For example, the nth item on a popup menu or the nth column header in a data table. Returns an empty string "" if the wrong type of item or no label is found.	S

Block dialog items	Description	Return
GetDialogNames(integer block, string nameArray[], integer typeArray[])	Returns a list of the dialog variables in the specified *block*. Both *nameArray* and *typeArray* are dynamic arrays. 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 *typeArray* will contain the type of dialog item. Values for *typeArray* are: 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 (obsolete as of ES10), 18:Dynamic Text, 19:Text Frame, 20:Calendar, and 21:Editable Text31 (31 characters)	
GetDialogVariable (integer blockNumber, 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. ☞ See also the GetStaticVariable function and the GetVariableNumeric function which should be faster for querying non-string values.	S
GetDraggedCloneList (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 Optimizer block (Value library).	I
getStaticVariable(integer blockNum, string staticVariableName, long row, col)	Similar to GetDialogVariable, this function allows the user to explicitly retrieve the value of a static variable. GetDialogVariable will already do this, but GetDialogVariable looks for any dialog variable with the specified name first. The advantage of this function is speed. If you know whether you are looking for a dialog variable or a static variable, you can call the appropriate function and the function will not have to look for the other kind of variable.	S
GetSystemColor (integer whichPart, integer whichColor)	Gets the system colors so that they can be matched in a dialog. Returns whichPart: 1:R, 2:G, 3:B, 4:H, 5:S, 6:V, 7:xxBBGGRR whichColor: 0:Dialog BackGround, 1:ToolTipColor, 3:ScrollBar color	I

Block dialog items	Description	Return
GetVariableNumeric (integer blockNum, string name, integer row, integer col)	Similar to GetDialogVariable, but returns a numeric (real) value. Should be faster for querying non-string variables.	R
HideDialogItem (string variableNameStr, integer hideShow)	Hides the named dialog item if *hideShow* is TRUE. *DialogNameStr* is the dialog item name as a string or in quotes. Returns 0 if it succeeds.	I
HideDialogItem2 (integer blockNumber, string name, integer hideShow)	Set the hidden/shown status of the item. See the HideDialogItem() function, with the addition of the block number argument, which allows you to call the function from outside a block. Returns 0 if it succeeds.	I
LastSetDialogVariableString()	Returns the last string value that was set by SetDialogVariable. This is useful if one is setting the value of a dialog item like a popup menu, where the stored value is not the value that the popup menu contains, but rather an additional string variable. In this case, using the WhoInvoked() function, below, and this function you can have the code of the block change the correct string for the dialog item, so the SetDialogVariable function will react as the user expects, even though the data is not directly contained in the dialog item.	S
OpenAndSelectDialogItem (integer blockNumber, string variableNameStr)	Obsolete. Please see OpenAndSelectDialogItem2(), just below this function. This version of the function did not have the row and column indexes to select a cell in a data table.	V
OpenAndSelectDialogItem2 (integer blockNumber, string variableNameStr, integer row, integer col)	Opens the block's dialog, highlighting (selecting) the dialog item corresponding to varName. If the item is a data table, row and col are the indexes. Row and col are ignored if the item is not a data table. If row and col are -1, selects the entire data table.	V
PopupCanceled()	Returns whether or not the last popup menu was canceled. This can be called immediately after a ‘flying’ popup menu has been created, to determine if the user canceled the popup menu action or not. Canceling would be clicking off the popup menu, without making a selection.	I
PopupItemParse(itemString)	This function parses the string that is passed in, and removes the special characters that are used in certain dialog item contexts. Specifically, it removes the formatting notation (e.g. <RB> for right adjusted, bold), characters that are user for formatting, and returns the stripped string.	S

Block dialog items	Description	Return
PopupMenu-AppendArray(string dialogItemName, stringArray array)	Appends the contents of the string array array to the popup menu specified by dialogItemName.	I
PopupMenuArray (string theArray[], integer initialValue)	This function creates a flying popup menu based on the strings in theArray, maximum 20 strings (5100 characters total). Other than the fact that an array of strings is passed in instead of separate strings, it's exactly the same as the CreatePopupMenu function.	I
SetDialogColors (integer blockNumber, integer HSVColorArray[][3])	Sets all the colors of the dialog items at once. Usually used with the array from GetDialogColors(), above.	I
SetDialogItemColor (integer blockNumber, string variableNameStr, longArray HSVValues)	Sets a color value associated with the dialog item. Some items will not redraw with this color, but instead have their color defined by the operating system settings. See Note: For ExtendSim 10 or later, use the SetDialogItemEColor function. That function also lists which dialog items are and are not able to be redrawn in the selected color.	V
SetDialogItemEColor (long blockName, string dialogItemName, long value)	Based on the EColor value that is passed in, sets the color of these dialog items: parameter, popup menu, static text, editable text, editable text 31, frame. Other dialog items (button, check box, radio button, dynamic text, data table, text table) and dialog controls are not impacted by the API but instead have their color defined by the operating system settings. See "EColors" on page 365 for more information.	I
SetDialogVariable (integer blockNumber, 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. ☞ See also SetVariableNumeric, which should be faster when setting non-string values.	V
SetDialogVariable-NoMsg(integer blockNumber, string variableNameStr, string value, integer row, integer col)	Same as SetDialogVariable function, but doesn't send a dialog item message to the block.	V

Block dialog items	Description	Return
SetPopupLabels(string variableNameStr, 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. *VariableNameStr* is the dialog item name as a string or in quotes. See AppendPopupLablels() above to add more labels.	V
SetVariableNumeric (integer blockNum, string name, real value, integer row, integer col, integer msg)	Similar to SetDialogVariable, except it takes a numeric (real) value. Should be faster when setting non-string values	V
SetVisibilityMonitoring (integer blockNumber, string variableNameStr, integer monitor)	Turns on monitoring of the visibility of a dialog item. This sets a flag on a dialog item that makes it send a message to the block code (DIALOGITEMREFRESH) when the dialog item (Or one of its clones) becomes visible. This is useful if you have a dialog item whose redraw includes some time-consuming calculation and you want to be able to turn off the calculation unless the dialog item is visible.	I
WhichDialogItem()	Returns the name of the current dialog item in certain contexts. This function can be used in the CELLACCEPT, DATATABLERESIZE, DATATABLESCROLLED, DIALOGITEMTOOLTIP, DIALOGCLICK, or dialog item name message handlers.	I
WhoInvoked()	Determines the source of the invocation of the current message handler. Currently implemented in two cases: 1) Called in a dialog item message handler, this function will return a 1 if the message handler was invoked from SetDialogVariable, or a 0 if the handler was invoked through user interaction. 2) Called in OpenModel, this function will return a 2 if the OpenModel handler is being executed from the placement of an Hblock, or a 0 if it's being called because the model containing the block is being opened.	I

Functions

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 data table from which you want to retrieve the selection; this name needs to be either a string variable or a string in quotes.

☞ Starting in ExtendSim 7, you must access all data tables using their dialog item variable name. Even dynamic data tables need to be accessed using the dialog variable name. This was not the case in Extend 6 or earlier, but is a ModL compiler change that is necessary to allow data tables to transparently access linked database or global array data, and to be variable column. Using the dynamic array name will not allow linked or variable column data tables to work.

☞ If you need large data tables, see the Dynamic data table functions below.

Variable column data tables

The DynamicDatatableVariableColumns function, and the supporting functions that have been built for it, allow the creation and use of a data table with both variable rows and variable columns. The starting point is a call to the DynamicDatatableVariableColumns function which will link a dynamic array to a data table, in much the same way as the DynamicDatatable function, with the addition of allowing the user to define the number of columns that the table supports.

Each time you want to change the number of columns in the data table, you call DynamicDataTableVariableColumns again with the new number of columns. The internal implementation of this construct is basically that the link between the data table and the dynamic array contains the information about the number of columns. This means that you cannot use the dynamic array variable name to refer to the data, as the dynamic array still has the original definition of how many columns it has. To reference your data with the variable number of columns, you need to refer to it using the data table variable name, as the data table will use the information in the link to determine how many columns it has. See "OLE/COM (Windows only)" on page 232 for OLE functions that fill variable column data tables.

Dynamic data table resizing

The following functions are designed to be support functions for dynamic data table resize functionality. This functionality is implemented partially through block code, and partially through the ExtendSim Application. The basic functionality, as seen on dialog boxes in both the Item and Value libraries, is a button on the lower right hand corner of the data table that can be clicked popping up a dialog that allows the user to specify the resizing options for the data table. The application level implementation is the drawing and behavior of the button on the data table. The presence or absence of the button can be controlled from the block code via the DTGrowButton function described below. When the button is clicked, the block will receive a DATATABLERESIZE message. The code of this message handler is where the majority of the block code controlling the resizing will be executed. Normally the code in this message handler will first be a call to NumericParameter, or NumericParameter2 to determine the desired resizing of the table, followed by the necessary code to implement the resizing. See the Variable Columns data table section above. See the blocks in the Value and Item libraries for examples of this code.

Data table linking

Data tables can be linked to global arrays or an ExtendSim database by the user clicking the Link button at the lower left corner, or using the linking functions below. When the data or structure that they are linked to changes, LINKCONTENT or LINKSTRUCTURE messages are sent to individual blocks that are subscribed to an ExtendSim database or global array. See "Dynamic linking" on page 280.

☞ Starting in ExtendSim V, you must access all data tables using their dialog item variable name. Even dynamic data tables need to be accessed using the dialog variable name. This was not the case in Extend 6 or earlier, but is a ModL compiler change that is necessary to allow data tables to transparently access linked database or global array data, and to be variable column. Using the dynamic array name will not allow linked or variable column data tables to work.

Formatting individual columns

For additional formatting options, see “Formatting/interactivity using column and parameter tags” on page 284. These facilitate both specialized formatting and mouse-interactivity with data tables.

Block data tables	Description	Return
AppendDataTable-Labels (string DTname, string theLabels)	Appends another string of labels to the data table labels list of the data table DTname.	I
DisableDTTabbing (integer blockNumber, string DTname, integer disableDT)	Disables or enables the tab key functionality for the specified data table.	I
DTGrowButton (integer blockNumber, string DTname, integer showButton)	Shows/hides the resize/grow button on the specified data table. This button is hidden by default, as block code is necessary for the resizing functionality.	I
DTHasDDELink (integer blockNumber, string DTname)	Returns true if the specified data table has an IPC advise link, or a Paste link, associated with it.	I
DTPaneFixed (integer blockNumber, string name, integer fixed)	Changes the behavior of the specified data table, setting or unsetting its internal ‘fixed’ flag. By default this flag is off, but if it is turned on, the data table will retain its width when the number of columns in the data is reduced, rather than resizing smaller, as it can do in the default case.	I
DTResizeToCols (integer blockNum, string DTName)	Forces the data table to resize to the width of the resized columns. This function will not resize the columns, but instead will resize the data table object to the width of the current column size. Returns a zero for success or a non zero value for an error.	I
DTRowFontSize (integer blockNum, string DTName)	Returns the size used for the font within the data table. This will change if the height is modified either within the structure of the block or via the DTRowHeightSet function.	R
DTRowHeightSet (integer blockNum, string DTName, integer height)	Sets the height of the rows for a data table object. This function overrides the row height value set in the structure for a data table. All the rows of the data table will be set to the same height. Returns a zero for success, or a non zero value for an error.	I
DTToolTipSet(string captionString)	In conjunction with the DataTableHover message handler discussed on page 199, this function allows you to show custom tool tips when the cursor hovers over a data table.	V

Functions

Block data tables	Description	Return
DynamicDataTable(integer blockNumber, string dataTableName, array dynamicArrayName)	This function attaches a dynamic array to a dialog data table. This allows dynamically resizable data tables, and the ability to change what data is displayed in a data table without recopying the data. You can attach the array to the data table 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
DynamicDataTable2(integer blockNumber, string dataTableName, string arrayName)	Has the same behavior as the dynamicDataTable function, with the exception that it can be called from an outside block, and doesn't have to be called from within the block that contains the array. Note that the arrayName argument is the name of the array as a string, not the array name itself.	I
DynamicDataTableVariableColumns(integer blockNumber, string dataTableName, array y, integer rows, integer columns)	Similar to DynamicDatatable, with the addition of adding the specification of how many rows and columns the resulting data table will have. It will correct the data to work with a new number of columns. You must use the variable name of the data table, not the dynamic array, to access the data.	I
DynamicDataTableVariableColumns2 (integer blockNumber, string dataTableName, string arrayName, integer rows, integer columns)	Has the same behavior as the dynamicDataTableVariableColumns function, with the exception that it can be called from an outside block, and doesn't have to be called from within the block that contains the array. Note that the arrayName argument is the name of the array as a string, not the array name itself.	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, a -4 if no selection was made or if the block's dialog is closed, and a -1 if an error of some other type occurred (usually an invalid *DataTableName*).	I
GetDTOffset (integer blockNumber, string DTname, integer clone, integer which)	Returns the offset (how many rows or columns the table has been scrolled) for a data table. If clone is true, it returns the value for the first clone of the item found. For which, it returns: 0: irst column 1: first row 2: last column 3: last row	I

Functions

Block data tables	Description	Return
RefreshDatatable-Cells(integer blockNumber, string dataTable-Name, integer star-tRow, 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
ResizeDTDur-ingRead(string name, integer old-Rows, integer old-Cols)	This function will allow you to inform ExtendSim that a data table is now a different size than 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 ExtendSim when a Data Table has been redefined. (See GetFileReadVersion for more information.) *OldRows* and *Old-Cols* will contain the original size of the data table.	I
ScrollDTTo(inte-ger blockNumber, string name, integer row, integer col)	Scrolls the named data table to the indicated *row* and *column*. (0 successful, 1 failed)	I
SetDataTable-CornerLabel (string datatablename, string15 label)	Sets a label to be displayed in the upper left-hand corner of the data table. 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
SetDataTableLa-bels (string DataT-ableName, string LabelString)	The *LabelString* variable will be used as a replacement string for the column header string of the specified data table. The format for this string is the same as that used in the dialog editor when creating or modifying a data table 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 (Item 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.	V

Block data tables	Description	Return
SetDTColumnWidth(integer blockNumber, string name, integer column, integer width, integer doClones)	Sets the specified column in the named data table to the specified width. Please note that column number for this function is zero based starting from the title column, to allow reference to the title column. This means that the first column with data in it is column one, unlike most other functions that reference data table columns.	I
SetDTRowStart (integer blockNumber, string name, integer rowStart)	Sets the named data table 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
SortArrayVariableColumns(short hNum, array datatableName, integer numCols, integer numRowsToSort, integer keyColumn, integer increase, integer sortStringAsNumbers)	Sorts the specified array, using the numCols argument to define how many columns the array to be sorted contains.	I
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.	V
WhichDTCell(integer rowCol)	Returns the Row or Column that is currently active. If rowCol is TRUE this function returns the Column, otherwise it returns the row. This function can be called in the CELLACCEPT, DIALOGITEMTOOLTIP, DIALOGCLICK, or dialog item name message handlers.	I

Dynamic linking

Dynamic linking creates a link between a dialog item (data table or parameter) and a data source (ExtendSim database or global array). This is a very powerful feature because the links are live and dynamic—they update instantly when there is a change in the data source.

The ExtendSim user interface allows the modeler to create dynamic links just by right-clicking on a parameter or clicking the Link button at the lower left of a data table. In addition, ExtendSim provides functions, described below, that allow control over how the links are to be created and controlled.

For database linking using scripting see the DBDatatable function on page 338 and the DBParameter function on page 340. For global array linking using scripting, see the GADatatable function on page 344 and the GAParameter function on page 347. Also see “Database functions” on page 318 and “Global arrays” on page 342.

To simply register a block so that it will be notified if there was a database change, see “Registered blocks” on page 113 and “Linking and notification” on page 337.

☞ Starting in ExtendSim 7, you must access all data tables using their dialog item variable name. Even dynamic data tables need to be accessed using the dialog variable name. This was not the case in Extend 6 or earlier, but is a ModL compiler change that is necessary to allow data tables to transparently access linked database or global array data, and to be variable column. Using the dynamic array name will not allow linked or variable column data tables to work.

Dynamic linking	Description	Return
DBDatatable(integer blockNumber, string datatableName, integer databaseIndex, integer tableIndex, integer showFieldNames)	Link a data table with an ExtendSim database table. NOTE: This is the same function as listed on "Linking and notification" on page 337; we just copied it here for your convenience DK.	I
DBParameter(integer blockNumber, string dialogItemName, integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	Link a parameter with an ExtendSim database cell. NOTE: This is the same function as listed on "Linking and notification" on page 337; we just copied it here for your convenience DK.	I
DILinkClear(integer blockNumber, string dialogItemName)	Clears a link from the specified link if it has one.	I
DILinkInfo(integer blockN, string dialogItemName, integer which)	Returns linking information about the specified dialog item. Values for *which*: 0 : Link type returns 0: no link, 1: global array, 2: Database, 3: dynamic array 1: DB Index returns the index of the database for the link (if it's a DB link) 2: table/array index returns the value of the index 3: returns column index 4: returns row index 5: user link: if the link was created by a user vs. a function 7: returns dialogItemID to identify a dialog item. See DIGetID function description. 10: ReadOnly: return value is true/false	I
DILinkingDisabled(integer blockN, string dialogItemName, integer disableIfTrue)	This function controls if the specified data table or parameter will allow linking. By default each data table has a link button in its lower left-hand corner and calling this function will allow the block developer to hide this button. This should have the added effect of disabling the link menu command. Return a negative error number or zero if no error.	I

Dynamic linking	Description	Return
DILinkModify(integer blockNumber, string dialogItem, integer which, integer value)	This function is used to modify a linked dialog item's Link dialog flags. The flags control whether the block receives messages when the values of the linked data change. Messages are sent to the LinkContent message handler and the dialog's message handler. The *which* argument specifies which flag will be set; *value* specifies what the value of the flag should be set to. *which*: 0: ReadOnly. Set *value* to TRUE to make a link 'read-only' 1: InitMsgs. Set *value* TRUE so link will get messages during INITSIM. 2: SimMsgs. Set *value* TRUE so link will get messages during SIMULATE. 3: FinalMsgs. Set *value* TRUE so link will get messages during FINALCALC. 4: AnyMsgs. This is only available by code; it has no corresponding checkbox in the Link dialog. By default this is set to True, so you get LinkContent messages when the linked data changes. If set to False, the block that is being linked to will not get any messages at all. Caution: the setting True/False is saved with the model so use this call carefully as the application will not reset to True each time the model is opened. 5: UserLink. Set *value* TRUE so link was created by the user interface (TRUE), or a MODL function call (FALSE). 11: ShowFieldNames Set *value* TRUE so link uses the field names for the header rows in the data table.	I
DILinkMsgs(integer sendMsgs)	Turns on and off the sending of messages associated with linking. This function should be used carefully, and should always be turned back on when the code has completed. This will disable all messages sent to *any* blocks that have dialog items linked to *any* data source, and many blocks will function incorrectly if these messages are disabled. The primary reason to use this function would be if you are making many changes to a data source, and don't want things to be overwhelmed with too many messages.	V
DILink-SendMsgs(integer dataSourceType, integer DBIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	Sends update messages to all the dialog items linked to this data source. In the Global Array case, the DBIndex is ignored. If you want to send messages to all dialog items linked to a whole table, pass in –1 for the fieldIndex, and –1 for the rowIndex. dataSourceType: 1: Global Array 2: Database	I

Dynamic linking	Description	Return
DILinkUpdateInfo(integer which)	Should be called in the LinkStructure, LinkContent, or dialog item name message handlers. Returns information about which link changed and what the change was. Values for *which*: 0:Link type (returns 0:no link, 1:global array, 2:database, 3: dynamic array) 1:DB Index returns the index of the database for the link (if it's a DB link) 2:table/array index returns the value of the index 3:returns column index 4:returns row index 5:what changed (see below for values) 6:number of rows or columns changed 7:dialog item ID 8: the block number of the block that changed the data 9: Returns True (1) if link messages are enabled; otherwise returns False (0). 10:returns the index of the database being imported or 0 if no database is being imported. A *which* value of 5 (what changed) returns 1:data changed, 2:field inserted, 3:field deleted, 4:field renamed, 5:record inserted, 6:record deleted, 7:table or GA deleted, 8:table or GA renamed, 9:DB deleted, 10:DB renamed, 11:link created, 12:link modified, 13:link cleared, 14:DB replaced via DBDatabaseImport(), 15:GA resized, 16:table inserted, 17:field properties modified, 18:field moved, 19:table sorted, 20:Table Properties modified, 21:record ID modified.	I
DILinkUpdateString(integer which)	This companion function to DILinkUpdateInfo returns a string containing information about a change to the link status. This can be called in the same message handlers as DILinkUpdateInfo. which values: 0 : DialogItemName The name of the dialog item associated with the link. 1 : Changed Name. If whatChanged is field or table renamed, for example, the changed name string will be the old name of the field or table.	S
DTHideLinkButton(integer blockN, string DTname, integer hideButton)	This function is still available however, there is one that is more complete. See DILinkingDisabled(), above, that works for both data tables and parameters. See that function for a description.	I

Dynamic text items

Dynamic Text items are a new type of dialog item that supports up to 32000 characters of text. They are useful where the 255 character limitation of editable text items is a hindrance. The text is stored in a string dynamic array, declared:

```
string dynTextArray[];
```

For an example of how to use dynamic text items, look at the Equation block (Value library).

Dynamic Text	Description	Return
DynamicTextArrayNumber (string dynamicTextArray[])	Returns the dynamic array index. Used to link the Dynamic Text Item to the correct dynamic array. For example: `myDTextItem = DynamicTextArrayNumber(dynTextArray);` where dynTextArray is declared as above.	I
DynamicTextIsDirty (integer blockNumber, string name)	Returns a true if the text item is dirty (the user has typed text into the dynamic text item or changed its contents in some way).	I
DynamicTextSetDirty (integer blockNumber, string variableNameStr, integer dirty)	Sets the dirty flag to 'dirty' (TRUE or FALSE) on the specified dynamic text item. Useful to set dirty if the block performs a text changing operation outside of the user editing the text.	I
StrFindDynamic (string dynamicTextArray[], string findStr, integer caseSens, integer diacSens, integer wholeWords)	Similar to the StrFind() function, except it searches a dynamic text array.	I
StrFindDynamicStartPoint (string dynamicTextArray[], string findStr, integer caseSens, integer diacSens, integer wholeWords, integer startPoint)	Similar to the StrFind() function, except it searches a dynamic text array. startPoint is the starting character index to search from, starting at zero.	I
StrReplaceDynamic (string dynamicTextArray[], integer start, integer numChars, string replaceStr)	Similar to the StrReplace function, except it replaces text in a dynamic text array.	

Formatting/interactivity using column and parameter tags

Column tag and Parameter tag functions provide a great amount of control over the display and formatting of certain dialog items. Column and parameter tags allow you to put strings, checkboxes, buttons, dates, popup menus, the infinity character, and so forth into a parameter field or the cell of a data table. They can also be used to hide or disable a column or parameter.

The data table and parameter dialog items check which column tags have been set, and draw their data, or respond to clicks appropriately.

☞ The Data Init block (Value library) is an example of using column tags.

The basic mechanism for setting a param tag, or column tag, is to call either the DIParamTagSet or the DTColumnTagSet function. Column and param tags are not saved when the model is closed, so the tags are commonly reset in the OpenDialog and CloneInit message handlers.

If you set a column tag that has a value higher than 99, it will be stored as a HeaderTag. HeaderTags are stored independently from column tags, and will store a behavior for the header for that column, not for the individual cells in the table. (See Header tags below for more information.)

In the data table case, column values start at zero, with zero referring to the first column after the row column. Each column has four pieces of information that can be associated with it: Two integer values, the tag itself, a tagOption value, and two string values, the TagString1 and TagString2. ParamTags for parameters have just one number, the paramTag, and the same two strings.

The primary tag sets the default behavior for the column/parameter. The TagOption value is used for specific purposes by some of the tags below, and can also be used to disable the column using the TAGDISABLE value.

The TagString1 value is the lookup name in the case of string Lookup table columns, for other tags it's used as needed. TagString2 is used by some tags. For any tag that displays a string, except the SL tags and the DBFIELD tag, putting a string into the string1, or string2 field will prepend the string from string1, and/or append the string from string2.

The column/parameter tags are listed below. The symbol name definitions can be found in the include file "\Extensions\Includes\ColumnTags v10.h".

Tag name	Value	Description
Tag_Block_Label	60	Tags the column/parameter as containing block labels.
Tag_Block_Label_Name	62	If the block has a block label it will show that, but if there is no block label, shows the block name (e.g. Activity)
Tag_Block_Name	61	Tags the column/parameter as containing block names.
Tag_Button	21	Tags the column/parameter as containing push buttons. To hide the button in a specific cell, set that data table cell to BLANK.
Tag_CheckBox	20	Tags the column/parameter as containing checkboxes. BLANK is equivalent to FALSE (unchecked).
Tag_Color	24	Tags the column as containing colored cells. The value in the row of the column will be used as an eColor value to set the color of the each cell. (Alpha channel will be ignored and set to 255.)

Tag name	Value	Description
Tag_Conditional_Popup	42	Tags the column as containing popups conditionally. This functionality is very specific. Each cell/row in the column will show a popup triangle under the following condition: The cell in the column to the left of the column specified must contain a string value that matches the stringTag that was passed in when the coltag was set. If the string value in the cell to the left of a given cell does not contain the string value, that cell will default to a normal editable cell. This functionality is used in the Item Equation block.
Tag_Conditional_Popup_Array	43	Tags the column as containing popups conditionally. This functionality is very specific. Each cell/row in the column will show a popup triangle under the following condition: The cell in the dynamic array specified by the tagOption parameter must contain a string value that matches the stringTag that was passed in when the coltag was set. If the string value in the dynamic array cell does not match the string value, that cell will default to a normal editable cell. This functionality is used in the Item Equation block. This functionality is the same as the TAG_Conditional_Popup function except that the comparison strings are kept in a dynamic Array instead of in the adjacent column.
Tag_Date	50	Tags the column/parameter as containing date values.
Tag_Date_Convert	51	Tags the column as containing simulation time values that should be represented as date values. The conversion is done in the application.
Tag_DB_Address	32	Tags the column as containing DB Addresses, but with no popup menu so the block designer can customize the DB part selection process.
Tag_DB_Address_Pop	31	Tags the column as containing DB Addresses. A popup menu appears when the user clicks the data table cell, enabling selection of the DB parts.
Tag_DB_Field	30	Tags the column as containing data from a specified database table field. TagOption should contain the DBIndex, String1, the tablename, and string2 the fieldname.
Tag_Disable	99	Tags the column/parameter as being disabled. Note that this tag can be used in Tag Option to mark a tagged column as disabled as well as the other meaning of the tag in the column tag.
Tag_Head_Tag_Popup	100	Tags the column/parameter header as being a popup menu. Note: this does not change the behavior of clicking on the cell, just the appearance of the cell. The popup behavior needs to be implemented in block code.
Tag_Hide	98	Tags the column/parameter as being hidden.

Tag name	Value	Description
Tag_Infinity	26	Tags the parameter as containing a checkbox that changes the value to a novalue to represent infinity. If ParamStr and paramStr2 are empty, the behavior is that the dialog item will display the word infinite, and an infinity symbol will be drawn near the checkbox. If there is a string in paramStr, it will be displayed instead of the word infinity, if there is a string in paramStr2, it toggles whether or not the infinity symbol is drawn, so if you want to change the meaning of novalue to something else, just put the meaning in paramStr, and an ‘x’. or some thing like that in paramStr2.
Tag_No_-Head_Tag	101	Reverts the header tag back to zero, meaning no defined behavior.
Tag_Nothing	0	Tags the column/Parameter as not having a tag. (Note that you can use this with the TagOption or the TagStrings to disable the column, or append/prepend text to the column.)
Tag_Number	71	Used in a string table to format numbers correctly to fit the cell.
Tag_Percent	70	Tags the column/parameter as containing percents. Only numerical parameters support this tag.
Tag_Popup	40	Tags the column/parameter as containing popupMenus. Note: this does not change the behavior of clicking on the cells, just the appearance of the cells. The popup behavior needs to be implemented in block code.
Tag_Precision	72	Specifies the precision of the numbers displayed in the column. The coltag2 value specified will determine how many characters of precision with which to display the numbers in the cells.
Tag_Progress	23	Tags the column as containing progress bars. The value of the progress bar will show using the content of the row to display a range from 0 to 100. If column tag 2 contains a valid eColor value, that color will be used to render the progress bars.
Tag_Radio	22	Tags the column/parameter as containing radio buttons.
Tag_SL	10	Tags the column/parameter as containing string lookup values.
Tag_SL_Append	17	Tags the column as containing string lookup values, appended with the string in tagSstring2.
Tag_SL_Pop	11	Tags the column/parameter as containing string lookup values that function as popup menus when clicked.
Tag_SL_Prepend	16	Tags the column as containing string lookup values, prepended with the string in tagString2.
Tag_Str_Flag	80	Tags the column as a numeric column where the colstring will be displayed when the value in a cell is a negative one. Otherwise, the column will be treated as a standard numeric data column.
Tag_Str_Flag_-Noed	81	The same as Tag_Str_Flag, except it doesn’t allow editing.

Column/parameter tag functions

Here is a list of the functions used to implement column/parameter tags. The tag values are listed in the table above:

Column/parameter tags	Description	Return
DIParamTagGet(integer blockNumber, string dialogItemName)	Returns the ParamTag for the specified parameter.	I
DIParamTagSet(integer blockNumber, string dialogItemName, integer tag, integer tag2, string tagString, string tagString2)	Sets the specified Parameter tag to the specified value. Values for the tag value are from the ColTag/ParamTag Values table above. The tagString value is currently only used for the string lookup tags, in which case it is the lookup name.	I
DIParamTagStringGet(integer blockNumber, string dialogItemName, integer which)	Returns a TagString for the specified parameter. *Which* should be 0 for the first string and 1 for the second string.	S
DTColumnTagGet(integer blockNumber, string DTname, integer col, integer which)	Returns the column tag for the specified column. Which value is: 0: columnTag 1 1: columnTag 2 2: the header tag	I
DTColumnTagSet(integer blockNumber, string DTname, integer col, integer tag, integer tagOption, string tagString, string tagString2)	Sets the specified column tag to the specified value. Col values start at zero, with zero referring to the first column after the row column. Values for the tag value are from the ColTag/ParamTag Values table above. The tagString value is the lookup table name in the case of string Lookup table columns, for other tags it's used as needed. For many types of tags it's not used. TagString2 is used by some tags as well, for many it's not used, and can just be set to “” the empty string. TagOption specifies an option for the tag. Some tags use this option for specific purposes. Using the flag for TAGDISABLE in this argument will disable most column tags. For any tag that displays a string, except the SL tags and the DBFIELD tag, putting a string into the string1, or string2 field will prepend the string from string1, and/or append the string from string2.	I
DTColumnTagStringGet(integer blockNumber, string DTname, integer col, integer which)	Returns a tagString value for the specified column. *Which* should be 0 for the first string and 1 for the second string.	S

Dialog item tool tips

The System message DialogItemToolTip is sent to the block when the cursor hovers over a dialog item. In that message handler, the dialog item tool tip functions (below) will allow you to customize and accesss the tooltip that will appear.

The functions WhichDialogItem(), and whichDTCell() can be used with this functionality to find which dialog item the cursor is currently hovering over.

Dialog item tool tips	Description	Return
DIToolTipSet(string-string, integer replace)	Allows you to set the string that will be displayed. If the replace flag is true, the default tool tip string will be replaced, if it is false, the default string will be appended to.	V

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	Return
BlockDialogIsOpen(integer blockNumber)	Returns TRUE if the block dialog it open. Returns FALSE otherwise.	I
CloseBlockDialogBox(i)	Closes the dialog for any block, where *i* is the global block number. See OpenBlockDialogBox, below.	V
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. NOTE: to prevent the dialog from closing conditionally, use an Abort statement in the DIALOGCLOSE message handler.	V
CloseEnclosingHblock()	Closes the hierarchical block in which this block resides, if any.	V
CloseEnclosingHblock2(integer blockNumber)	This is same as CloseEnclosingHblock(), except it refers to a global block number.	V
DialogGetSize(integer theBlockNumber, integer which)	Gets the width or Height of an open dialog. (Returns the width or height of the Hblock model window if the block number refers to an Hblock.) The 'which' argument takes the following values: 0: width 1: height.	I
DialogMoveTo(integer blockNumber, integer x, integer y)	Moves the dialog box to the specified x and y pixel location. If the block is an Hblock, this will move the Hblock submodel window.	I
DialogSetSize(integer blockNumber, integer w, integer h)	Resizes the dialog box of the block. Only works if the dialog is open. If the block is an Hblock, this will resize the Hblock submodel window. W and H are width and height in pixels.	I
MakeDialogModal (integer theBlockNumber, 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.	V

Block dialogs	Description	Return
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.	V
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.	V
OpenEnclosing-Hblock()	Opens the hierarchical block in which this block resides, if any.	V
OpenEnclosingH-block2 (integer blockNumber)	This is same as OpenEnclosingHblock(), except it refers to a global block number that is within the Hblock.	V

Block dialog tabs

These functions allow ModL code to manipulate block dialog tabs.

Block dialog tabs	Description	Return
DisableTabName (string tabName, integer trueOr-False)	Disables or enables the specified tab.	V
GetCurrentTab-Name(integer blockNumber)	Returns the currently selected tab name.	S
GetBlockTab-Names(integer blockNum, Str63 names[])	Returns the number of dialog tabs in the specified block and lists the names of the tabs in the Str63 dynamic array names.	I
OpenDialogBox-ToTabName(string tabName, integer blockNumber)	Opens a dialog box to a specified tab.	V
SetDefaultTab-Name (string tab-Name, integer blockNumber)	Sets the block so that the dialog will open at a specific tab name.	V
VariableNameTo-TabName(string varName, integer blockNumber)	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 ExtendSim's simulation engine (i.e. Item 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 153.

In the functions, *connName* is the name of the connector in the calling block and *block* is the global block number of the receiving block. (See "Block numbers, labels, names, categories, position" on page 256 .)

Block messaging	Description	Return
BlockSimStartPriority(integer blockNumber, integer priority)	Sets the *priority* value for receiving the SimStart message discussed on "Simulation messages" on page 194. Returns 0 for success or a negative value for failure. The default value is -1 and any block with a *priority* value of less than 0 will not receive the message. Blocks that have a *priority* value greater than -1 will receive the message in priority order (lowest to highest).	I
BlockSimFinishPriority(integer blockNumber, integer priority)	Sets the *priority* value for receiving the SimFinish message discussed on "Simulation messages" on page 194. Returns 0 for success or a negative value for failure. The default value is -1 and any block with a priority value of less than 0 will not receive the message. Blocks that have a *priority* value greater than -1 will receive the message in priority order (lowest to highest).	I
ConnectorMsgBreak()	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 and it should be called right before returning from the current message handler.	V
DuringHblockUpdate()	During an Hblock update, some connections are temporarily broken and then reconnected. Returns True if called during an Hblock update so you can determine if ConnectionMake or ConnectionBreak messages can be safely ignored.	I
GetConnectorMsgsFirst (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.	V
GetMsgSendingBlock ()	Returns the blocknumber of the block that sends the currently executing message to the current block.	I

Block messaging	Description	Return
GetSimulateMsgs(integer ifTrue)	In some cases, you may not want a block to get Simulate messages. (See “Residence blocks that do not post future events:” on page 149.) 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.	V
MsgEmulationOptimize(integer ifTrue)	See “Value connector messages” on page 158. This specialized function is used in the Executive block (Item library) to optimize the operation of Value blocks in discrete event models. It prevents the propagation of redundant messages that are produced by Value blocks emulating and propagating connector messages from Item 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 Itemand Value 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.	V
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 Executive block (Item library), as sending connector messages at the wrong times can cause problems in these libraries.	V
SendConnectorMsgToBlock (integer blockNumber, integer conn)	Sends a connector message to a block number that is not necessarily connected to the block sending the message. *Conn* is the index for the receiving connector in the connector pane (the index starts at 0).	V
SendMsgToAllCons(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.	V
SendMsgToBlock (integer block, integer messageConstant)	Sends a message to the specified global block. The *message* is an ExtendSim constant that corresponds to the message to be sent. The messageConstant is not a string. It is derived by taking the message name and adding MSG after the message name. For example, to send a USERMSG0 to a block, messageConstant would be USERMSG0MSG (not a string). See the chapter “Messages and Message Handlers” on page 193 for all the messages that can be sent. 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.	V

Block messaging	Description	Return
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.	V
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.	V
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.	V
SimulateConnectorMsgs(integer trueFalse)	This is used to disable the “simulation” of connector messages by Value blocks that have no connector message handlers, as described in “Value connector messages” on page 158. “Simulation” of connector messages is enabled by default for new and existing models, and is always re-enabled 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 Item library. Use this function in INITSIM or CHECKDATA.	V

Icon views

These functions are used to interrogate and control block icon classes and views. Classes are global (e.g. Flowchart class) and always set by the modeler, but views (e.g. Flow down) can be set by the modeler or the block.

The default class (ExtendSim classic) and view are both zero.

☞ As of ExtendSim 10, icon classes were eliminated.

Classes and views	Description	Return
IconGetClass (integer blockNumber)	Gets the current class (0 to 7) of the block, but classes are currently global, so the entire model will have this class.	I
IconGetView (integer blockNumber)	Gets the current view (0 to n) of the block. See IconGetViewName(), below.	I
IconGetViewName (integer blockNumber, integer view)	Returns the name of the view specified.	S
IconSetViewByIndex (integer blockNumber, integer index)	Sets the block’s view by index (0 to n).	V

Classes and views	Description	Return
IconSetViewByPartialName (integer blockNumber, string partialName)	Sets the block's view by the partial name entered. For example, "right" will set the icon view to "Right Angle." This is useful when the view names are slightly different for different classes and you want to set the view to a type of view rather tban a specific one.	I

Models, notebooks, and libraries

Models, notebook	Description	Return
DuringContinue()	Returns true if the model is being continued from a previous saved run.	I
GetBlockInfo(integer theBlockNumber, integer which)	Returns misc information about a block which can take the following values: 1: invisible – returns if the block is invisible. 2: scriptedBlock – returns if the block was created via scripting. 3: dialog box is open 4: block is in debugging mode	R
GetGlobalSimulationOrder(integer blockNumber)	Returns the actual simulation order index of a block.	I
GetLibraryContents (string libName, Str31 blockNames[], Str31 blockTypes[])	Fills the passed-in Dynamic Arrays with the names and types of the blocks in the named library. Returns the number of blocks in the library.	I
GetLibraryInfo (integer blockNumber, integer specify)	Returns 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 blockNumber, 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
GetLibraryStringInfo (integer blockNumber, integer specify)	Returns string information about the library that the block comes from. Specify takes the following values: 1: Name (others will be defined in the future)	S

Models, notebook	Description	Return
GetLibraryVersion(integer blockNumber)	This function returns the short version string for the library that includes the block specified by blockNumber.	S
GetLibraryVersionByName(string libName)	Returns the library version string.	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
GetModelSimulationOrder()	Returns the simulation order of the block that called this function.	I
GetRunParameter(integer which)	Similar to GetRunParameterLong except it returns a real. *Which* specifies a parameter in the Simulation Setup dialog. **In addition to the parameters for GetRunParameterLong**, this function returns values for: 1 - endTime 2 - startTime 10 - startDate 12 - endDate	R
GetRunParameterLong(integer which)	Gets the specified parameter from the Simulation Setup dialog using the value of *which*: 3 - numSims 4 - numSteps 5 - random seed 6 - seedControl (value, from 1-3, of the seed popup from the Random Numbers tab) 7 - checkRandomSeeds (value of the duplicate seeds check box from the Random Numbers tab) 8 - timeUnits 9 - calendarDates 11 - "__seed" table database index	I
GetSerialNumber()	Returns the serial number of the unit as a string.	S

Models, notebook	Description	Return
GetSimulation-Phase()	Returns the phase of the simulation: -1: No model open 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 9: PreCheckData 10: PostInitSim 11: SimStart 12: SimFinish 13: ModifyRunParameter 14: OpenModelPhase Note: If a hierarchical block is updated, ExtendSim sends an openModel message handler but this function will return a 0 rather than a 14.	I
GetWindowsHndl (integer which)	(Windows Only) Returns the Windows API handle of the main window in the ExtendSim 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
IsSimulation-Paused()	Returns TRUE if the simulation run is paused or is about to pause.	I
Librarie-sOpen(String-DynamicArrayName)	Returns the list of library names in the String dynamic array. Also returns the number of libraries that are open.	I
LibraryUsed(integer blockNum)	Returns the name of the library that owns blockNum.	S
ModelLock(integer lock, integer HblocksLocked, string password)	The function allows you to lock and unlock the model from MODL code. The Lock argument takes a value of one if the model is to be locked, a value of zero for unlocking. Hblocks-Locked is a flag that sets whether you want the Hblocks to be locked when locking a model. The password is the password to be set in the locking case, and the password to be compared with in the unlocking case. This function returns a zero if is succeeds, and the following error codes for specific error conditions: -1: Unlock password didn't match. -2: Model is already locked, it cannot be locked again without being unlocked first. -3: lock value out of range.	I
NotebookClose()	Closes the notebook if it is open.	I

Models, notebook	Description	Return
NotebookItem-Rect(blockNum, whichNotebook, iobjectID, rectArray)	Fills the 4 element array rectArray with the position values for the specified notebook item. Specification of whichNotebook works the same way as for OpenNotebook2. Coordinates are: 0: top 1: left 2: bottom 3: right Return value is the type of the object.	I
Notebook-Items(blockNum, whichNotebook, idArray, typeArray)	Fills the idArray and typeArray with information about objects in the specified notebook. IdArray is filled with a list of the id numbers of objects in the notebook. typeArray is filled with type specifiers for those same items. Specification of whichNotebook works the same way as for OpenNotebook2. The return value is the count of the number of items found (therefore also the number of rows filled in the arrays.)	I
Notebook-ItemInfo(block-Num, whichNotebook, itemId, which)	Returns information about the specified notebook object. Specification of whichNotebook works the same way as for OpenNotebook2. Which values are: 0: type 1: isClone 2: blockNumber 3: font size in pixels 4: border thickness	R
NotebookItemInfoString(block-Num, whichNotebook, itemId, which)	Returns string information about the specified notebook object.Specification of whichNotebook works the same way as for OpenNotebook2. Which values are: 1: object name 2: border color as HTML value string 3: fill color as HTML value string	S
NotebookIsOpen()	Returns a true value if the notebook is currently open.	I
Notebook-sInfo(integer parentHblock[], integer notebookIndex[], String notebookName[], integer notebook-Publish[])	Fills the dynamic arrays with information about all the notebooks in the model, including HBlock notebooks. For a model's notebook, parentHblock is -1. For an Hblock's notebook, parentHBlock is the number of the Hblock. The notebookIndex is the number of that notebook for that block or model, starting from 1. The notebookName is the name of that notebook. NotebookPublish is true if that notebook is marked by the user to be published. The results of this function call can be used as arguments for the NotebookItems() functions to see items in a particular notebook. Returns the integer number of notebooks in the model.	I

Models, notebook	Description	Return
OpenNotebook()	Opens the model notebook. Since this is from v9, which only had one notebook per model, see also OpenNotebook2.	V
OpenNotebook2(integer blockNumber, integer Index, integer ripOff)	Opens the specified notebook, either for the model worksheet or for an Hblock. BlockNumber controls which set of Hblock notebooks is referenced by the Index. Pass in the number of a block at the top level of the main worksheet or a -1 for blockNumber to get to the top level notebooks. Otherwise, the blockNumber will be used to identify which Hblock's set of notebooks will be referenced by the Index. If the blockNumber is for a block inside an Hblock, that Hblock's notebooks will be referenced, otherwise the blockNumber will identify the hblock itself. If ripOff is true, the notebook will be opened in the ripped-off state. If false, the notebook's tab will be selected.	V
PauseSimForSave()	Pauses the simulation at the end of a time step (currentTime for discrete event models and currentStep for continuous process). When the function is called, the application completes sending the onSimulate message handlers for the current step; pausing is delayed until that current step is finished. The simulation remains paused until you chose Run > Resume, click the Resume button, or call ResumeSimulation. This is important for continuous process models, to ensure that all the blocks are time synced and have finished executing their final current step. Note: the separate PauseSim() function stops the execution instantly.	V
ResumeSimulation()	Resumes the simulation and returns immediately.	V
RunSetup(trueFalse)	Opens Setup Simulation dialog and sets up a 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 below. NOTE: If called from a block, runs the model until it completes or is aborted, and then returns. If you use the ExecuteMenuCommand() function to call RunSimulation() (e.g. from scripting or OLE automation), it returns immediately after starting the model run.	I
SaveModel()	Can save the model before or after a run. Cannot save the model during a simulation run.	V

Models, notebook	Description	Return
SaveModelAs(string fileName)	Saves the active model under the name and path defined in fileName. Returns a zero if successful, a negative number otherwise. Saves after the current block is executed. Can save before or after a simulation run; cannot save during a run.	I
SaveTopDocAs(string filePath, integer aSync)	Saves the top document under the file name and path name defined by *filePath*. If *aSync* is TRUE, saves after the current block is executed. Similar to SaveModelAs except it will work on whichever the top ExtendSim document is. (Specifically used by the Equation block to save and close Include files.) Returns 0 for success or a negative value to indicate failure. Note that if the top document is a model and it is running a simulation, it cannot be saved. The model can only be saved before or after running a simulation.	I
SetBlockSimulationOrder (integer blockNumber, integer 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 than zero. -3: no active model document. -4: sim order is not set to custom. -5: no such block.	I
SetModelSimulationOrder (integer 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
SetRunParameter(real paramValue, integer which)	Sets a single parameter in the Simulation Setup dialog. As an enumerated list, this function is more expandable than SetRunParameters(). The values for *which* are: 1:endTime, 2:startTime, 3:numSims, 4:numSteps, 5:random seed), 6:seedControl (value, from 1-3, for the seed popup from the Random Numbers tab), 7:checkRandomSeeds (value for the duplicate seeds check box for the Random Numbers tab), 8:timeUnits, 9:calendarDates, 10:startDate, 11: "__seed" table database index (Note: "seed is preceded by 2 underscore characters).	I

Models, notebook	Description	Return
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
SpinCursor()	Spins the beachball cursor. Used to alert the user that a time consuming calculation is taking place.	V
SpinCursorStart(integer showDuringRun)	Starts a spinCursor that will continue until spinCursorStop is called. showDuringRun determines if the cursor is shown during a simulation or not.	V
SpinCursorStop()	Stops a spinCursor started by spinCursorStart.	V

DE Modeling Using Equation Blocks

The following function are convenience functions intended to be called from Equation blocks. They have the effect of querying a resource pool block for the number of available resource, or reuesting that the Resourcd Pool block allocate a resource. They are implemented through the sending of messages, and the use of globals. See the code of the resource pool block for more information if desired.

DE Modeling	Description	Return
ResourcePoolAllocate(integer ResourcePoolBlockNum, real NumToAllocate)	Requests the specified Resource Pool block to allocate the specified number of resources.	
ResourcePoolAvailable(integer ResourcePoolBlockNumber)	Queries the specified resource pool block for the number of resources that are available.	

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” on page 267.

See the Scripting blocks in the ModL Tips library for examples of scripting.

Scripting	Description	Return
ActivateApplication()	Brings the ExtendSim application to the foreground. This is primarily used for interapplication/scripting	V
ActivateWorksheet (string sheetName)	Activates (selects and brings to the front) the named worksheet. NOTE: Normally you would expect Success to return a 0 but for backwards compatibility this function returns a 1 for Success or 0 for Failure.	I
AddBlockToClipboard (integer blockNumber)	Adds the indicated block to the clipboard contents without otherwise changing the contents. Returns FALSE if successful.	I
AddBlockToSelection (integer blockNumber)	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
ApplicationFrame(integer frame[], integer inside)	Returns the application frame in global coordinates. The *Inside* argument specifies if the returned frame should be inside (includes the menubar and window frame) of the application frame window, or the outside. Declare frame as: Integer frame[4]; When the function returns, frame will contain: frame [0] - Top, frame[1] - Left, frame[2] - Bottom, frame[3] - Right	I
BlockAdjustPosition(integer blockNum)	Adjusts the position of the specific block relative to the location of the icon positioner, which is turned on or off in the Icon tab of a block's structure. See "Icon positioner" on page 10 for more information.	V
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: 12: simulation sound 14: connection type (0=free, 1=right, 2= smart, 3=straight) 21: show worksheet tool tips 22: show dialog tool tips 27: metric units 30: show library warnings 31: check application version 32: bump connectors and auto-connect 33: show dialog editor tool tips Note: XCMDs and patterns are not supported as of release 10. See GetPreference(), below.	V

Scripting	Description	Return
ChangePreferenceString (integer preference, string theString)	Change the string-based preferences using this function. The preferences that can be changed with this function are: Default Library Path: 1 Library 1: 2 Library 2: 3 Library 3: 4 Library 4: 5 Library 5: 6 Library 6: 7 Library 7: 8 Default Model Path: 9	V
ClearBlock(integer blockNumber)	Clears the specified block from the worksheet.	V
ClearBlockUndo (integer theBlockNumber)	Clears the block and adds it to the delete task list to allow undoing. Returns FALSE if successful.	I
ClearConnection(integer blockFrom, integer conFrom, integer blockTo, integer conTo)	Removes the connection between the specified blocks and connector numbers. Returns TRUE if successful.	I
ClearUndo(void)	Clears the Undo list and removes any existing undo tasks from the Edit menu. After executing this function, the Undo command will be disabled.	V
CloneCreate(integer blockNumber, string variableName, integer destination, integer left, integer top)	Returns a cloneID for the new clone, used in the other clone scripting functions. VariableName is the dialog item variable name. Destination is -1 for the top worksheet, -2 for the notebook or the block number of an H-block to put it into its submodel. Left and Top are the pixel coordinates of the left-top corner of the clone.	I
CloneDelete(integer cloneID)	Deletes the specified clone. Get the cloneID from the CloneCreate() or CloneFind() functions.	I
CloneFind(integer blockNumber, string variableName, integer n)	Returns a cloneID for the found clone, used in the other clone scripting functions. VariableName is the dialog item variable name. N is the nth instance of the clone specified, in order of creation. If variableName is blank, the nth clone from the block is returned.	I
CloneGetDialogItem(integer cloneID)	Returns the variable name of the dialog item that the specified clone is from.	S
CloneGetDialogItemLabel(integer cloneID, integer n)	Returns the Nth label on a cloned dialog item that has labels. This would be either a Popup menu, or a data table.	S

Scripting	Description	Return
CloneGetInfo(integer cloneID, integer whatInfo)	For whatInfo, 1:returns the type of the clone, 2:returns the block number of the clone, 3:returns True if the clone is selected and False if it is not. Values for the type of clone (whatInfo:1) are: 1:Button, 2:Check Box, 3:Radio Button, 4:Meter, 5:Parameter, 6:Slider, 7:data table, 8:EditText, 9:StaticText, 12:Switch, 13:String Table, 14:Plotter pane, 15:Plotter data table, 16:Popup Menu, 17:EmbeddedObject (obsolete as of ES10), 18:DynamicText, 19:Text frame, 20:Calendar, 21:EditText(31).	I
CloneGetList(integer blockNumber, string variableName, integer cloneIDArray[])	Returns the number of clones in the cloneIDArray array. Fills the dynamic array cloneIDArray with all of the cloneIDs for the variableName. If variableName is blank, all of the clone IDs for that block are returned in the array.	I
CloneGetPosition(integer cloneID, integer positionArray)	PositionArray is declared as an array of 4 integers. It returns the following information: positionArray[0] = cloneRect.top, positionArray[1] = cloneRect.left, positionArray[2] = cloneRect.bottom, positionArray[3] = cloneRect.right.	I
CloneHideDisable(integer cloneID, integer disable, integer disableIFtrue)	Disables/enables or hides/shows the specified clone. If *disable* is True and *disableIFtrue* is True, the clone is disabled. If *disable* is True and *disableIFtrue* is False, the clone is enabled. If *disable* is False and *disableIFtrue* is True, the function hides the clone; the clone appears if *disable* is False and *disableIFtrue* is False. Returns zero if successful; otherwise returns an error code (negative number).	I
CloneResize(integer cloneID, integer top, integer left, integer bottom, integer right)	Resizes the clone to the specified rectangle (position and size).	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.	V
CreateHblock (string theName)	Makes the current selection into an H-block with the specified name.	I
DialogRefresh(integer blockNumber)	Forces the dialog box of the specified block to refresh (redraw.) Does nothing if the dialog box is not open or the block doesn't exist. Returns true (1) if it succeeds, zero otherwise.	I
DialogFixedSize(integer blockNumber, integer height, integer width)	Specifies that the dialog box for *blockNumber* should be fixed to the specified height and width. User resizing of the block dialog will be restricted.	I

Scripting	Description	Return
DuplicateBlock (integer theBlock-Number)	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. ExtendSim 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 ensure that the correct model is in the active window. See Appendix C for a list of menu command numbers.	V
ExtendMaximize()	Maximizes the ExtendSim Application.	V
ExtendMinimize()	Minimizes the ExtendSim Application.	V
FindBlock (string searchStr, integer which, integer openDialogs, integer wholeWords, integer justBlock-Num)	Finds the first block that matches the string *searchStr* and returns its 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 justBlock-Num 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 ExtendSim application file. This function is useful for determining the location of files for which only the file's relative location to the ExtendSim application is known.	S

Scripting	Description	Return
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: 12: simulation sound 14: connection type (0=free, 1=right, 2= smart, 3=straight) 21: show worksheet tool tips 22: show dialog tool tips 27: metric units 30: show library warnings 31: check application version 32: bump connectors and auto-connect 33: show dialog editor tool tips See also ChangePreference Note: XCMDs and patterns are not supported as of release 10.	I
GetRecentFile-Path(integer which)	Returns the full path to the specified recent file from the list of 5 recent files at the bottom of the File menu. The specified file (*which*) will be a number from 1 to 5.	S
GetUserPath()	Returns the path to the user documents directory. Similar to GetAppPath.	S
GetWorksheet-Frame(integer blockNumber, 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
HBlockTopGet()	Returns the block number of the topmost open Hblock's sub-model, or a negative number if there are no Hblocks open *above* the model worksheet.	I
HBlockUnlink-FromLibrary(integer blockNumber)	Disconnects the specified Hblock from the library it has a link to. This will make the Hblock into a standalone Hblock without a library connection.	V
IsBlockSelected (integer blockNumber)	Returns a TRUE if the indicated block is selected, returns false otherwise. This function will return a –1 if the indicated block-number is not a valid block.	I
IsLibEnabled(integer type)	Returns TRUE if the type of ExtendSim product will allow a specific library to open. Type values are: 17: DE product (Item library), 18: Pro product. (Rate and Reliability libraries).	I
IsMenuIte-mOn(integer whichItem)	Returns TRUE if the given menu command is currently checked. The whichItem argument uses the same numbers as defined in Appendix C for the ExecuteMenuCommand() function.	I
IsMetric()	Returns TRUE if metric is set in the options dialog.	I

Scripting	Description	Return
LastBlockPlaced()	Returns the block number of the last block placed on the active worksheet.	I
LibraryGetInfoBy-Name(unsigned string libName, string blockName, integer specify)	Function to return information about a library, and blocks in that library. Specify takes the following values; 1: Hblock - TRUE/FALSE	I
MakeBlockInvisi-ble (integer global-BlockNum, 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 block-From, integer con-From, integer blockTo, integer conTo)	Makes a connection line from the From block to the To block.	I
ModelSettings-Get(string model-Name, integer which)	Returns the model setting value specified by the *which* argument. If modelName is an empty string, this function will access the active model. which values: 0: Model exists (returns a true or a false) 1: Animation on 2: Model is dirty (has unsaved changes) 3: Run Mode (0 means multithreaded; 1 means single threaded) 4: Sim Running	R
ModelSettings-Set(string model-Name, integer which, real value)	Sets the value of a model Setting to the real value specified by the *which* argument. Returns a 0 for success and a non zero integer for failure. *which* values: 1: Animation on 2: Dirty (needs to be saved) 3: Run Mode (0 means multithreaded; 1 means single threaded)	I
MoveBlock(inte-ger blockNumber, integer xPixel, inte-ger yPixel)	Moves the specified block the specified number of pixels. Coordi-nates refer to the upper left corner of the block.	I
Move-BlockTo(integer blockNumber, inte-ger 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 ExtendSim model, library, or text file, using the *full-path* name on the disk. This creates a new front window if open-ing a model or text file. Returns a zero if successful.	I

Scripting	Description	Return
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 if *neighbor* is -1 (no neighbor) 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
PlaceBlockIn-Hblock (string blockName, string libName, integer xPixel, integer yPixel, integer HblockNum)	Places a copy of the named block from the named library in the H-block that is specified by the HblockNum argument. See Place-Block in your manual for more information. Returns FALSE if successful.	I
PlaceDot-Block(integer xPixel, integer yPixel, integer neighbor, integer side, integer width, integer Hblock-Name)	Places a dot block into the worksheet. Arguments are similar to PlaceBlock, and PlaceTextBlock.	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
PlaceTextBlockIn-Hblock(string text, integer xPixel, integer yPixel, integer neighbor, integer side, integer width, integer Hblock-Num)	This function places a TextBlock in the Specified HBLock. See the description of PlaceTextBlock for more information.	I
QuickTimeAvail-able()	Returns whether or not quicktime is available on the current machine.	I
SelectConnection (integer block-From, integer con-From, 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

Scripting	Description	Return
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 has the effect of closing the worksheet without querying the user if they want to save, or not.	V
SuppressWork-sheetRedraw(integer suppress)	If supress is TRUE, stops any update drawing of the worksheet until a call to SuppressWorksheetRedraw(FALSE), whereupon the worksheet will be redrawn.	V
UnselectAll()	Unselects all blocks, connections, etc.	V
WinSetFore-groundWindow(integer handle)	Sets the application associated with the passed in handle to be the foreground window. Passing in a zero, sets ExtendSim as the fore-ground window. Windows handles for other applications then ExtendSim will need to be acquired through some means. One example would be querying the Excel object model through OLE/COM to return the object handle for Excel. See the code of the Command block in the Value library for an example of doing this.	V
WinShellExe-cute(string operation, string fileName, string params, string dir, integer show)	This calls the Windows ShellExecute() function. This specifies all arguments in the ShellExecute() function call but the first (HWND), as ExtendSim supplies that argument internally.	I
WorksheetRe-fresh(integer block-Number)	Forces a worksheet refresh (redraw) of the worksheet window containing the specified block.	I
worksheetSetting-Get(integer block-Num, integer which)	Gets a setting on the worksheet associated with the specified block. (Blocks on the top level will affect the main worksheet, blocks inside an Hblock will affect that Hblock worksheet, if it's open.) Which takes the following values: 1: left 2: top 3: width 4: height 5: scroll X 6: scroll Y	I
worksheetSet-tingSet(integer blockNum, integer which, integer value)	Sets a setting on the worksheet associated with the specified block. Also works on Hblocks. Uses “which” values the same way as WorkSheetSettingGet.	I

Reporting

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

Reporting	Description	Return
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 Block-Report message. Useful to set up column headers for that block type.	I

Plotting/Charts

These functions allow you to provide graphs and data in any block you create.

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.

Some of these functions only apply to the blocks in the Plotter library which is a Legacy library that was replaced by the Chart library as of ExtendSim 10.

The following arguments are used in the plotting functions:

Plot argument	Definition
Color	Color of the plot traces. You can use the numbers or their associated Extend-Sim constants. Note that these are different than the animation colors. 33: BlackColor 205: RedColor 341: GreenColor 409: BlueColor 273: CyanColor 137: MagentaColor
EndTime	X-axis value that corresponds with the last point of the array.
FunctionName	Name of a real, single argument function. For example, cos(x) should be entered as `cos`.
IsXLog	If TRUE, the X axis is logarithmic.
IsY2Log	If TRUE, the Y2 axis is logarithmic.
IsYLog	If TRUE, the Y axis is logarithmic.
LineFormat	Format for traces. 0: connected points 1: rectangular connected points (no interpolation between points) 2: dots
MaxLines	Initial number of rows in the plot’s data pane.
NumFormat	Format for the numbers in the data pane. 0: General 1: decimal (2 places) 2: integer 3: scientific (exponential)

Plot argument	Definition
Pattern	Pattern of the plot line. You can use the numbers or their associated ExtendSim constants. Note that these are different than the animation patterns. 0: BlackPattern 1: DkgrayPattern 2: GrayPattern 3: LtgrayPattern Note: patterns are not supported as of release 10.
Plot	Number (0 to 3) specifying one of the four possible plot pages. See the blocks in the Plotter library for an example. As of ExtendSim 10, the Plotter library is a Legacy library.
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.
SigName	Name for the trace
StartTime	X-axis value that corresponds with the first point (y[0]) of the array.
SymFormat	Symbols used on traces. 0 – .; 1 – □; 2 – △; 3 – ○; 4 – +; 5 – ■; 6 – ▲; 7 – ● 8 – # (displays the trace's number); 9 – ✕
UseY2Axis	If TRUE, the trace is plotted to the limits of the Y2 axis instead of the main Y axis.
WhichSig	Number (0, 1, 2 ...) specifying one of the plot traces
Y	Name of the dynamic array used in the plot. Use MakeArray to allocate Y before using it in a function.

General plotting

☞ See Pie Chart, Bar Chart, and Scatter Chart functions starting on page 316.

General plotting	Description	Return
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. See also the PlotterAutoScaleLimits function.	V
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.	V
BarGraph(integer plot, integer aBins, real aMin, real aMax)	Sets the number of bins, the minimum value, and maximum value for a bar graph.	V

General plotting	Description	Return
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	V
ChangeSignal-Color (integer plot, integer whichSig, integer color)	Specifies the color for the trace.	V
ChangeSignalSymbol(integer plot, integer whichSig, integer symFormat)	Specifies the symbol for the trace.	V
ChangeSignal-Width(integer plot, integer whichSig, integer width)	Specifies the width (in pixels) for the trace.	V
ClosePlotter(integer plot)	Closes the plot window, if open.	V
ClosePlotter2 (integer blockNumber, integer plot)	This function just closes the specified plotter window, if it is open.	V
DisposePlot(integer plot)	Disposes of a plot window and all its saved plots. This does not dispose of any installed data arrays.	V
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.	V
GetAxisName(integer plot, integer whichAxis)	Returns the name of the specified axis. whichAxis: 1 - x 2 - y1 3 - y2	S

General plotting	Description	Return
GetPlotter-Value(integer plot, integer which)	Gets a setting value on the specified plotter or chart. “which” takes the following values: 1: X axis decimal places 2: Y axis decimal places 3: Y2 axis decimal places 4: number of active traces (just get) 5: label angle for Bar Chart or pieSize for Pie Chart 6: plotter type ((1 – linear plot; 2 – scatter plot; 3 – error bars; 4 – strip chart;,5 – worm plot; 6 – bar plot, 7-pie chart, 8 - bar chart) Note: this function is a duplicate of PlotterValueGet.	R
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
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 of the line. The second and subsequent times, the formatting options are ignored, and the only action a call to installArray takes is to reinstall the array itself. If you precede a call to InstallArray with a call to RemoveSignal, the formatting information will be used. Note: patterns are not supported as of release 10.	V

General plotting	Description	Return
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.	V
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). Note: patterns are not supported as of release 10.	V
NumPlotPoints (integer plot, integer points)	Specifies the number of points to be stored for worm and strip plots.	V
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 than adding 1 to it as the PlotNewPoint function will do.	V
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.	V
PlotSignalFormat (integer plot, integer whichSig, integer lineFormat, integer numFormat)	Specifies the line format and number format for the trace.	V

General plotting	Description	Return
PlotterAutoscaleLimits (integer plot, real minX, real maxX, real minY, real maxY, real minY2, real maxY2)	Sets the limits for the autoscaling functionality in the specified plot. Autoscaling will force the max and min values to these values if they are set.	
PlotterBackground (integer plot, string backName)	Specifies that the named picture should be used as the background for the specified plotter. The picture file must be in the extensions folder. See "Picture and movie files" on page 90 for more information on how external pictures are used in ExtendSim.	I
PlotterNameGet (integer plot)	Returns the name of the specified plotter.	S
PlotterNameSet(integer plot, string newName)	Sets the name of the specified plotter.	I
PlotterSignalColorSet(integer plot, integer whichSig, integer Hue, integer Sat, integer Value)	Sets the color of the specified signal. This function will automatically set the color value of the signal to custom and will then define the custom color values to be the values you pass in. As of ExtendSim 10, use PlotterSignalEColorSet instead.	V
PlotterSignalEColorSet (integer plot, integer whichSig, integer EColorValue)	Sets the color of the specified signal. See "EColors" on page 365.	V
PlotterSignalValueGet (integer plot, integer whichSig, integer whichValue)	Returns a value associate with the specified signal. The whichValue argument specifies the number of the signal: 0: Color (prior to ExtendSim 10) 1: Hue (prior to ExtendSim 10) 2: Sat (prior to ExtendSim 10) 3: Value (prior to ExtendSim 10) 4: Signal visibility 5: use Y2Axis 6: Line format 7: Line symbol 8: Line thickness 10: EColor Value to set the signal line color 11: The number of data points in this signal	I
PlotterSignalValueSet (integer plot, integer whichSig, integer whichValue, integer value)	Sets the value of a specified aspect of a plotter signal. Which values are the same as for PlotterSignalValueGet.	I

General plotting	Description	Return
PlotterSquare(integer plot, integer trueFalse)	If true, forces the window to be square. If false, it removes the squaring restriction.	V
PlotterValueGet(integer plot, integer which)	Gets a setting value on the specified plotter or chart. “which” takes the following values: 1: X axis decimal places 2: Y axis decimal places 3: Y2 axis decimal places 4: number of active traces (just get) 5: label angle for Bar Chart or pieSize for Pie Chart 6: plotter type ((1 – linear plot; 2 – scatter plot; 3 – error bars; 4 – strip chart;,5 – worm plot; 6 – bar plot, 7-pie chart, 8 - bar chart) Note: this function is a duplicate of GetPlotterValue	R
PlotterValueSet(integer plot, integer which, real value)	Sets a setting value on the specified chart or plotter. “which” takes the following values: 1: X axis decimal places 2: Y axis decimal places 3: Y2 axis decimal places 5: label angle for Bar Chart or pieSize for Pie Chart	I
PlotterXAxisCalendar(integer plot, integer xAxisCalendar, integer format)	Turns on or off Calendar date behavior on the x axis of the specified plotter. The xAxisCalendar flag set to a true value will set the plotter to redraw with Calendar date values on the x Axis. The format parameter takes the following values: 0: everything 1: Just date, (ignore time) 2: Just time, (ignore date) 3: tight format. (Two digit year, and don't show time value if zero.)	V
PlotterXAxisIsTime(integer plot, integer xAxisTime)	This function sets a true/false flag in the specified plotter that tells the plotter if the X axis is defined as a time value or not. The primary purpose for which the plotter uses this is for optimization. If the X axis of the plotter is scaled to a certain max value, and the next X value is beyond that value, the plotter knows that it is done drawing because all subsequent X value will be higher than the current values. Without this optimization, the DE plotter, which is based on a scatter plotter, not a continuous plotter, will continue to attempt to draw the rest of the points in the data unnecessarily.	V
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.	V
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.	V

General plotting	Description	Return
RemoveSignal(integer plot, integer unused)	Removes the last installed array or function from the plot. For *unused*, enter any integer; 0 is suggested.	V
RenamePlotter(integer plot, string newname)	Renames the plotter.	V
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.	V
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.	V
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*.	V
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.	V
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 entered the plot dialog.	V
ShowPlot2(integer blockNumber, integer plot)	Shows the specified plotter. The only difference between this function and ShowPlot is the block number argument, which allows you to show a plotter in a remote block.	V
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.	V

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.

Pie and Bar chart functions

Pie charts	Description	Return
PieChart(integer plot, real pieSize, real holeSize	Sets the specified plotter to be a Pie Chart. PieSize and holeSize range from 0.0 to 1.0. PieSize specifies how big the Pie Chart is, relative to the size of the graph area. HoleSize specifies the same for the hole in the middle of the Pie Chart. A hole size of greater than 0.0 makes the Pie Chart a Donut Chart.	V
PieChartSlice(integer plot, integer whichSlice, string sliceLabel, long color, long explode, real explodePercent, real value)	Specifies the size and color of a slice of a Pie Chart. Explode should be set to TRUE if the pie slice should be displayed separated from the rest of the Pie Chart. Value is the relative size of the section.	V

Bar charts	Description	Return
BarChart(integer plot, integer stacked)	Sets the specified plotter to be a Bar Chart. When there are multiple Series (i.e. a grouped bar chart), Stacked allows the bars to be stacked on top of each other rather than next to each other.	V
BarChartCategoryCountGet(integer plot)	Returns the specified value for the specified bar chart set.	I
BarChartCategoryCountSet(integer plot, integer numCategories)	Sets the number of categories in the specified bar chart. New categories will be defined with empty strings. Will not change existing categories.	V
BarChartCategoryGet(integer plot, integer which)	Returns the specified category name. Returns an empty string if the *which* value is out of range.	S
BarChartCategorySet(integer plot, integer which, string categoryName)	Sets the specified category to the specified string. Has no effect if the *which* value is out of range.	V
BarChartSet(integer plot, integer set, string setLabel, integer color)	Creates a new set, or modifies an existing set in the specified Bar Chart. Set is the index value of the set.	V
BarChartValueGet(integer plot, integer whichSet, integer whichValue)	Gets the value for a specified Bar Chart.	R

Bar charts	Description	Return
BarChartValue-Set(integer plot, integer whichSet, integer which-Value, real value)	Sets the values within a specific set in the Bar Chart.	V

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	Return
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.	V
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.	V

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 blocks in the Chart library.

Database functions

These are used to create, read, write, import, export, and delete databases and their components. They are used by blocks to create and manipulate databases during a run. All database indexes are 1 based, so they start at 1 and end at N. Databases and tables maintain their indexes, even when other databases or tables are deleted. Fields and record indexes can change if earlier fields or records are deleted.

☞ Reserved databases use specially named functions to manipulate them so they don’t inadvertently get changed by the user or developer. These functions have the word “Reserved” in their name.

Blocks can be linked to parts of the database, so if the database structure or data changes, they will be notified and can take action. See “Dynamic linking” on page 280 and the LINKSTRUCTURE and LINKCONTENT messages.

Error codes

For database read/write functions, the Database menu command “Read/Write Index Checking“ can enable error messages if a read/write function call has illegal indexes. This is a good tool to find illegal indexes and leaving this check on does not impact the speed of a simulation run.

Leave it off if it is preventing a legacy model from running. New models have this check on by default.

In most cases, functions return zero if no error, or non-zero error codes when an error occurs:

Error code	Value	Description
no such record	-1	One of the indexes are incorrect.
no such parent	-2	Trying to set a child field to a value that is not a parent value.
not unique error	-3	Trying to set a cell to a non-unique value in a field with the "Unique" property set.
not unique index	>0	DBSetDataAs... functions will return the record index of the currently existing unique value if you try to set a cell to a non-unique value.
not linked error	-4	Returned by DBDataGetAsNumber() on a *list of tables* field if a table name found in a database cell doesn't exist and doesn't have an index in the database.

Creating and deleting database components

DB creating/deleting	Description	Return
DBDatabaseCreate (string databaseName)	Creates database databaseName and returns its index. Error: returns negative error code if existing name already used.	I
DBDatabaseDelete(string databaseName)	Deletes entire database. Returns negative error code if DB doesn't exist.	I
DBDatabaseDeleteByIndex(integer databaseIndex)	Deletes entire database using index, returns error code. See also DBDatabaseDelete() in the Technical Reference.	I
DBDatabaseTabDelete(integer databaseIndex, Str255 existingTabName)	Deletes existingTabName tab in databaseIndex database viewer. Returns -1 if error.	I
DBFieldCreate(string databaseName, string tableName, string fieldName, integer fieldFormat, integer decimals, integer unique, integer readOnly, integer invisible)	Creates field fieldName with specified attributes tableName and returns its index. Error: returns negative error code if existing name already used. Field format constants defined: DB_FIELDTYPE_INTEGER_VALUE, DB_FIELDTYPE_INTEGER_BOOLEAN (checkbox), DB_FIELDTYPE_REAL_GENERAL, DB_FIELDTYPE_REAL_SCIENTIFIC, DB_FIELDTYPE_REAL_PERCENT, DB_FIELDTYPE_REAL_CURRENCY, DB_FIELDTYPE_REAL_DATE_TIME, DB_FIELDTYPE_REAL_DB_ADDRESS, DB_FIELDTYPE_STRING_VALUE, DB_FIELDTYPE_TABLELIST	I

DB creating/deleting	Description	Return
DBFieldCreateByIndex(integer databaseIndex, integer tableIndex, string fieldName, integer fieldFormat, integer decimals, integer unique, integer readOnly, integer invisible)	Creates field *fieldName* using specified indexes, returns error code. See also DBFieldCreate() in the Technical Reference.	I
DBFieldDelete(string databaseName, string tableName, string fieldName)	Delete field. Returns negative error code if field doesn't exist.	I
DBFieldDeleteByIndex(integer databaseIndex, integer tableIndex, integer fieldIndex)	Deletes field using index, returns error code. See also DBFieldDelete() in the Technical Reference.	I
DBRecordsDelete(integer databaseIndex, integer tableIndex, integer startRecord, integer endRecord)	Delete records from a table. Returns negative error code if table doesn't exist.	I
DBRecordsInsert(integer databaseIndex, integer tableIndex, integer insertAtRecord, integer numberRecords)	Insert at record index or append (insertAtRecord is zero) new records to a table, returns tableIndex if ok. Returns negative error code if table doesn't exist.	I
DBRelationCreate(string databaseName, string tableChildName, string fieldChildName, string tableParentName, string fieldParentName)	Create relation, returns negative error code if anything doesn't exist.	I
DBRelationDelete(string databaseName, string tableChildName, string fieldChildName, string tableParentName, string fieldParentName)	Delete relation. Returns negative error code if relation doesn't exist.	I

DB creating/deleting	Description	Return
DBTableCloneToTab(integer databaseIndex, integer tableIndex, string tabName)	Creates the tabName if not there, and clones the table to the tab.	V
DBTableCreate (string databaseName, string tableName)	Creates table tableName and returns its index. Error: returns negative error code if existing name already used.	I
DBTableCreateByIndex(integer databaseIndex, string tableName)	Creates table *tableName* in current database using indexes, returns negative error code. See also DBTableCreate() in the Technical Reference.	I
DBTableDelete(string databaseName, string tableName);	Delete table. Returns negative error code if Table doesn't exist.	I
DBTableDeleteByIndex(integer databaseIndex, integer tableIndex)	Deletes table using index, returns error code. See also DBTableDelete() in the Technical Reference.	I
DBToolTipsGet(integer databaseIndex, integer tableIndex, integer fieldIndex)	Gets the string value of the tooltip. 1) All indexes good, gets field tooltip. 2) FieldIndex is zero, gets table tooltip. 3) Field and Table index zero, gets database tooltip.	S
DBToolTipsSet(integer databaseIndex, integer tableIndex, integer fieldIndex, string value)	Sets the tooltip to *value*. 1) All indexes good, sets field tooltip. 2) FieldIndex is zero, sets table tooltip. 3) Field and Table index zero, sets database tooltip.	I

Selecting parts of a database

These functions open a database component selection dialog so the user can select a desired component.

DB selecting	Description	Return
DBChildPopupSelector(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordNum, integer trueForSelectorDialog)	Opens a child field parent value selector similar to clicking on a child field popup arrow, and changes the child value according to which parent value the user selects. If trueForSelectorDialog is TRUE, a selector dialog is shown with a scroll bar, good for a large numbers of parent values. If FALSE, a popup menu is shown, which is good for a small number of parent values. Returns the selected parent record index or zero if canceled.	I

DB selecting	Description	Return
DBDatabasePopupSelector(real currentDBIndex)	Opens a database selector dialog and selects the currentDBIndex if not BLANK or zero. If BLANK or zero, doesn't select any entries in the list. Returns the selected database index or zero if canceled.	I
DBFieldPopupSelector(integer databaseIndex, integer tableIndex, real currentFieldIndex)	Opens a field selector dialog and selects the currentFieldIndex if not BLANK or zero. If BLANK or zero, doesn't select any entries in the list. Returns the selected field index or zero if canceled.	I
DBRecordPopupSelector(integer databaseIndex, integer tableIndex, integer fieldIndex, real currentRecordIndex)	Opens a record selector dialog and selects the currentRecordIndex if not BLANK or zero. If BLANK or zero, doesn't select any entries in the list. Returns the selected record index or zero if canceled.	I
DBTablePopupSelector(integer databaseIndex, integer tableIndex)	Opens a table selector dialog and selects the currentTableIndex if not BLANK or zero. If BLANK or zero, doesn't select any entries in the list. Returns the selected table index or zero if canceled.	I

Copying parts of a database

DB copy	Description	Return
DBDatabaseCopy(integer fromDatabaseIndex, string newDatabaseName)	Copies entire database to a new database. Returns index of the new database or -1 if error (bad index or same name as existing database).	I
DBTableCopy(integer fromDatabaseIndex, integer fromTableIndex, integer toDatabaseIndex, string newTableName)	Copies and optionally renames an existing table into its current database (must use newTableName) or another database (If newTableName is blank, keeps the existing table name). Returns index of the new table or -1 if error (bad index or same table name as existing table in that database). Note: if the target table's name starts with a + (plus sign), this function will append the existing table's data to the target.	I

Import and export a database

DB import/export	Description	Return
DBDatabaseExport(string databaseName, string pathName)	Export database to DB text file pathName. If pathName is a blank string, let the user select a text file. Returns -1 if error.	I

DB import/export	Description	Return
DBDatabaseImport(string databaseName, string pathName)	Import database from text file pathName, replacing the database named databaseName or creating it if it didn't exist. If the database name is in the form "databaseName<delete>", it deletes any left over tables that were not imported in this file. If pathName is a blank string, let the user select a text file. Returns the database index or -1 if it fails. Sends both a LinkStructure message (what changed: DB replaced) and a LinkContent message (what changed: data changed) to linked blocks. See DILinkUpdateInfo() for an explanation of what changed.	I
DBTableExportData(string pathName, string userPrompt, string format, integer databaseIndex, integer tableIndex, integer rows, integer columns)	Exports the table data to a delimited text file. If *pathName* is a blank string, lets the user select a text file. If *format* is a blank string, uses a tab as delimiter (see Import() functions). If the *databaseIndex* is negative, it outputs the field names (separated by delimiters) as the first line of the text file. Returns -1 if error.	I
DBTableImportData(string pathName, string userPrompt, string format, integer databaseIndex, integer tableIndex)	Imports a delimited text file into a database table and returns the number of rows imported. If *format* is a blank string, uses a tab as delimiter (see Import() functions). If *databaseIndex* is negative, it assumes that the first line of the text file contains the field names and skips that line. This function maintains existing relations in the table but will discard relations that violate parent/child data requirements: all child data in a relation must be blank or exist in the parent data set. NOTE: This function automatically resizes the table to the number of rows imported, so you do not need to allocate any records in the table before calling this function.	I

Database properties

DB properties	Description	Return
DBDatabaseExists (integer databaseIndex)	Passing in a database index, returns TRUE if the database exists, FALSE if it doesn't exist. Also see DBTableExists(), DBFieldExists(), and DBRecordExists().	I
DBDatabaseGetIndex(string databaseName)	Returns database index or negative error if not found.	I
DBDatabaseGetName(integer databaseIndex)	Gets the name of the database or blank string if bad index.	S

DB properties	Description	Return
DBDatabaseRename(integer databaseIndex, string newDatabaseName)	Renames a database. Returns a negative error code if *newDatabaseName* already exists or the old database doesn't exist.	I
DBDatabasesGetNum()	Because database indexes remain constant even if some databases are deleted, this returns number of database slots of which some could be empty. To count how many actual databases there are, use a loop and count the databases that have a non-blank name.	I
DBDatabaseShowHideReserved(integer showIfTrueHideIfFalse)	Use this to hide or show reserved databases without using the menu command. Reserved databases are discussed on page 114. **Note**: When the model closes, the reserved databases return to being hidden. So this command must be called again whenever the model is opened.	V
DBDatabaseTabChangeName(integer databaseIndex, Str255 existingTabName, Str255 newTabName)	Changes the name of existingTabName to newTabName in databaseIndex database viewer. Returns -1 if error.	I
DBFieldExists(integer databaseIndex, integer tableIndex, integer fieldIndex)	Passing in a database index, table index, field index, returns TRUE if the field exists, FALSE if it doesn't exist. Also see DBDatabaseExists(), DBTableExists(), and DBRecordExists().	I
DBFieldGetIndex(integer databaseIndex, integer tableIndex, string fieldName)	Returns field index or negative error if not found.	I
DBFieldGetName(integer databaseIndex, integer tableIndex, integer fieldIndex)	Gets the name of the field or blank string if bad index.	S
DBFieldGetProperties(integer databaseIndex, integer tableIndex, integer fieldIndex, integer which)	*which*: 1: fieldType 2: fieldDecimals 3: fieldUnique 4: fieldReadOnly 5: fieldInvisible 6: IfFieldID 7: isParentField (note: use judiciously; this is very slow) 8: isChildField, (note: use judiciously; this is very slow) -1 if error Also see the DBFieldCreate() function on page 319.	I

DB properties	Description	Return
DBFieldGetPropertiesUsingAddress(real dbAddress, integer which)	This uses a DBAddress (see the DBAddress functions). For the values of *which*, see the DBFieldGetProperties function, above. Also see the DBFieldCreate() function on page 319.	I
DBFieldMove(integer databaseIndex, integer tableIndex, integer fieldIndex, integer newFieldIndex)	Moves field *fieldName* to *newFieldIndex*, moving other fields in the process. Returns a negative when indexes are incorrect.	I
DBFieldRename(integer databaseIndex, integer tableIndex, integer fieldIndex, string newFieldName)	Renames a field. Returns a negative error code if the field already exists or the old field doesn't exist.	I
DBFieldSetInitialize(integer databaseIndex, integer tableIndex, integer fieldIndex, integer initFlag, integer initSimFlag, string value)	Set the field's initialization parameters. *initFlag*: 0 no init (default), 1 init to initDataValue. *initSimFlag*: 0 init each run (default), 1 init first run of multisim. *value*: what value to init the field, real or string for string fields. Returns an error code if bad indexes.	I
DBFieldSetProperties(integer databaseIndex, integer tableIndex, integer fieldIndex, integer which, integer newValue)	*which*: 1: fieldType, 2: fieldDecimals, 3: fieldUnique, 4: fieldReadOnly, 5: fieldInvisible, 6: IfFieldID. Returns -1 if index error. See the DBFieldCreate() function, above.	I
DBFieldsGetNum(integer databaseIndex, integer tableIndex)	Returns number of fields. Returns negative error code if no fields or no such table index.	I
DBRecordExists(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	Passing in a database index, table index, field index, record index, returns TRUE if the record (database cell) exists, FALSE if it doesn't exist. Also see DBDatabaseExists(), DBTableExists(), and DBFieldExists().	I

DB properties	Description	Return
DBGetSize(integer databaseIndex, integer tableIndex, integer fieldIndex)	Used to find the amount of memory used by the DB. Returns the size in bytes of the selected database, table, or field by specifying its indexes. If the index is -1, all of the sizes of the databases, tables, or fields are added up. For example, if databaseIndex is -1, the size of all of the databases are totaled and returned. If any of the indexes are -1, the next indexes are ignored, so if the databaseIndex is 1, tableIndex is -1, and fieldIndex is 2, the function ignores the fieldIndex and returns the sum of sizes of all of the tables in databaseIndex 1.	R
DBRecordIDFieldGetIndex(integer databaseIndex, integer tableIndex)	Returns field index of the recordID field, if any. Returns negative error if no fieldID found.	I
DBRecordsGetNum(integer databaseIndex, integer tableIndex)	Returns number of records in a table. Returns negative if no such table index.	I
DBRelationsGetNames(integer databaseIndex, integer relationIndex, string relationNames[])	Returns zero error code and tableChild, fieldChild, tableParent, fieldParent in string relationNames[4]	I
DBRelationsGetNum(integer databaseIndex)	Returns number of relations in the DB. Returns negative error code if no DB.	I
DBTableExists (integer databaseIndex, integer tableIndex)	Passing in a database and table index, returns TRUE if the table exists, FALSE if it doesn’t exist. Also see DBDatabaseExists(), DBFieldExists(), and DBRecordExists().	I
DBTabGetTableIndexList(integer databaseIndex, string tabName, integer integerArray)	Puts database table indexes in integerArray, where indexes represent the tables in the tab named tabName. IntegerArray is a dynamic array declared as integer integerArray[]. Returns an integer that specifies how many table indexes are in integerArray. Returns -1 if the either or both of the databaseIndex or the tabName don't exist.	I
DBTableGetIndex(integer databaseIndex, string tableName)	Returns table index or negative error if not found.	I
DBTableGetProperties(integer databaseIndex, integer tableIndex)	Returns the initialization method: 0 is no initialization, 1 is delete all records for all runs, 2 is delete all records for only the first run. Returns -1 if error.	I

DB properties	Description	Return
DBTableGetName(integer databaseIndex, integer tableIndex)	Gets the name of the table or blank string if bad index.	S
DBTableSetProperties(integer databaseIndex, integer tableIndex, integer initializeMethod)	Sets the initialization method: 0 is no initialization, 1 is delete all records for all runs, 2 is delete all records for only the first run. Returns -1 if error.	I
DBTableRename(integer databaseIndex, integer tableIndex, string newTableName)	Renames a table. Returns a negative error code if the table already exists or old table doesn't exist.	I
DBTablesGetNum(integer databaseIndex)	Because table indexes remain constant even if some tables are deleted, returns number of table slots of which some could be empty. To count how many actual database tables there are, use a loop and count the tables that have a non-blank name.	I

Read and write to a database

☞ Note that reading a “List of Tables” field returns the table index if read as a number, and the name of the table if read as a string.

DB read/write	Description	Return
DBDataGetAsNumber(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	Reads the value as a number without any formatting. Returns a NoValue if the cell is blank, even if it is an integer cell. Note: Random distributions in cells will return a different random number each time that this function is called. Note: Using this function to read a “List of Tables” field returns the index of the table read.	R
DBDataGetAsNumberParentAltField(real childDBAddress, integer altFieldIndex)	This function, when called with a child DBAddress, allows returning the value from a different field of the parent record. Reads the value as a number without any formatting. Note that random distributions in cells will return a different random number each time that this function is called. Note: Using this function to read a “List of Tables” field returns the index of the table read.	R
DBDataGetAsNumberUsingAddress(real dbAddress)	Using a DBAddress, read the value as a number without any formatting. Note that random distributions in cells will return a different random number each time that this function is called. Note: Using this function to read a “List of Tables” field returns the index of the table read.	R

DB read/write	Description	Return
DBDataGetAsString(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	Reads the value as a string at full precision. Percentage format returns the normalized value as “1.00” for database cells that read 100%. NOTE that if the table is a table of tables, this returns the table name (see DBGetDataAsNumber() below. Note that random distributions in cells will return a different random number each time that this function is called. Note: Using this function to read a “List of Tables” field returns the name of the table read.	S
DBDataGetAsStringParentAltField(real childDBAddress, integer altFieldIndex)	This function, when called with a child DBAddress, allows returning the value from a different field of the parent record. Reads the value as a string at full precision. Percentage format returns the normalized value as “1.00” for database cells that read 100%. NOTE that if the table is a table of tables, this returns the table name (see DBGetDataAsNumber() below. Note that random distributions in cells will return a different random number each time that this function is called. Note: Using this function to read a “List of Tables” field returns the name of the table read.	S
DBDataGetAsStringUsingAddress(real dbAddress)	Using a DBAddress, read the value as a string at full precision. Percentage format returns the normalized value as “1.00” for database cells that read 100%. NOTE that if the table is a table of tables, this returns the table name (see DBGetDataAsNumber() below. Note that random distributions in cells will return a different random number each time that this function is called. Note: Using this function to read a “List of Tables” field returns the name of the table read.	S
DBDataGetDateAsSimTime(integer dbIndex, integer tableIndex, integer fieldIndex, integer recordIndex, integer timeUnits)	Reads a database cell as a date, and converts it to a simulation time value. This allows the user to read a date value directly as a simulation time value, avoiding the conversion process.	R
DBDataGetDateAsSimTimeUsingAddress(real addressValue, integer timeUnits)	Using a dbAddress, reads a database cell as a date, and converts it to a simulation time value. This allows the user to read a date value directly as a simulation time value, avoiding the conversion process.	R

DB read/write	Description	Return
DBDataGetParent(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex, integer parentArray[3])	Gets the parent information from a child field. Use it to either return these database indexes of the parent: parentArray[0] returns parent Table index parentArray[1] returns parent Field index parentArray[2] returns parent Record index (or zero if no child value has been set) Or, as with parentArray[2], to return the parent record index (or zero if no child value has been set). Returns a negative error code.	I
DBDataGetParentUsingAddress(real dbAddress, parentArray[3])	Using a DBAddress, gets the parent information from a child field address. Use it to either return these database indexes of the parent: parentArray[0] returns parent Table index parentArray[1] returns parent Field index parentArray[2] returns parent Record index (or zero if no child value has been set) Or, as with parentArray[2], to return the parent record index (or zero if no child value has been set). Returns a negative error code.	I
DBDataSetAsNumber(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex, real valueDouble)	Write value of field as number to a record. Returns a negative error code. Only sends LINKCONTENT message to block if value has changed. If setting a unique field cell to a non-unique value, returns the record index of the original unique value. This function does not work with reserved databases. See DBDataSetAsNumberReserved(), which works only with reserved databases.	I
DBDataSetAsNumberReserved(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex, real valueDouble)	This function works only with reserved databases. Otherwise, it is the same as DBDataSetAsNumber(), which works only with non-reserved databases.	I
DBDataSetAsNumberUsingAddress(real dbAddress, real valueDouble)	Using a DBAddress, write value of field as number to a record. Returns a negative error code. Only sends LINKCONTENT message to block if value has changed.If setting a unique field cell to a non-unique value, returns the record index of the original unique value. See DBDataSetAsNumberUsingAddressReserved(), which works only with reserved databases.	I
DBDataSetAsNumberUsingAddressReserved(real dbAddress, real valueDouble)	This function works only with reserved databases. Otherwise, it is the same as DBDataSetAsNumberUsingAddress(), which works only with non-reserved databases.	I

Functions

DB read/write	Description	Return
DBDataSetAsParentIndex(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex, integer parentIndex)	For a child field, sets the parent index directly rather than using DBDataSetAs... functions. Only sends LINKCONTENT msg to block if value changed. Note that you can set the parentIndex to zero if you want to set the child value to <no value yet>. Also see DBDataSetAsParentIndexReserved(), which works only with reserved databases.	I
DBDataSetAsParentIndexReserved(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex, integer parentIndex)	This function works only with reserved databases. Otherwise, it is the same as DBDataSetAsParentIndex(), which works only with non-reserved databases.	I
DBDataSetAsParentIndexUsingAddress(real dbAddress, integer parentIndex)	Using a DBAddress for a child field, sets the parent index directly rather than with DBPutDataAs... functions trying to set the parent index by finding the data value in the parent. Only sends LINKCONTENT msg to block if value changed. Note that you can set the parentIndex to zero if you want to set the child value to <no value yet>. See DBDataSetAsParentIndexUsingAddressReserved(), which works only with reserved databases.	I
DBDataSetAsParentIndexUsingAddressReserved(real dbAddress, integer parentIndex)	This function works only with reserved databases. Otherwise, it is the same as DBDataSetAsParentIndexUsingAddress(), which works only with non-reserved databases.	I
DBDataSetAsString(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex, string valueString)	Write value of field as string to a record. Percentage format expects the normalized value as "1.00" for database cells that read 100%. Returns a negative error code. Only sends LINKCONTENT message to block if value has changed.If setting a unique field cell to a non-unique value, returns the record index of the original unique value. See DBDataSetAsStringReserved(), which works only with reserved databases.	I
DBDataSetAsStringReserved(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex, string valueString)	This function works only with reserved databases. Otherwise, it is the same as DBDataSetAsString(), which works only with non-reserved databases.	I

DB read/write	Description	Return
DBDataSetAsStringUsingAddress(real dbAddress, string valueString)	Using a DBAddress, write value of field as string to a record. Percentage format expects the normalized value as “1.00” for database cells that read 100%. Returns a negative error code. Only sends LINKCONTENT message to block if value has changed.If setting a unique field cell to a non-unique value, returns the record index of the original unique value. See DBDataSetAsStringUsingAddressReserved(), which works only with reserved databases.	I
DBDataSetAsStringUsingAddressReserved(real dbAddress, string valueString)	This function works only with reserved databases. Otherwise, it is the same as DBDataSetAsStringUsingAddress(), which works only with non-reserved databases.	I
DBDataSetDateAsSimTime(integer dbIndex, integer tableIndex, integer fieldIndex, integer recordIndex, real simTimeVal, integer timeUnits)	Takes a simulation time value, converts it to a date, and sets a DB cell with that date value. See DBDataSetDateAsSimTimeReserved(), which works only with reserved databases.	I
DBDataSetDateAsSimTimeReserved(integer dbIndex, integer tableIndex, integer fieldIndex, integer recordIndex, real simTimeVal, integer timeUnits)	This function works only with reserved databases. Otherwise, it is the same as DBDataSetDateAsSimTime(), which works only with non-reserved databases.	I
DBDataSetDateAsSimTimeUsingAddress(real addressValue, real simTimeVal, integer timeUnits)	Using a dbAddress, takes a simulation time value, converts it to a date, and sets a DB cell with that date value. See DBDataSetDateAsSimTimeUsingAddressReserved(), which works only with reserved databases.	I
DBDataSetDateAsSimTimeUsingAddressReserved(real addressValue, real simTimeVal, integer timeUnits)	This function works only with reserved databases. Otherwise, it is the same as DBDataSetDateAsSimTimeUsingAddress() which works only with non-reserved databases.	I

Random data in a database

DB random data	Description	Return
DBDataGetCurrent-Seed(integer databa-seIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	Returns current running seed value from a random cell speci-fied by indexes. Returns zero if bad cell.	I
DBDataSetCurrent-Seed(integer databa-seIndex, integer tableIndex, integer fieldIndex, integer recordIndex, integer seedValue)	Sets current running seed value for a random cell specified by indexes. Returns 0 if bad cell.	I

DB random data	Description	Return

DBRandomDistributionGet(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex, string distName, real params[],real empTable[][2])	Gets the cell's distribution, if any assigned. Returns the info on the distribution in the *returnParams* array. Can use locally or statically declared *returnParams* (Real *returnParams*[10]) and *empTable* (Real empTable[maxEmpiricalrows][2]) or dynamic arrays for *returnParams*[], *empTable*[][2]; If using dynamic arrays, no Makearray() needed. You can dispose dynamic arrays after the call if not needed (Local arrays allow use in an Equation block).
	
	Case: All database indexes good (ignores *distName*), return info for a DB cell:
	1) Return all info in arrays. Function returns name of distribution or blank if not named.
	2) Return distributionIndex of 0 in array if not a random cell.
	
	Case: Only databaseIndex is good, rest are zero. Return info for *distName* named distribution, if it exists:
	1) If distName is good name: Return all info for that distribution (useSeed and seedInit will be zero as it is defined for each DB cell).
	2) distName doesn't exist: Return string "-1"
	
	Note that the returned distributionIndex values are the same as used in the RandomCalculate() function, with these additions:
	EmpiricalDiscrete 200
	EmpiricalStepped 201
	EmpiricalInterpolated 202
	
	returnParams[0] = distributionIndex;
	returnParams[1] = meanDistParam1;
	returnParams[2] = argDistParam2;
	returnParams[3] = modeParam3;
	returnParams[4] = locationParam4;
	returnParams[5] = lowerLimit;
	returnParams[6] = upperLimit;
	returnParams[7] = useSeed;
	returnParams[8] = seedInit;
	returnParams[9] = empiricNumRows;
	
	empTable contains the empirical distribution array, if any.
		S

DB random data	Description	Return
DBRandomDistributionSet(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex, string distName, integer distributionIndex, real param1, real param2, real param3, real locationParam4, real lowerLimit, real upperLimit, integer useSeed, integer seedInit, real empTable[][2])	This can be used in several cases: 1) Define or modify a named random distribution without setting a DB cell to it (must use this for empirical distributions): a) Set databaseIndex, other DB indexes should be zero: b) Use distributionIndex to set the distribution type. c) Set remaining params for this distribution. d) Fill empTable if this is an empirical distribution. 2) Set a DB cell to a named random distribution (must use this for empirical distributions): a) Set all DB indexes to point to a DB cell. b) Set distribution name as it is defined in database or via case 1 above. c) Set useSeed and seedInit for this DB cell. Other parameters are ignored as name defines distribution to use. 3) Set a DB cell to an unnamed non-empirical random distribution: a) Set all DB indexes to point to a DB cell. b) Set distribution name to "" (blank string). c) Set distribution index and parameters as in case 1 above. 4) Remove the random distribution from a cell and make it a constant: a) Set all DB indexes to point to a DB cell. b) Set distribution name to "" (blank string). c) Set distribution index to zero. *Note that the distributionIndex values are the same as used in the RandomCalculate() function, with these additions:* EmpiricalDiscrete 200 EmpiricalStepped 201 EmpiricalInterpolated 202 Returns -1 as error.	I

Functions

Sort and search in a database

DB sort/search	Description	Return
DBRecordFind(integer databaseIndex, integer tableIndex, integer notRecordID-FieldIndex, integer startingRecordIndex, integer exactMatch, string findValue)	Finds a record and returns index of record found. If fieldIndex is zero, uses RecordID field for search. startingRecordIndex can be zero or one to start at the first record. To keep searching for more matches, increment the returned record index by one for startingRecordIndex. If exactMatch is FALSE, finds record containing findValue without having to match entire field value. Returns 0 if record not found. Returns negative error if index error. NOTE: Different find functions return different error codes because of legacy concerns.	I
DBRecordFindMultipleFields(integer databaseIndex, integer tableIndex, integer startingRecordIndex, integer fieldIndex1, string findValue1, integer exactMatch1, integer fieldIndex2, string findValue2, integer exactMatch2, integer fieldIndex3, string findValue3, integer exactMatch3)	Finds a record and returns index of record found. Up to 3 fields can be searched. If only one is searched, make both fieldIndex2, fieldIndex3 zero. If only two are searched, make fieldIndex3 zero. StartingRecordIndex can be zero or one to start at the first record. To keep searching for more matches, increment the returned record index by one for startingRecordIndex. If exactMatch is FALSE, finds record containing findValue without having to match the entire field value. Returns -1 if record not found or index error. NOTE: Different find functions return different error codes because of legacy concerns.	I
DBRecordFindMultipleFieldsArray(integer databaseIndex, integer tableIndex, integer startingRecordIndex, integer fieldIndexArray[], string findValueArray[], integer exactMatchArray[])	Finds a record and returns index of record found. Any number of fields can be searched. Create three arrays that can be local, static or dynamic. For example, to declare local arrays in an Equation block to search 4 fields: integer fieldIndexArray[4], exactMatchArray[4]; string findValueArray[4]; The arrays can be larger than you need as only nonzero fieldIndexArray members will be searched. The first zero fieldIndex will end the fields searched. StartingRecordIndex can be zero or one to start at the first record. To keep searching for more matches, increment the returned record index by one for startingRecordIndex. If exactMatch is FALSE, finds record containing findValue without having to match the entire field value. Returns -1 if record not found or index error. NOTE: Different find functions return different error codes because of legacy concerns.	I

DB sort/search	Description	Return
DBRecordFindNumericalRange(integer databaseIndex, integer tableIndex, integer notRecordIDFieldIndex, integer startingRecordIndex, real lowerDouble, real upperDouble)	Finds a record and returns index of record found where number >= lowerDouble and <= upperDouble. To find exactly, make lowerDouble and upperDouble the same. If fieldIndex is zero, uses RecordID field for search. StartingRecordIndex can be zero or one to start at the first record. To keep searching for more matches, increment the returned record index by one for startingRecordIndex. Returns 0 if record not found or index error. NOTE: Different find functions return different error codes because of legacy concerns.	I
DBRecordFindParentRecordIndex(integer databaseIndex, integer tableIndex, integer childFieldIndex, integer startingRecordIndex, integer findParentIndexValue)	Returns the index of the child record found pointing to *startingParentIndex*. *StartingRecordIndex* can be 0 or 1 to start at the first record. To keep searching for more matches, increment the returned record index by 1 for *startingRecordIndex*. Returns index of found record or 0 for record not found, -1 for indexing error.	I
DBTableSort(integer databaseIndex, integer tableIndex, integer fieldIndex1, integer direction1, integer fieldIndex2, integer direction2, integer fieldIndex3, integer direction3)	Sorts the table using up to three fields and directions. Set the fieldIndexes of the fields you need sorted, and set the other fieldIndexes to zero. For example, if you want to sort only one field, set fieldIndex1 to that field index and set fieldIndex2 and 3 to zero. The direction arguments are TRUE for ascending and FALSE for descending. Returns -1 if there is an error.	I

DB address functions

These functions stuff the database or global array component indexes into a real number. For databases, it includes the database index, table index, field index, and record index. The resulting real number is very useful as an attribute value that can completely describe a database or global array address.

Depending on whether there are more than 1 million records, the top index limits for DB Addresses are:

Component	Algorithm 1	Algorithm 2
Databases	500	500
Tables	10,000	1,000
Fields	1,000	1,000
Records	1,000,000	10,000,000

Depending on the number of records needed, ExtendSim automatically switches to the correct algorithm. For example, Algorithm #2 is used if more than 1 million records are needed.

DB address	Description	Return
DBAddressCreate(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	Returns a DBAddress value from the indexes.	R
DBAddressGetAllIndexes(real addressValue, integer returnIndexesArray[4])	Return the indexes from a DBAddress value. ReturnIndexesArray can be declared as a local variable array (can be declared in equation blocks): integer returnIndexesArray[4]; returnIndexesArray[0] returns Database index returnIndexesArray[1] returns Table index returnIndexesArray[2] returns Field index returnIndexesArray[3] returns Record index	
DBAddressGetAsString(real addressValue)	Returns the string representation of a DBAddress value.	S
DBAddressGetDlg(real theInitVal)	Puts up a dialog for the user to specify the coords of a DBAddress. theInitVal is a starting value. BLANK leaves all blank.	R
DBAddressGetDlg2(integer DBIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	Similar to DBAddressGetDlg. Displays a dialog so the user can enter a database address. Note that in addition to entering the DB coordinates as separate values, you can also pass -2 for a DB coordinate, and this will hide that coordinate on the dialog. This allows you to enter just a DB index, for example, or just a DB and table index. Returns a database address.	R
DBAddressGetFromString(string dbAddressStr)	Returns the real database address when the string argument is of the form “D:T:F:R” where D, T, F, and R are the numerical indexes of the address.	R
DBAddressIncrementIndex(real addressValue, integer whichElement, integer incrementValue)	Increments or decrements (negative incrementValue) a DBAddress. *whichElement*: 1: database, 2: table, 2: field, 4: record	R
DBAddressReplaceIndex(real addressValue, integer whichElement, integer newValue)	Replaces an element (whichElement) of the DB address with a new element (newValue) and returns the new DBAddress. *whichElement*: 1: database, 2: table, 3: field, 4: record	R
DBDatabaseGetIndexFromAddress(real addressValue)	Returns a dabase index from the DBAddress value.	I
DBFieldGetIndexFromAddress(real addressValue)	Returns a field index from the DBAddress value.	I

DB address	Description	Return
DBRecordGetIndex-FromAddress(real addressValue)	Returns a record index from the DBAddress value.	I
DBTableGetIndexFromAddress(real addressValue)	Returns a table index from the DBAddress value.	I

Viewing a database

DB viewing	Description	Return
DBDatabaseOpen-Cell(integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	Opens the desired database table and scrolls to and selects the cell specified by fieldIndex and recordIndex. Returns a -1 if indexes are bad.	I
DBDatabaseOpen-Viewer(integer databaseIndex, string tableName)	Open the table data viewer. If tableName is blank or doesn't exist, open the default database viewer. Returns table index if successful or -1 if error.	I
DBDatabaseOpen-ViewerToTab(integer databaseIndex, Str255 tableName, Str255 openTabName)	Opens the table data viewer. If tableName is blank or doesn't exist, open the default database viewer. If openTabName exists, opens to that tab. If it is blank or doesn't exist, opens the last clicked tab. Returns table index if successful or -1 if error.	I
DBDatabaseClose-Viewer(integer databaseIndex, string tableName)	Close the table data viewer. If tableName is blank or doesn't exist, close the database viewer. Returns table index if successful or -1 if error.	I

Linking and notification

These functions register and unregister blocks from databases, independent of dialog items and their links, so linked blocks can be notified if the data that they are depending on changes in structure or content. This type of linking is different than user linking of parameters and data tables to a database because you don't specify anything to link. If the database changes content (e.g. the value in a cell that you registered changes) you will get a LINKCONTENT message. If the database table that you are linked to changes name or number of fields or rows, you will get a LINKSTRUCTURE message. Please see "Dynamic linking" on page 280, for information on finding out what changed when you get the LINKCONTENT or LINKSTRUCTURE messages.

☞ All registration set up by DBBlockRegisterContent() or DBBlockRegisterStructure() is good only for this model session. This disposal of links is done to prevent obsolete links sending unneeded messages that could slow down a simulation. When the model is closed and reopened, you will have to call these functions again (usually in the OPENMODEL message handler) for all links that you want.

For an overview, see “Registered blocks” on page 113.

DB linking/notify	Description	Return
DBBlockRegisterContent(integer blockNumber, integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	This will register the block with the selected part of the database so it will get LINKCONTENT messages when that data is changed. Returns negative code if block not found. Note that for content registration, individual cells as well as whole databases, tables, fields, and records can be registered. If databaseIndex is negative, register all databases. If tableIndex is negative, register all tables. If fieldIndex is negative and recordIndex is negative, register all cells in table. If fieldIndex is negative and recordIndex is good, register whole record. If fieldIndex is good and recordIndex is negative, register whole field.	I
DBBlockRegisterStructure(integer blockNumber, integer databaseIndex, integer tableIndex)	This will register the block with the selected part of the database so it will get a LINKSTRUCTURE message when the database or table structure is changed. Returns negative code if block not found. Note that for structure registration, only whole databases or whole tables can be registered. If databaseIndex is negative, register all databases for structure changes. If tableIndex is negative, register all tables for structure changes.	I
DBBlockUnregisterContent(integer blockNumber, integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	For non-user-linked data, take it out of the registered block list. Return negative code if block not found. If databaseIndex is negative, unregister all databases. If tableIndex is negative, unregister all tables. If fieldIndex is negative and recordIndex is negative, unregister all cells in table. If fieldIndex is negative and recordIndex is good, unregister whole record. If fieldIndex is good and recordIndex is negative, unregister whole field.	I
DBBlockUnregisterStructure(integer blockNumber, integer databaseIndex, integer tableIndex)	For non-user-linked data, take it out of the registered block list. Return negative code if block not found. If databaseIndex is negative, unregister all databases. If tableIndex is negative, unregister all tables.	I
DBDatatable(integer blockNumber, string datatableName, integer databaseIndex, integer tableIndex, integer showFieldNames)	Link a data table with an ExtendSim database table.	I

DB linking/notify	Description	Return
DBGetLinkedContent-List(integer DBIndex, integer TableIndex, integer Array-BlockID[][1], integer ArrayDBLoca-tion[][4])	For blocks that have database **content** registered, returns the number of links found, changes the dynamic arrays to have a number of rows equal to the number of links, and puts their link information into dynamic arrays. *ArrayBlockID[][1]* has one column that contains the block-Number. *ArrayDBLocation[][4]* has four columns that con-tain the 4 database indices (DB, Table, Field, Record) needed to specify the location of the cell linked. If the Field index is -1, the whole table is linked to a data table dialog item. By default, only lists the blocks which are registered to data-base *DBIndex* and table *TableIndex*. If DBIndex== -1, it lists all registered blocks in all databases with the entered TableIn-dex. If DBIndex == -1 and TableIndex == -1, it lists all blocks which are registered to any databases and any tables.	I
DBGetLinkedDialog-sList(integer DBIn-dex, integer TableIndex, integer ArrayBlockAndDia-logID[][2], integer ArrayDBLoca-tion[][4])	For blocks that have a dialog item that is dynamically linked to a database location, returns the number of links found, changes the dynamic arrays to have a number of rows equal to the number of links, and puts their link information into dynamic arrays. *ArrayBlockAndDialogID[][2]* has two columns. The first is the blockNumber and the second is the dialogID that is linked. *ArrayDBLocation[][4]* has four columns that contain the 4 database indices (DB, Table, Field, Record) needed to specify the location of the cell linked. If the Field index is -1, the whole table is linked to a datatable dialog item. By default, the arrays list the blocks that have a link just to database *DBIndex* and table *TableIndex*. If DBIndex== -1, they list all links in all databases with the entered TableIndex. If DBIndex == -1 and TableIndex == -1, they list all blocks which are linked to any databases and any tables.	I
DBGetLinkedStruc-tureList(integer DBIn-dex, integer TableIndex, integer ArrayBlockID[][1], integer ArrayDBLoca-tion[][2])	For blocks that have database **structure** registered, returns the number of links found, changes the dynamic arrays to have a number of rows equal to the number of links, and puts their link information into dynamic arrays. *ArrayBlockID[][1]* has one column that contains the block-Number. *ArrayDBLocation[][2]* has two columns that contain the two database indices (DB, Table) needed to specify the location of the table linked. By default, the arrays list the blocks that have a link just to database *DBIndex* and table *TableIndex*. If DBIndex== -1, they list all links in all databases with the entered TableIndex. If DBIndex == -1 and TableIndex == -1, they list all blocks which are linked to any databases and any tables.	I

DB linking/notify	Description	Return
DBParameter(integer blockNumber, string dialogItemName, integer databaseIndex, integer tableIndex, integer fieldIndex, integer recordIndex)	Link a parameter with an ExtendSim database cell.	I

Arrays, pointers, queues, delay, linked list, and string lookup table 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];
```

☞ 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	Return
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).	V
DynamicArrayIndexByName (long blockNum, string arrayName)	Returns the index number of the named dynamic Array.	I
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
FindMinimumWithThreshhold (realArray y, integerArray 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 threshhold, or above.	R

Dynamic arrays	Description	Return
GetDimension (array)	The size of the first dimension (rows). 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
GetDimensionByName(integer blockNumber, string arrayName)	This function is a variation of the GetDimension function that has two differences. The first is that it takes a block number argument, which specifies which block contains the array, and the second is that it takes the arrayname of the array as an argument rather than the array itself. The combination of these two differences mean that you can call the function from a remote block, query the number of rows in a array, and also pass in a string variable for the arrayname if desired.	I
GetDimensionColumns(array)	Returns the number of columns (second dimension in a two dimensional array) in the specified array.	I
GetDimensionColumnsByName(integer blockNumber, string arrayName)	Has the same behavior as the getDimensionColumns function, with the exception that it can be called from an outside block, and doesn't have to be called from within the block that contains the array. Note that the arrayName argument is the name of the array as a string, not the array name itself. This also means that the function can be called with a string variable as the second argument, and not a hard coded array name.	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).	V
MakeArray2(integer blockNumber, string arrayName, integer dim)	Has the same behavior as the MakeArray function, with the exception that it can be called from an outside block, and doesn't have to be called from within the block that contains the array. Note that the arrayName argument is the name of the array as a string, not the array name itself. This allows resizing of dynamic arrays passed in to functions *as a string name*.	V
SortArray(array, integer numRows, integer keyColumn, integer increase, integer sortStringAsNumbers)	Sorts the array up to *numRows*. Uses the *keyColumn* as the sorting column. To sort a table using multiple keyColumns, call the function multiple times using keyColumns from the lowest to the highest priority. 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.	V

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 104. Also see the "Passing Arrays" exam-

ple (Windows*:* PassArry.mox in the ModLTips folder; Mac OS*:* Passing Arrays in the ModL Tips folder).

Passing arrays	Description	Return
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 104, for more information.	R

Pointer functions

Dynamic array pointer functions allow you to copy the contents of dynamic arrays into an independent data structure that can be created, used by other blocks if desired, and disposed of by any block. You can create a very large number of independent pointers, if desired. They are also useful for creating complex structures of pointers.

Using a disposed pointer will cause a crash (uses malloc() and free()).

Pointers	Description	Return
PointerFromDynamicArray(dynamicArray)	Copies data from any type of dynamic array to a new pointer created by "malloc()" and returns the pointer. If you pass the pointer to an external DLL, the first integer in the pointer's data is the size of the pointer's data in bytes. Including the size data, the pointer is actually that size plus 4.	I
PointerDispose(integer pointer)	Frees the memory used by the pointer using "free()". The pointer is now invalid and can cause a crash if used.	V
PointerToDynamicArray(integer pointer, dynamicArray)	Takes a pointer created by PointerFromDynamicArray() and copies it to a dynamic array, so ModL code can access it. This action writes over any old data in that dynamic array. The dynamic array must be the same type as the array in the pointer. The pointer must still be disposed of by PointerDispose().	V

Global arrays

The Global Array functions define and manage global arrays (see "ModL function overview" on page 206). All of the functions except the ones that start with "GAGet" 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
GAString63	6
GAString127	7

Global Arrays	Description	Return
DBTabletoGA(integer databaseIndex, integer tableIndex, integer GAIndex)	Copies data from the specified Database table to the specified Global Array. The function makes its best attempt to copy the data over. If the DB table defines multiple fields in different formats, not all the data may be copyable.	I
GABlockRegisterContent(integer blockNumber, integer GAIndex, integer row, integer col)	This function is used in conjunction with the Dynamic Linking functionality in version 7. This function will set up a registration entry for the specified Global Array, such that whenever there is a change in the Global Array, the block will receive ContentChanged messages informing it of the change. This works similarly to the way a dialog item linked to the Global Array would work, except that by calling this function you can establish this kind of link through code without a specific dialog item involved. If you specify the row and column, then the message will only be sent when the specific cell is affected by the change. (Specify -1 if you want the entire array.) See the GABlockRegisterStructure() function below to register to receive structure changes.	I

Global Arrays	Description	Return
GABlockRegister-Structure(integer blockNumber, integer GAIndex, integer row, integer col)	This function is similar to the GABlockRegisterContent function, except that it registers to receive StructureChanged messages when the structure of the GA changes, not when the content changes.	I
GABlockUnregis-terContent(integer blockNumber, integer GAIndex, integer row, integer col)	Unregisters the specified GA from the Block's dynamic link registry. See GABlockRegister above for information about what it means for a Global Array to be listed in the registry.	I
GABlockUnregis-terStructure(integer blockNumber, integer GAIndex, integer row, integer col)	Unregisters the specified GA from the Block's dynamic link registry. See GABlockRegister above for information about what it means for a Global Array to be listed in the registry.	I
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.	V
GACopyAr-ray(integer arrayIndex, string newName)	Creates a new Global Array that is a duplicate of the array specified by arrayIndex, with the name *newName*. The return value of the function is the arrayindex of the new Global Array.	I
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 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 quickly.	I
GADataTable (integer blockNumber, string DTname, integer arrayIndex)	Associates a Global Array with a dialog data table in the same way the DynamicDatatable function associates a dynamic array with one. See the DynamicDatatable() function description for more information.	I

Global Arrays	Description	Return
GADelete-Row(integer array-Index, integer row)	Deletes a row in the specified Global Array. This will move down the data on rows after the specified row.	I
GADispose(string name)	Disposes the named Array.	I
GADisposeByIn-dex (integer array-Index)	Disposes the Global Array referenced by *arrayIndex.*	I
GAExport(string pathName, string userPrompt, string format, integer GAIndex, integer rows, integer col-umns)	Exports data from the Global Array specified by GAIndex directly into a text file. See the Export() function for information about the other arguments.	I
GAFindStringAny (integer arrayIndex, string findString, integer column, integer numRows, integer numChars, integer caseSensi-tivity)	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(inte-ger 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 speci-fied *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
GAGetColumns (string name)	Returns the number of columns defined for the named array.	I
GAGetColumns-ByIndex (integer arrayIndex)	Returns the number of columns defined for the Global Array referrenced by *arrayIndex.*	I
GAGetIn-dex(string name)	Returns the Index value of the named array.	I
GAGetInfo(integer ArrayIndex, inte-ger Which)	This Global Array function returns the value of some of the Global Array flags for the specified array. Values for *Which* are: 0:non saving, 1:initializing.	I
GAGetInit-Value(integer arrayIndex)	Gets the initialization value for the global array. This value is set by GASetInitValue(), and is used during array initialization.	R

Global Arrays	Description	Return
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
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
GAGetString63(integer arrayIndex, integer row, integer column)	Supports the string 63 type, otherwise the same as the GAGetString() function.	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

Global Arrays	Description	Return
GAImport(string pathName, string userPrompt, string format, integer GAIndex)	Imports data from a text file directly into the Global Array specified by GAIndex. See the Import() function for information about the other arguments.	I
GAInitializing(integer ArrayIndex, integer Initializing)	Sets the initializing flag for the specified global array. The flag determines if the array is automatically initialized during initsim of a model run or not. The Initializing flag takes the following values: 0: don't initialize (default), 1: initialize to 0, 2: initialize to blank (real numbers only.), 3: initialize to specified value. See GASetInitValue(), below.	I
GAInsertRow(integer arrayIndex, integer row)	Inserts a row in the specified Global Array. This will cause all the data on rows after the specified row to be moved up a row.	I
GALastUsedIndex()	Returns the GAIndex of the last defined Global Array. This can be used to loop through all the Global Arrays in a model.	I
GAMultisim(integer arrayIndex, integer multisim)	Sets a flag on the Global Array that determines how the initialization deals with multiple simulation runs. 0: initialize at beginning of each run, 1: initialize at beginning of first run of a multiple run only.	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
GAParameter(integer blockNumber, StringdialogItemName, integer arrayIndex, integer colIndex, integer rowIndex)	Link a parameter with a Global Array cell.	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 Item library block as an example.	I
GAPtr (integer arrayIndex)	Returns the memory pointer to the data associated with a particular global array. (For passing data to dll's only, and it should be called only immediately before the DLL call, as memory can move, making the pointer invalid.)	I
GAResize(string name, integer rows)	Changes the number of rows defined for a global array.	I

Global Arrays	Description	Return
GAResizeByIndex (integer arrayIndex, integer size)	Changes the number of rows defined for the Global Array referenced by *arrayIndex.*	I
GASearch(integer GAIndex, integer lValue, real rValue, string sValue, integer whichCol, integer startIndex)	Searches a Global array for the occurrence of the specified value and returns the first index where found. This function can be used to search any type of Global Array. It will search for the value lValue, rValue, or sValue, as appropriate, based on the type of the Global Array and will search in the column specified by the whichCol parameter. The startIndex parameter specifies which row to begin searching on. For second and subsequent searches, just pass the last value returned by the function *plus one* as the startIndex parameter. You should end the search when the function returns a negative 1, as this will mean that the desired element was not found.	I
GASearch-Count(integer GAIndex, integer lValue, real rValue, string sValue, integer whichCol, integer startIndex)	Returns the number of occurrences in a Global array of the specified value. This function can be used to search any type of Global Array. It will search for the value lValue, rValue, or sValue, as appropriate, based on the type of the Global Array and will search in the column specified by the whichCol parameter. The startIndex parameter specifies which row to begin searching on. You should end the search when the function returns a negative 1, as this will mean that the desired element was not found.	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
GASetInit-Value(integer arrayIndex, real value)	Sets the initialization value for the specified global array. This value is used during array initialization.	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
GASetLong(integer value, integer arrayIndex, integer row, integer column)	This function is alos known as GASetInteger. See GASetInteger for description.	I
GASetReal(real value, integer arrayIndex, integer row, integer column)	Sets the real value at the specific row and column of the array with the specified index.	I

Global Arrays	Description	Return
GASet-String(string value, integer arrayIndex, integer row, integer column)	Sets the string value at the specific row and column of the array with the specified index.	I
GASet-string15(string value, integer arrayIndex, integer row, integer column)	Sets the str15 value at the specific row and column of the array with the specified index.	I
GASet-string31(string value, integer arrayIndex, integer row, integer column)	Sets the str31 value at the specific row and column of the array with the specified index.	I
GASet-String63(string value, integer arrayIndex, integer row, integer column)	Supports the string 63 type, otherwise the same as the GASet-String() function.	I
GASort(integer arrayIndex, integer numRows, integer keyColumn, integer increase, integer sortstringAsNumbers)	Sorts the Array with the specified index. 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*.	I
GAtoDBTable(integer GAIndex, integer databaseIndex, integer tableIndex)	Copies data from the specified GA to the specified Database table. The function makes its best attempt to copy the data over. If the DB table defines multiple fields in different formats, not all the data may be copyable.	I

Linked lists

Linked lists are queue-like, multiple type structures that maintain internal pointers between the different elements. This speeds the moving around of elements (sorting) within the list. Each structure element can simultaneously contain any number of integer, real, Str15, Str31, and Str255 data types, so complex sorted structures can be created. They will be slightly slower to access than their linear equivalent if used as a simple queue (FIFO, or LIFO) and faster than their linear equivalent if their internal sorting functionality is taken advantage of, as in a Queue block (Item library). See that block's code for a good example of using linked lists to sort.

These functions create and manipulate linked lists which are referred to by an index and are associated and stored with a block. Because these functions have a global block number as an

argument, linked lists associated with a specific block can also be accessed globally, from any other block in the model.

The normal sequence for working with linked lists is:

```
ListCreate(...); // create the list structures
ListCreateElement(…); // creates a new empty element
ListSetxxx(…); // sets a field in the element
ListSetxxx(…); // sets another field in the element
… // set the rest of the fields in the element
ListAddElement(…); // adds the new element to the list
... // manipulate the list, adding and deleting elements
... // get elements from the list to use in a calculation
ListDispose(...); // we are done with the list
```

Linked lists	Description	Return
ListAddElement(integer blockNumber, integer listIndex, integer where)	Adds an element previously created with ListCreateElement to the specified queue. *where*:-2 sorted by preset sortType and field index value *where*:-1 the front of the list Otherwise, *where* is an index value and the item will be added after the specified index item. Returns zero for success.	I
ListAddString63s(integer blockN, integer listIndex, integer string63Count)	This function should be called right after ListCreate, if you wish your linked list to contain String63s. It has the same effect as specifying, for example, n String15s in the ListCreate function, it defines the number of string63s that will be present in each element of the Linked List.	I
ListCopyElement(integer blockN, integer listIndex, integer fromIndex, integer targetBlockN, integer targetListIndex, integer targetIndex)	Copies an element from one linked list to another. The first three parameters specify the element in the first list, the next two specify the target list, and the last one specifies where in the new list to copy the element. As with any linked list function that adds an element, you can specify a -2 to mean that the new element should be added in its sorted order.	I

Functions

Linked lists	Description	Return
ListCreate(integer blockNumber, integer longCount, integer realCount, integer str15Count, integer str31Count, integer strCount, integer sortType, integer fieldIndex)	Creates a new list with the specified attributes. LongCount, realCount, etc are counts of the number of fields of each of the specified types each list element contains. SortType and fieldIndex determine which field is to be used as the sorting field for the list. sortType:0 don't sort sortType:1 real field is key sortType:2 integer field is key sortType:3 str255 field is key sortType:4 str15 field is key sortType:5 str31 field is key FieldIndex is used to determine which index of the specified type is the key field. Zero is the first field.	I

ListCreateElement(integer blockNumber, integer listIndex)	Creates a new empty element for a list. The normal sequence is
	
	ListCreateElement(…); // creates a new empty element
	ListSetxxx(…); // sets a field in the element
	ListSetxxx(…); // sets another field in the element
	… // set the rest of the fields in the element
	ListAddElement(…); // adds the new element to the list
		I

ListDeleteElement(integer blockNumber, integer listIndex, integer indexToDelete)	Deletes the specified element. Zero is the first element.	I
ListDispose(integer blockNumber, integer listIndex)	Disposes of the specified list and recovers its memory.	I
ListDisposeAll(integer blockNumber)	Disposes all linked lists in a block. Returns TRUE if the block doesn't exist or the lists have already been disposed.	I
ListElementMinMax(integer blockNumber, integer listIndex, integer compareType, integer compareIndex, integer max)	This function searches the list for the maximum or minimum value of the specified value. If the Max flag is true, it will return the index of the element that contains the maximum value of that entry, otherwise it will return the minimum. (Currently just integer and real are implemented. string comparisons are not yet implemented.)	I

Linked lists	Description	Return
ListGetCount(integer blockN)	Returns the count of the number of linked lists the block has defined. Please note that in a similar way to the way GetNumBlocks works for the model worksheet, the number returned by this function can contain 'empty slots'. An 'empty slot' is defined as a list index that specifies a list that has been disposed, or is otherwise not defined. This function can be used to execute a loop that looks at all the linked lists in a block, but you should check each list to confirm that it exists. Because of this aspect of how this functions, you should not call ListGetCount, and assume that the returned value is exactly the number of lists the block supports.	I
ListGetDouble(integer blockNumber, integer listIndex, integer elementIndex, integer fieldIndex)	Returns the real (double) value at that element and field index. If elementIndex is passed in as a value less than zero, it refers to the current newly created, but not yet added, item. If elementIndex is zero or greater it is used as an index value into the specified list.	R
ListGetElements(integer blockNumber, integer listIndex)	Returns the number of elements in the specified list.	I
ListGetIndex(integer blockN, string name)	Gets the index of a linked list by its name. Linked lists do not automatically have names. If the name specified is not found, the function will return a negative value as an error code.	I
ListGetInfo(integer blockNumber, integer listIndex, integer infoType)	Returns the specified info about the specified Linked List. InfoType takes the following values: 1: returns the number of real values in the specified list 2: returns the number of integer values in the specified list 3: returns the number of string values in the specified list 4: returns the number of string15 values in the specified list 5: returns the number of string31 values in the specified list 6: returns the number of string63 values in the specified list 10: returns TRUE if this list exists, FALSE if it doesn’t	I
ListGetLong(integer blockNumber, integer listIndex, integer elementIndex, integer fieldIndex)	Returns the integer (integer) value at that element and field index. If elementIndex is passed in as a value less than zero, it refers to the current newly created, but not yet added, item. If elementIndex is zero or greater it is used as an index value into the specified list.	I
ListGetName(integer blockN, integer listIndex)	Returns the name of the linked list. List names are new in version 7. Lists do not have a name by default, and will not have a name, until one has been set with the ListSetName function.	S

Linked lists	Description	Return
ListGetString(integer blockNumber, integer listIndex, integer elementIndex, integer stringType, integer fieldIndex)	Returns the string value at that element and field index. If elementIndex is passed in as a value less than zero, it refers to the current newly created, but not yet added, item. If elementIndex is zero or greater it is used as an index value into the specified list. StringType takes the following values: 3: string (str255) field 4: str15 field 5: str31 field	S
ListLastElementIndex(integer blockN, integer listIndex)	Return the index value of the last item added to the specified list (not the end of the list, the last item actually added).	I
ListLocked (integer blockN, integer listIndex, integer locked)	If Locked is true, this function call marks the specified Linked List as locked. This has the effect of making that List not be disposed if ListDisposeAll is called. If the function ListDispose is called explicitly on this list, it will still be disposed, this function only prevents accidental disposal of the list through the ListDisposeAll call. Returns a zero of the call succeeds. Returns a negative error code value if the function fails.	I
ListSearch(integer blockN, integer listIndex, integer searchType, integer searchIndex, integer lVal, real rVal, string sVal, integer startIndex)	Searches the list for the specified value. Search type determines which type to search for, and also which value string is used. StartIndex specifies where in the list to start searching. This function returns the index number of the first list element that matches the search criteria. If you want to search for multiple elements in the same list, just call the function multiple times, using the last index found *plus one* as the 'startIndex' parameter of the next search. You should end the search when the function returns a negative 1, as this will mean that the desired element was not found.	I
ListSearchCount(integer blockN, integer listIndex, integer searchType, integer searchIndex, integer lVal, real rVal, string sVal, integer startIndex)	Similar to the ListSearch function, except that this function returns the number of occurrences of the specified value in the list.	I
ListSearchCountLongs(integer blockN, integer listIndex, integer Array y, integer startIndex)	Functions similarly to the ListSearchCount function, except that the integer Array Y argument is similar to the ListSearchLongs function, below. (I.e. this function will count the number of elements in the list that contain matches for all the integer elements in the integer Array.)	I

Linked lists	Description	Return
ListSearchLongs(integer blockN, integer listIndex, integer Array y, integer startIndex)	Similar to the ListSearch function, except that the type to be searched for is longs, and the function will search for more than one integer at a time within a single element of the list. The integer Array Y argument is a two column array by any number of rows long. These longs are in pair of index followed by search value. This allows you to search a linked list for more than one integer condition at a time. The search will match only list elements where all the longs in the array match.	I
ListSetDouble(integer blockNumber, integer listIndex, integer elementIndex, integer fieldIndex, real value)	Sets the real (double) value at that element and field index. If elementIndex is passed in as a value less than zero, it refers to the current newly created, but not yet added, item. If elementIndex is zero or greater it is used as an index value into the specified list.	I
ListSetLong(integer blockNumber, integer listIndex, integer elementIndex, integer longIndex, integer value)	Sets the integer (long) value at that element and field index. If elementIndex is passed in as a value less than zero, it refers to the current newly created, but not yet added, item. If elementIndex is zero or greater it is used as an index value into the specified list.	I
ListSetName(integer blockN, integer listIndex, string name)	Sets the name of the specified linked list to the name defined by the ‘name’ parameter.	I
ListSetSort(integer blockNumber, integer listIndex, integer sortType, integer fieldIndex)	ListSetSort allows you to change the sort criteria for the list. SortType and fieldIndex are defined as described above in the ListCreate function. The list will be resorted by this call.	I
ListSetSort2(integer blockN, integer listIndex, integer sortType, integer sortIndex, integer sortType2, integer sortIndex2, integer sortType3, integer sortIndex3)	Sets the sorting criteria for the specified list. This function enhances the existing ListSetSort() function in that there are now multiple sorting criteria. I.e. there is a secondary, and tertiary sort.	I
ListSetString(integer blockNumber, integer listIndex, integer elementIndex, integer stringType, integer stringIndex, string newString)	Sets the string newString at that element and field index. If elementIndex is passed in as a value less than zero, it refers to the current newly created, but not yet added, item. If elementIndex is zero or greater it is used as an index value into the specified list. StringType takes the following values: 3: string (str255) field 4: str15 field 5: str31 field	V

String lookup table functions

String lookup tables are data structures that allow you to associate a string with an integer value. This is used to lookup a string value in the table whenever needed. An example might be associating the strings "Red", "Green" and "Blue" with the values 1, 2 and 3. This allows you to display the string values in a dialog box or popup menu when internally you are storing a numeric value. Because the String Lookup blocks use hash tables internally they will access and convert the integer values to the strings quickly.

These functions are used to implement the string attribute behavior in the Item library.

Note that string lookup table information is stored in the model, but it is not saved when the model is closed. This has a couple of implications. First, if you copy a block that has information that is based on a string lookup from one model to another one where the string lookup is not defined, the results will be unpredictable. The correct work around for this issue would be to make sure that the string lookup is defined in the target model before the block is copied. The second implication is that, as the string lookup information is not saved when the model is closed, you will need to recreate any necessary string lookups in the model when it is opened. (In the case of the String Attributes, this is done in the executive block during the OpenModel message handler.) Also, see "Strings" on page 358

String lookup	Description	Return
SLClear(string slName)	Clears the specified lookup of all strings, and sets it to not be a string lookup, until SLSet is called on it again.	I
SLCreate(string slName)	Creates a string lookup table with the specified name.	I
SLDelete(string slName)	Deletes a string lookup table with the specified name.	I
SLFlagGet(string slName, integer which)	Returns one of the twenty user defined flag values for the specified string lookup. Each string lookup has twenty flags associated with it. 'Which' should take a value from 0 to 19.	I
SLFlagRealGet(string slName, integer which)	Returns one of the twenty real user defined flag values for the specified string lookup. Each string lookup has twenty real flags associated with it. 'Which' should take a value from 0 to 19.	R
SLFlagRealSet(string slName, integer which, integer tableAttribute)	Sets one of the twenty real user defined flag values for the specified string lookup. Each string lookup has twenty real flags associated with it. 'Which' should take a value from 0 to 19.	I
SLFlagSet(string slName, integer which, integer tableAttribute)	Sets one of the twenty user defined flag values for the specified string lookup. Each string lookup has twenty flags associated with it. 'Which' should take a value from 0 to 19. Note that these flags are stored internally as a single byte of data, which means that you can only store values from zero to 127 in the flag. This is normally intended to store Boolean (True/False) values, but you can store values up to 127 if you wish. (Negative values will not be stored.)	I
SLGetCount()	Returns the count of the number of string Lookups defined in a model.	I

String lookup	Description	Return
SLGetCount-Strings(string slName)	Returns the count of the number of strings that are associated with the specified lookup.	I
SLIs(string slName)	Returns a true value if the specified name is a string lookup.	I
SLPopup-Menu(string slName, dialog-Item, init, flying)	Either fills a popup menu with the lookup string values, or creates a flying popup menu at the location of the last click, based on whether or not the flying flag is set.	I
SLSort(string slName)	Sorts the strings for the specified lookup alphabetically.	I
SLStringAp-pend(string slName, string stringVal)	Sets a string value for the specified lookup. By default the first string will have value one, the second one value two, and so on. This can be revised by changing the order of the list by either calling SLSort, to sort the list alphabetically, or calling SLRemove to remove a string from the list. The first time this function is called, It will create a string lookup table with the name slName if that string lookup table does not already exist. This is an alternative to calling SLCreate.	I
SLStringGet(string slName, integer index)	Returns the string value of the specified index on a string lookup.	S
SLStringGetIn-dex(string slName, string string)	This function is the inverse of the SLStringGet, it returns the index value of a string on a string lookup.	I
SLStringInsert(string slName, string-stringVal, integer index)	Inserts the specified string at the specified index.	I
SLStringRe-move(string slName, string string)	Removes the specified string from the list of strings associated with the specified lookup.	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 cannot be directly accessed via the array and an index. You must use the following functions to access elements within the queue. Also, see “Linked lists” on page 349.

Queues	Description	Return
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.	V
PutRear(real array[], real x)	Adds *x* to the rear of the queue.	V
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.	V
QueLength(real array[])	Length of the queue. If the queue is empty, the function returns 0.	I
QueSetAlloc (integer alloc, integer 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.	V
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.	V

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 MakeArray 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	Return
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 DelayTime)	Initializes a dynamic array delay line to *DelayTime*. Call this procedure in the InitSim message handler for each delay line.	V

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.

Please see "String lookup table functions" on page 355

Strings	Description	Return
ArrayLabelParse(integer item, string array)	Parses an array of semicolon delimited strings and treats it as a single long string so you can get the nth label. This is the same parsing that ExtendSim does internally with both the data table labels, and the popup menu strings.	S
DIFontSize(integer blockNum, string DIName)	Returns the point size of the font used by the named dialog item.	R
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
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

Strings	Description	Return
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. NOTE: if -1 is used for maxchar, format is ignored and no scientific notation is used even if the value is very small or large. This special output format is needed to be compatible with Proof Animation.	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
RealToStrShortest(real value, integer sigFigs, integer alwaysPadZeroes)	Converts a double or real variable to a string value, like the RealToStr function. The alwaysPadZeros argument specifies if you want the zero trimming behavior. What this does is to remove any trailing zeros that may have appeared in the string from the sigfigs being higher than the number of actual digits in the resulting value. If you call this function with alwaysPadZeros TRUE, it will behave exactly the same as RealToString.	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

Strings	Description	Return
StringTrim (string input, integer which)	This function is used to trim blank spaces (including CR, LF, and TAB characters) off the input string. The resulting string is returned. The argument "which," takes the following values: 0-both leading and trailing blanks are trimmed. 1-leading blanks trimmed. 2-trailing blanks trimmed. 3-both leading and trailing blanks are trimmed plus blanks anywhere else in the string 4-empty spaces (not including CR, LF and TAB) are replaced with the _ character	S
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 255 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, real size)	This function returns the width that the specified string would draw at in pixels. If font, face, and size are all zero the function will use the default values (Arial 9 point) for the animation text functions. Font values: 0: Arial 2: New York 20: Times 21: Helvetica 22: Courier any other number: Arial Face values: Bold: bit 31 is 1 Italic: bit 30 is 1 Underline: bit 29 is 1 Size is the point size of the font.	I

Strings	Description	Return
StrPartDynamic(stringarray, integer start, integer numChars)	Same as the strPart function, on a dynamicText dialog item. See "Dynamic text items" on page 283.	S

Attributes

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

Obsolete. 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	Return
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

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 ExtendSim User Reference). GetTimeUnits and ConvertTimeUnits use the following values for Time Units.

Time Unit	Value
Generic Time Units	1
Milliseconds	2
Seconds	3
Minutes	4
Hours	5
Days	6
Weeks	7
Months	8
Years	9

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

Calendar Date functions

ExtendSim Calendar Date values are numeric representations of a date value that is the same as is used by Microsoft Excel, and is based on the number of days since January 1, 1900 and the fraction of the current date for the time value. The integer part of the number is the number of days since 1900, and the decimal part of the number is the fraction of the current day. For example, the number 37256.5 would be twelve noon, on January first, 2006. There is a Macintosh version of this date calculation, (also implemented based on the same numbers as Excel,) which bases the first part of the date number on January first, 1904. Make sure that if you are communicating with an outside application, you have selected the same version of these date choices in both applications.

If you have a model saved in ExtendSim, and it has saved date values in one of these formats, running it in the other date setting will give unexpected results. This legacy Macintosh date setting is selectable in the Simulation Setup Dialog. Calendar Dates can only be selected in the simulation setup dialog if the time units for the model are set to seconds or longer, and not to generic time units, or milliseconds.

☞ Prior to ExtendSim 7, a different time and date format was used, as discussed on page 364.

Date and time	Description	Return
EDCalcDate(integer year, integer month, integer day, integer hour, integer minute, integer second)	Construct a date value from its individual components.	R
EDCalendarDateGet(real startDate)	Opens the calendar input dialog for the user to input a date value and returns that date value. The input dialog will show the value of the parameter startDate as a starting point.	R

Date and time	Description	Return
EDCalendarDates()	Returns the value of the CalendarDates checkbox in the simulation setup dialog box: FALSE is unchecked, TRUE is checked.	I
EDCalendar-Show(true/false)	This function opens or closes the calendar window.	I
EDConvert-Date(real value, integer fromType, real startDate)	Modifies a date value by adding additional time. The additional time added to the date value will be in the value parameter, and what time units it is in will be in the fromType parameter. Note that this function returns a date value, not a number of time units.	R
EDDateToSim-Time(real current-Date, integer timeUnits)	Converts from a date value to a simulation time value (e.g. a possible value of currentTime). Putting in a zero for the timeUnits argument will make the function use the currently specified model time units. (see Time Units)	R
EDDateTo-String(real dateV-alue, integer format)	Converts an ExtendSim date value to a string according to format: 0: Date and Time, 1: Just date (ignore Time), 2: Just Time (ignore Date), 3: Tight format (two digit year, don't show time value if zero). 4: Looser format (two digit year but show time value even if zero) The following query the operating system to see if it uses the **European format** where day comes before month: 10: Date and Time, 11: Just date (ignore Time), 12: Just Time (ignore Date), 13: Tight format (two digit year, don't show time value if zero). 14: Looser format (two digit year but show time value even if zero)	S
EDDateValue(real value, integer which)	Gets one part of a date value. *Which* is the time unit value (see "Time units" on page 361, above). NOTE: Week information is not available in this function.	I
EDDayOfThe-Week(real current-Date)	Converts from an ExtendSim Date value to an integer value representing the day of the week. Returns a zero for Sunday.	I
EDGetCurrent-Date()	Returns the current date equivalent for the CurrentTime simulation variable during a simulation run.	R
EDGetStartDate()	Returns the date value of start time in the Simulation Setup Dialog.	R
EDNow()	Returns the real time current date and time like the Now() function, but as an ExtendSim date.	
EDSimTimeTo-Date(real sim-Time, integer timeUnits)	Converts a simulation time value to a date value. A simTime value is a time number in simulation time format. Putting in a zero for the timeUnits argument will make the function use the current model time units. For example in most models, the simTime value for startTime is zero. Calling with a zero value of simTime will return the ExtendSim Date value for the StartDate value of the model.	R

Date and time	Description	Return
EDStringTo-Date(string dat-eString)	Parses a string to retrieve an ExtendSim Date value.	R

Date and time, legacy format

☞ The following date and time functions were used in releases prior to ExtendSim 7. See “Calendar Date functions” on page 362 for the functions used in ExtendSim 7.

The legacy format stored dates and times in an integer that is the number of seconds since midnight, January 1, 1904.

- The Now function returns the current date and time. To determine the amount of time that has elapsed since the beginning of the simulation, subtract Start Time from Now.

Date and time	Description	Return
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
GetBlock-Dates(integer blockNumber, 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 EDdateToString function 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 ExtendSim 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

Timer functions

These allow real time measurements to be set up. The TimerID functions are an extension of the timer functions that allow multiple (Up to 200) timers to run simultaneously. Each timer is referenced by an ID number from 0 to 199. The StartTimer() and StopTimer() functions (without a timerID argument) always refer to timer zero.

Timers	Description	Return
PrecisionTimer()	Returns the current timer count of the highest resolution timer found on the system. Used in conjunction with PrecisionTimerScale(), below.	R
PrecisionTimerScale ()	Returns the numbers of counts per second of the timer made available in the PrecisionTimer function.	I

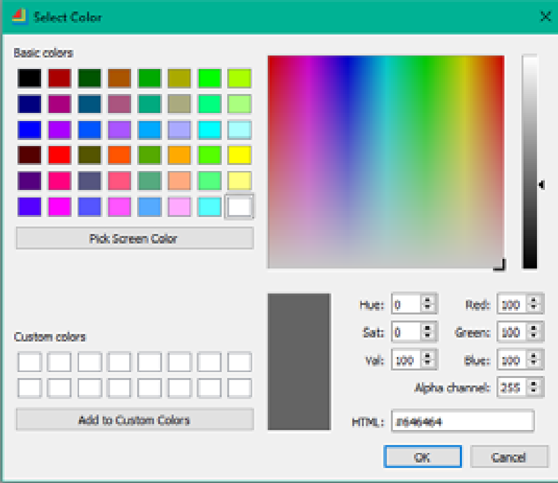
Timers	Description	Return
StartTimer (integer blockNumber, integer waitTicks)	Starts a timer chore that will periodically send messages to either the specified block, or every block in the active model if the blockNumber is zero. This chore is performed when the CPU is idle, so you cannot count on it happening exactly every period, unless the CPU is idle. (e.g. 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 its message every second. The message sent is TIMERTICK. See the Breakout.mox model for an example of timer events.	V
StartTimerID (integer blockNumber, integer waitTicks, integer index)	Starts a timer with an ID tag, which can range from 0 to 199. As with the StartTimer() function, blockNumber specifies which block should receive the TIMERTICK message, and waitTicks specifies how often it should be sent.	V
Stoptimer()	This procedure stops the idle timer chore started by startTimer above.	V
StopTimerID (integer index)	Stops a timer with an ID tag, which can range from 0 to 199.	V
TickCount()	Clock count (in 60ths of a second) since the computer was powered up. This is useful for timing operations.	I
TimerID()	This function is to be used in the TimerTick message handler to find which timer is responsible for triggering the message. Returns the ID number of the timer that is currently activating the message.	I
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.	V

EColors

The following color functions are new in ExtendSim 10. They support a new color information variable type called an EColor, which is stored in long (integer) variables. EColors contain the previously supported RGB/HSV color information as well the alpha channel information that allows color transparency and is new in ExtendSim 10.

Select Color window

You can see any color’s RGB (red, green, blue), HSV (hue, saturation, and value or brightness), and HTML values as well as other settings by selecting the Fill Color tool in the Shapes toolbar:



Color selector

Pick a *basic color* from the matrix of colors. Or use your cursor and the button labeled *Pick Screen Color* to choose a color from the color swatch at the right. Adding any selected color to the custom colors section saves the color for other models. Use the slider at the far right to increase or decrease the value (brightness) of a selected color.

The alpha channel is for the level of transparency; the lower the number, the more transparent the color is.

Some functions in ExtendSim releases prior to 10 had the option of selecting a pattern in addition to a color. Patterns remain as arguments for those functions but are no longer supported.

EColors	Description	Return
EColorFromHSV (h, s, v, alpha)	Converts HSV values to EColor values. H (Hue) should be in the range from 0 to 359. All other values must be in the range 0 to 255	I
EColorFromOldExtendColor (old)	Converts a color value from ExtendSim 9 or earlier to an EColor value.	I
EColorFromRGB (r, g, b, alpha)	Converts RGB values to EColor values. All values must be in the range 0 to 255.	I
EColorIsValid (colorValue)	Returns a true value if the EColor is a valid color, and a false (0) value if it is not.	I
EColorToHTML(integer eColor)	Returns the string representing the hexadecimal value of the EColor.	S

EColors	Description	Return
EColorPart (color, which)	Returns an integer value for the specified part of the EColor value. *which* can take on the following values: 1: red (0-255) 2: green (0-255) 3: blue (0-255) 4: hue (0-359) 5: saturation (0-255) 6: value (0-255) 7: alpha (0-255)	I
EColorPicker (colorValue)	Displays a color picker and returns an EColor value from the user selected result.	I
EColorUpdateValue (old, isHue)	Converts an ExtendSim 9 or earlier color component value to a part of an EColor value. (For example, if you have a number from some v9 code that supports v9 RGB values, calling this function on each one of those separate values will convert them to an EColor component value.)	I

Debugging

Also see the Abort statement in “Control statements and loops” on page 70 and UserError in “Alerts and prompts” on page 248.

Note: To stop a simulation such that it neither puts up an error message nor opens windows to indicate in which process the simulation was stopped, put the following code in the SIMULATE message handler of an equation-based or custom block:

- currentStep = numSteps;
- currentSim = numSims;
- //don’t use the Abort statement or call any abort functions

Debugging	Description	Return
AbortAllSims()	Aborts the simulation and all multiple simulations. Note that the Abort statement only aborts the current simulation.	V
AbortAllSimsSilent()	Aborts a multiSim run without an error message.	I
AbortSilent()	Aborts the simulation run without giving any error messages.	V
DebuggerBreakpoint(integer 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 CreatBlock message and in putting a global breakpoint for all blocks of a type.	V
DebugMsg(string errorString)	Operates like the UserError function. The difference is that this function flags the block as having debugging code – the next time a library is opened that contains any blocks having this function, ExtendSim will issue a warning message. This function automatically displays the simulation time and the block number of the block from which the function was called.	V

Debugging	Description	Return
DebugWrite(integer fileNum, string errorString, delimStr, integer tabCR)	Operates like the FileWrite function. The difference is that this function flags the block as having debugging code – the next time a library is opened that contains any blocks having this function, ExtendSim will issue a warning message.	V
FreeMemory(integer memoryType)	Returns the amount of memory available to ExtendSim. The *memoryType* argument determines what type of memory will be checked. Windows: 1 - Total memory 2 - Resources (only for Windows 3.1 and Windows 95) Mac OS: 1 - Total memory 2 - Contiguous memory	I
heapCheck()	Posts an error message, and returns a true value for an error condition if memory has been corrupted. Don't call this function unless you are in a debugging situation, as heap checking is slow.	I
PauseSim()	Immediately pauses the simulation until you choose Run > Resume, click the Resume button, or call ResumeSimulation(). For continuous process models, see PauseSimForSave() which allows the application to finish sending onSimulate messages until current step is finished executing.	V
ProfileBlockGet(integer blockNumber)	Returns the block profile results for the specified block. This will only return a meaningful number if the command Profile Block Code is selected. Can be called in finalCalc to get the total profile results for that block or during the simulation to get partial results.	R
SelectBlock(integer trueFalse)	Selects the block and scrolls to it if the argument is TRUE, unselects it if the argument is FALSE.	V
SelectBlock2(integer block, integer trueFalse)	This is same as SelectBlock(), except it refers to a global block number.	V
TraceModeEnableDisable(integer enableIfTRUE)	Call to enable or disable the current trace when desired. Trace mode must be on for this function to work. It returns zero if no error and -1 if trace mode is not on.	V

Web and Help connectivity

See also the ShowFunctionHelp function which brings up a list of ExtendSim's functions and arguments. This function is listed with "Equations" on page 218.

Help	Description	Return

CallHelp(string fileName, integer command, integer data, integer file-Type)	Used to load a WinHelp file, an HTML Help file, or a pdf file.
	(For the WinHelp and HTMLHelp Windows API calls, see the Microsoft documentation for more information.)
	The fileType flag takes the following values:
	0: Calls the WinHelp function (Windows only. Opens compiled Help files. Typically have an extension of .hlp.)
	1: Calls HTMLHelp. (Opens compiled Html help files. Typically have an extension of .chm.)
	2: Opens a file with a .pdf extension.
	This function is used to bypass the standard ExtendSim Help system via the following code in the "on helpbutton" message handler:
	`On helpbutton`
	`{`
	`CallHelp("C:\helpfile.chm", 1,1,1);`
	`Abort;`
	`}`
		V

OpenURL(string theURL)	Access the specified URL using your computer's default browser. Returns non-zero error code if it fails.	I
ShowBlockHelp (integer block)	Opens the Help dialog, showing the online 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.	V
ShowHelp()	Opens the Help dialog, showing the online Help for the block.	V

Platforms and versions

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

Platforms and versions	Description	Return
GetCurrentPlatform()	Determines the operating system under which ExtendSim is currently running. This function returns 0 for 68K Mac OS, 1 for Power Mac OS or Mac OSX, 2 for Windows 3.1/Win32s, 3 for Windows NT or XP, and 4 for Windows 95, 98, and ME.	I

Platforms and versions	Description	Return
GetExtendVersion(integer which)	If which is 1 or 2 it returns a number in the format 1025.1 where 10 is the major version, 2 is the middle, 5 is the minor version, and the digit after the decimal is 1 for a, 2 for b, and so on (e.g., version10.2.5a). If which is 1, the final digit is always zero. If which is 2, it returns True if it is a debug version. Which: 0: application version 1: file version 2: debug version	R
GetExtendVersionString()	Returns the version number of ExtendSim as a string.	S
GetExtendType()	Returns the type of the ExtendSim application. Normal: 0 LT/RunTime: 2 Demo/Player: 4	I
GetFileReadVersion()	Returns 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 ExtendSim that a data table has changed size from the size it was in an older version. (This is useful for using the Dynamic data table 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. NOTE: 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 than version 4.0.	

Application privileges

Functions to allow specific application features, such as for Distributed Analysis.

Platforms and versions	Description	Return
MaintenanceSupportPlanExpired()	Returns TRUE if the Maintenance & Support Plan (MSP) has expired, FALSE if it has not yet expired. Gets the MSP expiration date from the license file (extendsim.lic).	I
NumMultiLaunchesAllowed()	Returns the total number of run instances allowed. If MSP is current (i.e. MaintenanceSupportPlanExpired() is False), the base number is 4. If additional run instances have been purchased, returns the total (base number plus number of additional instances purchased).	I

User-defined functions for ADO

User-defined functions are custom functions, coded in ModL, that can be declared in a block for local use or declared in an include file for use by multiple blocks. For how to create, use, and override them, see “User-defined functions” on page 72 and “Include files” on page 81.

The following ActiveX Data Object (ADO) functions are used to implement ADO features in the Data Import Export block (Value library). These ModL-coded functions are stored in an include file named ADO_DBFunctions.h. To reference that include file in your block’s code, use a format discussed on page 81, such as #include “ADO_DBFunctions.h”.

ADO Function	Description	Return
ADO_AddRecords(integer ADOApp, string ADO_TableName, integer EX_DBIndex, integer EX_TableIndex	Inserts an ExtendSim table into an ADO database table. ADO_TableName - name of the table in the ADO database EX_DBIndex - ExtendSim Database index EX_TableIndex - ExtendSim table index Note 1: The tables must have exactly the same structure. Note 2: This is the fastest way to transfer information.	V
ADO_CheckCompatibleFieldType(string ADODataType, integer ExtendSimType)	Returns True (1) if the ADO field type is compatible with the ExtendSim field type.	I
ADO_Close(integer ADOAppHandle, integer Force)	Closes the connection to the ADO DLL. Call when done accessing the DLL.	I
ADO_CreateTable(integer ADOApp, string ADO_TableName, string63 ADO_FieldArray[][4])	Creates a table in an ADO database. ADO_TableName - name of the table ADO_FieldArray contains the table names, their type, “Is nullable”, and the number of characters	V
ADO_DeleteRecords(integer ADOApp, string ADO_TableName, string Criteria)	Deletes records from an ADO database table. ADO_TableName - name of the table in the ADO database. Criteria - SQL statement indicating which rows to delete. To delete all rows, leave blank.	V
ADO_ExecuteNonQuery(integer ADOApp, string SQLStr)	Executes a Non-query SQL statement. SQLStr - SQL statement	V
ADO_ExecuteQuery(integer ADOApp, string SQLStr)	Executes a Query SQL statement. SQLStr - SQL statement	V

ADO Function	Description	Return
ADO_GetFields(integer ADOApp, String ADO_TableName, string63 ADO_-FieldArray[][4])	Gets a list of fields in the database. ADO_TableName - name of the table ADO_FieldArray contains the table names, their type, “Is nullable”, and the number of characters. This is returned by the function	V
ADO_GetNumFields(integer ADOApp, string63 ADO_TableName)	Gets the number of fields in table ADO_TableName. ADO_Tablename - name of the table in the ADO database	I
ADO_GetNumRows(integer ADOApp, string63 ADO_TableName)	Gets the number of records in table ADO_TableName. ADO_Tablename - name of the table in the ADO database	I
ADO_GetNumTables(integer ADOApp)	Returns the number of tables in an ADO Database	I
ADO_GetTableColumns(integer ADOApp, integer EX_DBIndex, integer EX_TableIndex, string ADO_TableName, string63 ADO_-Columns[])	Transfers a set of columns to an ExtendSim database table. EX_DBIndex - ExtendSim Database index EX_TableIndex - ExtendSim table index ADO_TableName - name of the table in the ADO database ADO_Columns - array containing the column (field names) to import into the ExtendSim data table Note 1: Allocate records in the ExtendSim table before calling this function. Note 2: The data types for the fields in the ExtendSim table and the ADO table must be compatible.	V
ADO_GetTables(integer ADOApp, string63 ADO_FieldArray[])	Gets the list of tables in the ADO database. ADO_FieldArray contains the table names, their type, “Is nullable”, and the number of characters.	V
ADO_OpenConnection(string DatabaseType, string FileName, string UserName, string Password, String Server)	Opens a connection with an ADO database. This is where database information is specified: DatabaseType - Access, SQLServer, MYSql, or XML Filename - name of the database UserName Password Server - name of the database server (not used in Access or XML)	I

ADO Function	Description	Return
ADO_SetTable-Columns(integer ADOApp, string ADO_TableName, string63 ADO_-Columns[], integer EX_DBIndex, integer EX_TableIndex)	Transfers a set of columns to an ADO database table. ADO_TableName - name of the table in the ADO database ADO_Columns - array containing the column (field names) to import into the ExtendSim data table EX_DBIndex - ExtendSim Database index EX_TableIndex - ExtendSim table index Note: The data types for the fields in the ExtendSim table and the ADO table must be compatible.	V
ADO_Setup()	Sets up the connection to the ADO DLL. Call before accessing the DLL Returns the ADO Application Handle - referred to in other functions as ADOApp.	I
ADO_SQLServer-GetServers(string63 ServerInfo[][4])	Returns a list of SQL Server Servers. ServerInfo - array containing the list of the servers. Allocate this array before calling the function.	V
ADO_SQLSserver-GetDatabases(string63 Server, string63 DBArray[][4])	Returns a list of SQLServer databases. Server - name of the SQL Server DBArray - returns name, size, description of the SQL Server database. The forth column is unused.	V
ConvertADOData-Type(string ADO-DataType)	Converts an ADO field type to an ExtendSim Constant. ADODataType - string containing the type for the ADO type (float, string, int ...)	I
ConvertExtend-SimDataType(integer ExtendSimType)	Converts an ExtendSim constant for data type to SQL string for data type.	S
DB_FieldGetTypeString(integer ExtendSimType)	Returns the string description given an ExtendSim field type.	S

Appendix

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 300). This is functionally the same as selecting the command from the menu. Below is a list of the command numbers that are used with that function.

File Menu	Command Number
New Model	2
New Text File	1601
Open	3
Recent Files (1-5)	1555-1559
Close	4
Save Model	5
Save Model As	6
Print	9
Update Launch Control	1410
Properties	2001
Exit or Quit	1

Edit menu	Command Number
Cut	18
Copy	19
Paste	20
Clear	21

Model menu	Command Number
Hide Connections	2012
Hide Connectors	2071

Model > Connection Line Style sub menu	Command Number
Smart	5004
Right-Angle	5001
Straight	5002

Model > Connection Line Style sub menu	Command Number
Free Form	5003
Output to Input	1208
Input to Output	1209
Solid Line	1205
Dotted Line	1206
Default Thickness	1250
Double Line	1204

Database Menu	Command Number
Read/Write Index Checking	1931

Run Menu	Command Number
Run Simulation	6000
Simulation Setup	6001
Run Optimization or Scenarios	6002
Show 2D Animation	2020
Stop	30000
Pause	30001
Resume	30002
Step	30003

Run > Model Debugging Menu	Command Number
Step Entire Model	2023
Step Each Block	2024
Generate Trace	2040
Add Selected Blocks To Trace	2041
Add All Blocks To Trace	2045
Remove Selected Blocks From Trace	2042

Run > Model Debugging Menu	Command Number
Remove All Blocks From Trace	2043
Show Tracing Blocks	2044
Profile Block Code	2029

Toolbars	Command Number
Slower Animation (Model tool)	30004
Faster Animation (Model tool)	30005
Normal Size (Edit tool)	2010
Reduce to Fit (Edit tool)	2009
Courier (Text tool)	6100
Helvetica (Text tool)	6101
Times (Text tool)	6102
Arial (Text tool)	6103

Appendix

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, ExtendSim has its limits. It is unlikely you will find them in your normal work, but it is good to know what they are. Note: Some limits depend on available memory.

Steps in a simulation run	2 billion
Total model data size	2.8E14 bytes
Number of simulations in a multiple run	2 billion
Block name or label length, characters	31
Blocks in a model	2 billion
Blocks in a library	200
Libraries open at one time	40
Text files open at one time	200
Databases per model	2 billion
Tables per database/fields per table/records per table	2 billion/1,000/2 billion
Output connectors in a model (nodes)	2 billion
Connectors per block	255
Length of a block's ModL code (characters)	10 megabytes
Dialog items in a block	1024
2D animation objects per block	255/icon; unlimited through function
Dynamic arrays (each array)	2 billion elements
Number of array dimensions	5
Maximum index for array dimensions	2 billion elements total
Dynamic arrays per block	255
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
Variable name length: dialog item msgs, connector names	63
Variable name length: ModL local, static variables	127
Static and local variable declaration limit (See page 62)	32,767 bytes
Maximum popup menu length	5100 characters or 20 strings
User defined function arguments	127
Nested loops	32
Maximum, minimum of real numbers	±1E±308

Maximum, minimum of integer numbers	±2,147,483,647
Significant figures in real calculations	16 (double)
Number of attributes for discrete event item	500

Appendix

ASCII 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 Windows and Mac OS. 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

Index

Symbols

187
_(underscore) character 114
_leftClickDB 100, 102
** 33
*/ 33
/* 33
// 33
#define 83
#else 78, 83
#endif 78, 83
#ifdef 78, 82
#ifndef 83
% operator 69

Numerics

2D animation
- adding to a block 53
- between blocks 134
- changing a level 132
- color 130, 135
- functions 250
- hiding a shape 131
- hierarchical blocks 131
- moving a shape 132
- objects 10, 53
- overview 10
- pictures 134
- pixels 136
- programming 128
- shapes 129
- showing a shape 131
- stretching a shape 133
- text 136

A

Abort statement 70, 146
- in CheckData 72
- in Simulate 72

AbortAllSims 146, 367
AbortAllSimsSilent 367
AbortDialogMessage 200
aborting multiple simulations 142, 146
AbortSilent 367
absolute value 208
Acos 209
ActivateApplication 300
ActivateModel 196
ActivateWorksheet 300
ActiveX
- Automation 119
- BlockMsg 123
- C++ examples 120
- client 119
- Execute 121
- GetObjectHandle 124
- Poke 122
- Request 122
- Visual Basic 125
- Visual Basic examples 125

ActiveX Automation 119
ActiveX Data Objects functions 371
Add External Code to Libraries command 84
Add to Custom Colors button 366
add to right click menu 19
AddBlockToClipboard 300
AddBlockToSelection 300
AddC 210
ADO functions 371
ADO_AddRecords 371
ADO_CheckCompatibleFieldType 371
ADO_Close 371
ADO_CreateTable 371
ADO_DeleteRecords 371
ADO_ExecuteNonQuery 371
ADO_ExecuteQuery 371
ADO_GetFields 372
ADO_GetNumFields 372
ADO_GetNumRows 372
ADO_GetNumTables 372
ADO_GetTableColumns 372
ADO_GetTables 372
ADO_OpenConnection 372
ADO_SetTableColumns 373
ADO_Setup 373
ADO_SQLServerGetServers 373
ADO_SQLSserverGetDatabases 373
AdviseReceive 203
alert functions 248
AlignConnection 261
alignment of dialog item labels 20
alpha channel 366
Alt key 249

alternate views 9
animation
  2D 128
  2D functions 250
animation object (2D) 129
Animation Object button 10
animation object tool 22
animation objects 10
  Properties dialog 129
  zOrder 54
AnimationAntialias 251
AnimationBlockToBlock 134, 251, 256
AnimationBorder 251
AnimationBorderColor 251
AnimationBorderEColor 251
AnimationColor 251
AnimationEColor 129, 135, 251
AnimationGetHeight 252
AnimationGetHeightR 252
AnimationGetLeft 252
AnimationGetLeftRelative 252
AnimationGetLeftRelativeR 252
AnimationGetSpeed 252
AnimationGetTop 252
AnimationGetTopRelative 252
AnimationGetTopRelativeR 252
AnimationGetWidth 252
AnimationGetWidthR 252
AnimationHide 131, 252
AnimationLevel 132, 253
AnimationMoveTo 132, 253
AnimationMovie 253
AnimationMovieFinish 253
AnimationObjectCopyData 253
AnimationObjectCreate 129, 253
AnimationObjectDelete 253
AnimationObjectExists 254
AnimationObjectExists2 254
AnimationOn 130, 190
AnimationOval 129, 254
AnimationPicture 254
AnimationPixelRect 254
AnimationPixelRectEColorInit 254
AnimationPixelSet 254
AnimationPixelSetEColor 254
AnimationPoly 128, 129, 254
AnimationRectangle 254
AnimationRndRectangle 129, 255
AnimationSetDelayMode 255
AnimationSetSpeed 255
AnimationShow 129, 131, 255
AnimationStatus 196
AnimationStretchTo 133, 255
AnimationText 136, 255
AnimationTextAlign 255
AnimationTextSize 255
AnimationTextTransparent 136, 255
AnimationZOrderGet 255
AnimationZOrderSet 256
AntitheticRandomVariates 190
API 2
AppendDataTableLabels 276
AppendPopupLabels 268
application.ini file 79
ApplicationFrame 301
arguments
  arrays as 67, 74
  call tips 79, 207
  data type expectations 207
  of function calls (converting) 64
  pass by value or reference 104
array segment 67
ArrayDataMove 340
ArrayLabelParse 358
arrays 65
  array segment 67
  as arguments 67, 74
  copying using FOR loops 110
  definition 30
  dimensions 65
  disposing 106
  disposing of global 109
  dynamic 26, 66
  fixed 26, 66
  fixed or dynamic 30
  functions 340
  global arrays 67, 109
  import function 110
  in discrete event modeling 147
  memory usage 103
  passing 105, 341
  passing (precautions) 106
  subscripts 65
  working with 103
ASCII value table 384

Asin 209
ASP license
  form-based interface 125
assignment operators 68
Atan 209
Atan2 209
AttribInfo 202
AttribType array 152
AttributeList array 152
attributes
  arrays for items 152
  flow 153
  functions 361
AttribValues array 153
auto indentation 79
automation
  methods 119
automation, OLE
  client 119
  server 119
AutoScaleX 310
AutoScaleY 310

B

BarChart 317
BarChartCategoryCountGet 317
BarChartCategoryCountSet 317
BarChartCategoryGet 317
BarChartCategorySet 317
BarChartSet 317
BarChartValueGet 317
BarChartValueSet 317
BarGraph 310
basic math functions 207
Beep 248
bidirectional connectors 103
Bidirectional Flows model 103
Binomial distribution function 210
BitAnd 218
BitClr 218
bit-handling functions 218
BitNot 218
BitOr 218
BitSet 218
BitShift 218
BitTst 218
BitXor 218
BLANK 63
block number 260
block parts
  accessing from code 35
  animation 8
  code 7
  connector types 21
  connectors 8
  Connectors pane 8
  Dialog Item Names pane 9
  dialog items 8, 14
  Help tab 8
  icon 8
  icon views 8
  ModL code 7
  script 7
  Script tab 12
  tabs 8
block status messages 197
block to block messages 202
BlockAdjustPosition 301
blockAdjustPosition 10
BlockClick 197, 249
BlockDialogIsOpen 288
blocked message 156
BlockIdentify 197
blocking the flow 156
BlockLabel 197
BlockMove 197
BlockMsg 119, 123
BlockName 115, 257
BlockRead 197, 278
BlockReceive 202
BlockReceive0 150
BlockReceive3 150
BlockRect 257
BlockReport 142, 146, 196, 308
BlockRightClick 197
blocks
  2D animation between 134
  adding connectors 46
  animating in 2D 53
  categories 55, 256
  communicating with each other 99
  communication 99, 256, 260, 267, 290
  compiling 47
  data table functions 274

dialog items when creating 35
dialog tab functions 290
dialogs 6, 288
dynamic text functions 283
equation-based 94
Executive 142
for debugging 170
hierarchical (animating) 131
icon view functions 293
icons 9
intermediate results 51
labels 256
labels (length) 380
Make Your Own 146
messaging functions 290
names 256
names (length) 380
new 44
numbers 256, 260
parts of a block 7
popup menus 48
programming discrete event 146
programming discrete rate 165
protecting 90
red border for debugging mode 173
registered 113
remote communication 99
structure 7
submenus in library 55
that don't post future events 149
that post events 149
types of discrete event 148
uncompiled 45
BlockSelect 198
BlockSimFinishPriority 291
BlockSimStartPriority 291
BlockTableInfo 202
BlockUndelete 198
BlockUnselect 198
blue circle 185
Boolean operators 69
Border Color tool for dialog items 17
brace matching 78
Break statement 70, 72
Breakpoint Conditions dialog 187
breakpoints
condition 172
definition 172
disabling 182
margin 174
removing 182
setting 179
setting conditions 180
window 186
Breakpoints window 186
Breakpoint and Condition columns 181
button
creating 52
dialog item 14
message 201
titles (changing) 38, 98
Buttons block 94

C

C to Pascal string function 61
CalcDate
see EDCalcDate 362
CalcFV 213
CalcNPER 213
CalcPMT 214
CalcPV 214
CalcRate 214
calendar 15
calendar date
functions 362
call chain
yellow location arrow 176, 185
Call Chain pane 174
Call Tips 79, 207
CallHelp 369
Cancel 200
case sensitivity 60
CASE statement 72
categories of blocks 55
Ceil 208, 209
CellAccept 200
ChangeAxisValues 310
ChangePlotType 310
ChangePreference 301
ChangePreferenceString 301
ChangeSignalColor 310
ChangeSignalSymbol 311
ChangeSignalWidth 311
changing parameters globally 98
chart functions 308
checkbox

programming for 37
titles (changing) 98
checkboxes 14
CheckData 138, 140, 195
ClearBlock 301
ClearBlockUndo 301
ClearConnection 302
ClearStatistics 202
ClearUndo 302
CloneCreate 302
CloneDelete 302
clone-drop 99
code for a custom stand-alone block 100
CloneFind 302
CloneGetDialogItem 302
CloneGetDialogItemLabel 302
CloneGetInfo 302
CloneGetList 302
CloneGetPosition 303
CloneHideDisable 303
CloneInit 198
CloneResize 303
CloseBlockDialogBox 288
CloseDialogBox 289
CloseEnclosingHBlock 289
CloseEnclosingHBlock2 289
CloseModel 196
ClosePlotter 311
ClosePlotter2 311
CM 84
cm extension 84
code
colorization 78
conventions 49
example 47
language overview 27
layout 32
management 84
script editor 78
type declarations 33, 60
code completion 33, 79
customizing 79
for functions 207
code folding 79
code management 84
code marker 179
CodeExecute 303
coding conventions 49
color 130, 135
HSV 130
color selector window 366
colored code 78
column index 94
column separators 223
column tag functions 284
COM DLL example 125
comments 12
multi-line 33
single line 33
compile
conditionally 29, 76, 82
Compiled_Debug 83
complex number function 210
ConArrayChanged 201
ConArrayChangedComplete 201
ConArrayChangedWhichCon 265
ConArrayCollapseChanged 201
ConArrayGetCollapsed 265
ConArrayGetConNumber 265
ConArrayGetDirection 265
ConArrayGetNthCon 265
ConArrayGetNthCon2 265
ConArrayGetNumCons 265
ConArrayGetOwnerCon 265
ConArrayGetTotalCons 265
ConArrayGetValue 266
ConArrayMsgFromCon 266
ConArraySendMsgToAllCons 266
ConArraySendMsgToInputs 266
ConArraySendMsgToOutputs 266
ConArraySetCollapsed 266
ConArraySetNumCons 266
ConArraySetValue 266
concatenation 68, 69, 249
conditional breakpoints 187
conditional compilation 29, 76, 82
ConjugateC 215
ConnectionBreak 201
ConnectionClick 201
ConnectionMake 198, 202
connector labels
defined 23
formatting 23
connector messages 154, 201

connector names
  used as variables in ModL 34
connector tool tips 267
ConnectorLabelsGet 266
ConnectorLabelsSet 267
ConnectorMsgBreak 291
ConnectorName 202
ConnectorRightClick 198, 202
connectors 34
  adding 22, 46
  bidirectional 103
  changing 102
  changing an input to an output 46
  changing from normal to variable 21
  changing the type of 23
  checking in CheckData 195
  deleting 102
  functions 260
  initializing 102
  labels 23
  messages 154, 159
  naming 22, 34
  no resize bar 21
  normal 22, 35
  normal vs variable 21
  renaming 46
  tools 21
  tooltips 23
  types 21
  User Defined 21
  variable 21, 22, 35, 102
Connectors pane 8, 22
ConnectorShowHide 198
ConnectorToolTip 202
ConnectorToolTipGet 267
ConnectorToolTipSet 267
ConnectorToolTipWhich 267
constant definitions 32
constants 61, 63
  definition 30, 33, 63
  in equation-based blocks 63
  names 60
Continue statement 70, 72
Continue tool 184
ContinueSim 140, 195
control 41
control statements 70, 72
conversion
  of numeric types 64
convert a variable connector 21
ConvertADODataType 373
ConvertExtendSimDataType 373
ConvertTimeUnits 362
CopyBlock 198
copying
  dialog items 97
Cos 209
Cosh 209
cost array 151
costing attributes 153
Create New Library button 44
CreateBlock 50, 198
CreateFolder 224
CreateHBlock 303
CreatePopupMenu 268
CurrentScenario 190
CurrentSense 190
CurrentSim 140, 143, 190
CurrentStep 141, 190
CurrentTime
  changed by Executive block 190
currentTime
  in Debugger window 174
custom colors 366

D

data
  checking in CheckData 195
  consumption by type of variable 61
  linking to databases 55
  linking to global arrays 55
  managing data in complex models 113
  repository 113
  source (organizing) 94
  source indexing 94
data consumption 61
data tables 15, 38
  functions 274
  row and column 18
data type declarations 32
data types 33, 60
  definition 30
Database
  Read/Write Index Checking 318
database errors

no such parent 318
no such record 318
not linked error 319
not unique error 318
not unique index 318
database functions 318
databases (ExtendSim)
_character at beginning of name 114
accessing 113
copying functions 322
creating 113
creating/deleting functions 319
DB address functions 335
functions 318
import/export functions 322
linking/notify functions 337
no such parent error 318
no such record error 318
not linked error 319
not unique error 318
not unique index error 318
number of databases per model 380
properties functions 323
random data functions 331
read/write functions 327
registering blocks 55, 280
reserved 114
selecting functions 321
sort/search functions 334
viewng functions 337
working with 113
DataTableHover 200
DataTableResize 200
DataTableScrolled 200
date functions (legacy) 364
DateToString
see EDdateToString 363
DB_FieldGetTypeString 373
DBAddressCreate 336
DBAddressGetAllIndexes 336
DBAddressGetAsString 336
DBAddressGetDlg 336
DBAddressGetDlg2 336
DBAddressGetFromString 336
DBAddressIncrementIndex 336
DBAddressReplaceIndex 336
DBBlockRegister 113
DBBlockRegisterContent 338
DBBlockRegisterContents 113
DBBlockRegisterStructure 113, 338
DBBlockUnregisterContent 338
DBBlockUnregisterStructure 338
DBChildPopupSelector 321
DBDatabaseCloseViewer 337
DBDatabaseCopy 322
DBDatabaseCreate 319
DBDatabaseDelete 319
DBDatabaseDeleteByIndex 319
DBDatabaseExists 323
DBDatabaseExport 322
DBDatabaseGetIndex 323
DBDatabaseGetIndexFromAddress 336
DBDatabaseGetName 323
DBDatabaseImport 322
DBDatabaseOpenCell 337
DBDatabaseOpenViewer 337
DBDatabaseOpenViewerToTab 337
DBDatabasePopupSelector 321
DBDatabaseRename 323
DBDatabasesGetNum 323
DBDatabaseShowHideReserved 324
DBDatabaseTabChangeName 324
DBDatabaseTabDelete 319
DBDataGetAsNumber 327
DBDataGetAsNumberParentAltField 327
DBDataGetAsNumberUsingAddress 327
DBDataGetAsString 327
DBDataGetAsStringParentAltField 328
DBDataGetAsStringUsingAddress 328
DBDataGetCurrentSeed 331
DBDataGetDateAsSimTime 328
DBDataGetDateAsSimTimeUsingAddress 328
DBDataGetParent 328
DBDataGetParentUsingAddress 329
DBDataSetAsNumber 329
DBDataSetAsNumberReserved 329
DBDataSetAsNumberUsingAddress 329
DBDataSetAsNumberUsingAddressReserved 329
DBDataSetAsParentIndex 329
DBDataSetAsParentIndexReserved 330
DBDataSetAsParentIndexUsingAddress 330
DBDataSetAsParentIndexUsingAddressRe-served 330
DBDataSetAsString 330

DBDataSetAsStringReserved 330
DBDataSetAsStringUsingAddress 330
DBDataSetAsStringUsingAddressReserved 330
DBDataSetCurrentSeed 331
DBDataSetDateAsSimTime 331
DBDataSetDateAsSimTimeReserved 331
DBDataSetDateAsSimTimeUsingAddress 331
DBDataSetDateAsSimTimeUsingAddressReserved 331
DBDatatable 280, 338
DBFieldCreate 319
DBFieldCreateByIndex 319
DBFieldDelete 319
DBFieldDeleteByIndex 320
DBFieldExists 324
DBFieldGetIndex 324
DBFieldGetIndexFromAddress 336
DBFieldGetName 324
DBFieldGetProperties 324
DBFieldGetPropertiesUsingAddress 324
DBFieldMove 325
DBFieldPopupSelector 321
DBFieldRename 325
DBFieldSetInitialize 325
DBFieldSetProperties 325
DBFieldsGetNum 325
DBGetLinkedContentList 339
DBGetLinkedDialogsList 339
DBGetLinkedStructureList 339
DBGetSize 325
DBinomial 210
DBParameter 280, 340
DBRandomDistributionGet 332
DBRandomDistributionSet 333
DBRecordExists 325
DBRecordFind 334
DBRecordFindMultipleFields 334
DBRecordFindMultipleFieldsArray 334
DBRecordFindNumericalRange 335
DBRecordFindParentRecordIndex 335
DBRecordGetIndexFromAddress 337
DBRecordIDFieldGetIndex 325
DBRecordPopupSelector 322
DBRecordsDelete 320
DBRecordsGetNum 326
DBRecordsInsert 320
DBRelationCreate 320
DBRelationDelete 320
DBRelationsGetNames 326
DBRelationsGetNum 326
DBTabGetTableIndexList 326
DBTableCloneToTab 320
DBTableCopy 322
DBTableCreate 320
DBTableCreateByIndex 320
DBTableDelete 321
DBTableDeleteByIndex 321
DBTableExists 326
DBTableExportData 323
DBTableGetIndex 326
DBTableGetIndexFromAddress 337
DBTableGetName 326
DBTableGetProperties 326
DBTableImportData 323
DBTablePopupSelector 322
DBTableRename 326
DBTableSetProperties 326
DBTablesGetNum 327
DBTableSort 335
DBTabletoGA 343
DBToolTipsGet 321
DBToolTipsSet 321
DDE (dynamic data exchange) 229
DE Modeling Using Equation Blocks 300
Debug Tutorial model 173
debugger 172
- arrays 188
- breakpoint margin 174
- breakpoints 172
- call chain pane 174
- Debugger window 174
- indicators in the margin 185
- location arrow 174, 175
- red border around blocks 173
- removing breakpoints 182
- setting breakpoints 179
- setting conditions 180, 187
- source pane 174
- toolbar 184
- tutorial 173
- variables pane 174
- WatchPoint conditions 188
- window 184

Debugger window 174
- empty circle 182
- indicators 185
- toolbar 184

DebuggerBreakpoint 367
debugging 171
- block code using UserError 248
- blocks for 170
- functions 367
- functions using alerts and prompts 248
- models 170
- source code (see "debugger") 172
- tips 188
- tracing 170
- using DebugMsg function 171

DebugMsg 171, 367
DebugWrite 171, 368
decision blocks 148
DEExecutiveArrayResize 202
Delay 358
delay line functions 357
DelayInit 300, 358
DeleteBlock 198
DeleteBlock2 198
delimiters 223
DeltaTime 138, 140, 141, 190
Determinant 215
DeterminantC 215
Develop command 44
DExponential 210
DGamma 210
DI ID 18
diagnostic functions 248
dialog functions 274, 283, 288
Dialog Item Names pane 9
dialog items 6, 15, 35
- add to right click menu option 19
- alignment of labels 20
- Border Color for labels 17
- buttons 14, 38
- buttons (creating) 52
- calendar 15
- changing text and titles 96
- checkbox 14
- checkboxes 37
- color for labels 17
- copying 97
- data tables 38
- display only option 19, 50
- dynamic text 15, 36
- editable text 15, 36
- editable text 31 character limit 15
- Fill Color option 17
- format text 20
- frame 15, 40, 49
- functions 267
- hidden 19
- ID 18
- labels 18
- linking to ExtendSim databases 55
- linking to global arrays 55
- list 14
- messages 36, 199
- meter 15, 41
- move manually 97
- move to another tab 97
- move using code 97
- names 17, 35
- new 44
- numeric formats for 17
- parameter 14, 36, 49
- parameter fields 14
- popup menus 14, 39, 48
- radio button 15, 37, 48
- show and hide 97
- slider 15, 41
- static text label 15, 39
- style 20
- switch 41
- switch control 15
- tab order 18
- tabs 8, 13
- text frame 40
- text tables 15, 38
- tool tips 20, 288
- types (definition) 17
- types of 14
- visible 19
- visible in all tabs option 19
- W and H coordinates 17, 98
- X and Y coordinates 17, 97
- zOrder 17

Dialog Items 2 toolbar 44
dialog messages 199
dialog names
- used as message names 36
- used as variable names 36

Dialog Resizer 13

Dialog tab 8, 13, 44
dialog tab functions 290
dialog variables
getting and/or setting remotely 99
remote interface with 99
DialogClick 200, 249, 268
DialogClose 200
DialogFixedSize 303
DialogGetSize 289
DialogHasEmbeddedObject 268
DialogItem 201
DialogItemRefresh 200
DialogItemToolTip 200
DialogItemVisible 268
DialogMoveTo 289
DialogOpen 200
DialogPicture 256
DialogRefresh 303
dialogs
Dialog Item Names pane 9
exploring 6
functions for opening and closing 288
Help tab 14
programming tips 95
tab for dialog editing 13
tabs 8, 13
text (changing) 95
DialogSetSize 289
DiffDate 364
DIFontSize 358
DIGetID 268
DIGetName 268
DILinkClear 280
DILinkInfo 281
DILinkingDisabled 281
DILinkModify 281
DILinkMsgs 282
DILinkSendMsgs 282
DILinkUpdateInfo 282
DILinkUpdateString 283
dimensions of an array 65
DIMoveBy 268
DIMoveTo 268
DIMsgNumber 269
DIParamTagGet 287
DIParamTagSet 287
DIParamTagStringGet 287
DIPopupButton 269
DIPositionGet 269
DIPositionHome 269
DIPositionSet 269
DirPathFromPathName 224
DisableDialogItem 269
DisableDialogItem2 269
DisableDTTabbing 276
DisableTabName 290
discrete event programming
arrays 147
blocked and query messages 156
continuous blocks 158
explicit connector messages 159
functions 160
how ExtendSim runs DE models 142
item data structures 150
item messaging 153
Make Your Own blocks 146
message emulation 158
notify message 157
overview 146
pseudocode of the simulation loop 142
pulling 155
pushing 154
scheduling events 147
SysGlobal variables 160
system variables 142
types of blocks 148
zero time events 149
discrete rate programming 165
DISetFocus 269
DISetParent 269
display only option 19, 36, 50
DisposeArray 66, 340
DisposePlot 311
distributions
Binomial function 210
Exponential function 210
Gamma function 210
Gaussian function 210
LogNormal function 210
Mean distribution function 210
Pascal function 210
Poisson function 210
Random function 211
RandomCalculate function 211
RandomCheckParam function 212
RandomGetModelSeedUsed function 212

RandomRead function 212
students t 213
TStatisticValue function 213
DITitleGet 270
DITitleSet 270
DIToolTipSet 288
DivC 210
division by integer 69
DLL Add block 88
DLLBoolCFunction 246
DLLBoolPascalFunction 246
DLLBoolStdcallFunction 246
DLLCtoPString 61, 87, 245, 246
DLLDoubleCFunction 246
DLLDoublePascalFunction 246
DLLDoubleStdcallFunction 246
DLLLoadLibrary 247
DLLLongCFunction 247
DLLLongPascalFunction 247
DLLLongStdCallFunction 247
DLLMakeProcInstance 247
DLLMakeProcInstanceLibrary 247
DLLPtoCString 61, 87, 245, 247
DLLs 86, 245
example code for DLL Add block 88
example code for Miles block 87
interface 86
Python example 89
VB.net example 89
DLLUnloadLibrary 247
DLLVoidCFunction 247
DLLVoidPascalFunction 247
DLLVoidStdcallFunction 247
DLogNormal 210
Donut Chart 316
double (data type) 33, 207
definition 60
Do-While 70, 71
DPascal 210
DPoisson 210
DragCloneToBlock 100, 198
drawing tools 9
DTColumnTagGet 287
DTColumnTagSet 288
DTColumnTagStringGet 288
DTGrowButton 276
DTHasDDELink 276
DTHideLinkButton 283
DTPaneFixed 276
DTResizeToCols 276
DTRowFontSize 276
DTRowHeightSet 277
DTToolTipSet 277
DuplicateBlock 303
DuringContinue 293
DuringHBlockUpdate 291
dynamic arrays 26, 30, 66, 340
functions 340
number per block 380
dynamic data exchange (DDE) 229
dynamic data link functions 280
dynamic data link messages 203
dynamic data linking 55
Dynamic Link Libraries (DLLs) 86
dynamic text 15, 36
add to right click menu 19
display only 19
display only option 19
functions 283
visible in all tabs 19
DynamicArrayIndexByName 340
DynamicDataTable 277
DynamicDataTable2 277
DynamicDataTableVariableColumns 66, 275, 277
DynamicDataTableVariableColumns2 277
DynamicTextArray 283
DynamicTextIsDirty 283
DynamicTextSetDirty 283

E

E (scientific) notation
definition 30
EColorFromHSV 366
EColorFromOldExtendColor 366
EColorFromRGB 366
EColorIsValid 366
EColorPart 367
EColorPicker 367
EColors 365
EColorToHTML 366
EColorUpdateValue 367
EDCalcDate 362
EDCalendarDateGet 362
EDCalendarDates 363

EDCalendarShow 363
EDConvertDate 363
EDDateToSimTime 363
EDDateToString 363
EDDateValue 363
EDDayOfTheWeek 363
EDGetCurrentDate 363
EDGetStartDate 363
editable text 36
    31 character limit 15
    add to right click menu 19
    dialog item 15
    display only 19, 36
    visible in all tabs 19
editable text 31 character 15
EDNow 363
EDSimTimeToDate 363
EDStringToDate 364
EigenValues 215
embedded objects 119
empty circle 182, 185
end of line 78
endif 83
EndSim 142, 146, 196
EndTime 138, 140, 142, 191
Enter Selection command 80
entry boxes 14
Equation block 94, 218
equation functions 218
Equation(I) block 94
equation-based blocks 2, 27, 94
    differences from custom block code 31
    user-defined functions 94
EquationCalculate 219
EquationCalculate20 219
EquationCalculateDynamic 219
EquationCalculateDynamicVariables 219
EquationCompile 220
EquationCompile20 220
EquationCompileDynamic 220
EquationCompileDynamicVariables 220
EquationCompileDynamicVariablesSilent 221
EquationCompilePlatform 198
EquationCompileSetStaticArray 221
EquationDebugCalculate(221
EquationDebugCompile 221
EquationDebugDispose 221
EquationDebugSetBreakpoints 221
EquationGetStatic 222
EquationIncludeSet 222
equations 94
    differ from custom block code 31
EquationSetStatic 222
Erf 208
Euler integration 214
evaluating expressions 69
event
    list 149
    posting 149
    scheduling 146, 147
event clock 147
Example block 6
Excel Client-Server Model Workbook.xls 125
Excel interface 125
Execute 119, 121
ExecuteMenuCommand 303
Executive block 142, 292
    controlling simulation 145
    controlling timing 146
Exp 208
Exponential distribution function 210
exponential number 30
Export 223
ExportText 223
expressions
    evaluating 69
ExtendMaximize 303
ExtendMinimize 303
ExtendSim
    databases 113
    upper limits 380
ExtendSim IDE 2
ExtendSim OLE.exe 125
ExtendSim_10 83
extensions 85
    DLLs 86
    folder 85
    pictures 90
    sounds 89
external source code 42, 83
    source folder 84
External Source Code command 84

F

FALSE 63
fast Fourier transform 208
FFT 208
file I/O functions 222, 224
file types for extensions 85
FileChoose 224
FileClose 224
FileDelete 224
FileEndOfFile 224
FileExists 224
FileGetDelimiter 225
FileGetPathName 225
FileInfo 225
FileIsOpen 225
FileNameFromPathName 225
FileNew 225
FileOpen 225
FileRead 226
FileRewind 226
FileWrite 226
Fill Color for dialog items 17
FinalCalc 142, 146, 195
FinalCalc2 142, 146, 195
financial functions 213
Find command 80
Find in Files tab (Find String dialog) 80
Find String dialog 80
FindBlock 304
FindInHierarchy 257
FindInHierarchy2 258
FindMinimum 340
FindMinimumWithThreshhold 340
FindNext 304
FixDecimal 208
fixed arrays 26, 30, 66
Floor 208
flow attributes 153
flow order 191
FlowAttList array 153
FlowAttType array 153
FlowAttValues array 153
FlowBlockReceiveN 203
For construct 70, 71
format of numbers 17
formats
- data sources 94
- number formats 17

FormatString 358
FormatStringReal 358
form-based interface 125
Fortran 29
forward declaration 73
frame 15, 40, 49
free() 342
FreeMemory 368
functions 205
- 2D 250
- alerts, prompts 248
- animation 2D 250
- arrays as arguments to 67
- attributes 361
- bit-handling 218
- block categories 256
- block numbers 256
- calendar date 362
- charts 308
- column tag 284
- connections 260
- data tables 274
- database 318
- database copying 322
- database creating/deleting 319
- database DB address 335
- database import/export 322
- database linking/notify 337
- database properties 323
- database random data 331
- database read/write 327
- database selecting 321
- database sort/search 334
- database viewing 337
- date (egacy) 364
- DDE 229
- debugging 367
- definition 30, 33
- delay line 357
- diagnostic 248
- dialog 288
- dialog items 267
- dialog tabs 290
- distributions 210
- dynamic arrays 340
- dynamic data linking 280
- dynamic text 283
- equation 218

- file I/O, formatted 222
- file I/O, unformatted 224
- financial 213
- global 112
- help 368
- icon view 293
- integration 214
- internet access 226
- Interprocess communication (IPC) 229
- labels 256
- libraries 293
- linked list 349
- Mailslots 240
- math 207
- matrix 214
- messaging blocks 290
- models 293
- names 60, 256
- netlist 260
- notebook 293
- ODBC 241
- OLE 232
- overriding 34
- parameter tags 284
- passing arrays 341
- plotting 308
- pointers 342
- queue 356
- recursive 73
- returns 207
- scripting 300
- serial I/O 244
- statistical 210
- string 358
- text files 222, 224
- time (legacy) 364
- time units 361
- timer 364
- trigonometric 209
- types 206
- user-defined 72, 104
- web 368

Functions popup in Script tab 12
Functions popup menu 185
future value 213

G

GABlockRegister 113
GABlockRegisterContent 343
GABlockRegisterContents 113
GABlockRegisterStructure 113, 344
GABlockUnregisterContent 344
GABlockUnregisterStructure 344
GAClipboard 344
GACopyArray 344
GACreate 344
GACreateQuick 344
GACreateRandom 344
GADataTable 344
GADeleteRow 345
GADispose 345
GADisposeByIndex 345
GAExport 345
GAFindStringAny 345
GAGetArray 345
GAGetColumnsByIndex 345
GAGetColums 345
GAGetIndex 345
GAGetInfo 345
GAGetInitValue 345
GAGetInteger 346, 348
GAGetLong 346
GAGetName 346
GAGetReal 346
GAGetRows 346
GAGetRowsByIndex 346
GAGetString 346
GAGetString15 346
GAGetString31 346
GAGetString63 346
GAGetType 346
GAGetTypeByIndex 346
GAImport 347
GAInitializing 347
GAInsertRow 347
GALastUsedIndex 347
Gamma distribution function 210
GammaFunction 208
GAMultisim 347
GANonSaving 347
GAParameter 347
GAPopupMenu 347
GAPtr 347
GAResize 347
GAResizeByIndex 348

GASearch 348
GASearchCount 348
GASetArray 348
GASetInitValue 348
GASetInteger 348
GASetReal 348
GASetString 349
GASetstring15 349
GASetstring31 349
GASetString63 349
GASort 349
GAtoDBTable 349
Gaussian 210
Gaussian distribution function 210
GetAppPath 304
GetAttributeValue 361
GetAxis 311
GetAxisName 311
GetBlockDates 364
GetBlockInfo 294
GetBlockLabel 258
GetBlockMemSize 258
GetBlockTabNames 290
GetBlockType 258
GetBlockTypeNumeric 258
GetBlockTypePosition 197, 258, 305
GetConBlocks 135, 261
GetConHBlocks 261
GetConName 261
GetConnectedTextBlock 261
GetConnectedType 261
GetConnectedType2 261
GetConnectionColor 262
GetConnectionEColor 262
GetConnectorMsgsFirst 291
GetConnectorPosition 262
GetConnectorType 262
GetConNumber 135, 262
GetCurrentPlatform 369
GetCurrentTabName 290
GetDataTableSelection 278
GetDialogColors 270
GetDialogItemColor 270
GetDialogItemEColor 270
GetDialogItemInfo 270
GetDialogItemLabel 271
GetDialogNames 271
GetDialogVariable 98, 271
GetDimension 66, 341
GetDimensionByName 66, 341
GetDimensionColumns 66, 341
GetDimensionColumnsByName 66, 341
GetDraggedCloneList 100, 271
GetDTOffset 278
GetEnclosingHBlockCon 262
GetEnclosingHBlockNum 258
GetEnclosingHBlockNum2 258
GetExtendType 370
GetExtendVersion 370
GetExtendVersionString 370
GetFileReadMachineType 226
GetFileReadVersion 197, 278, 370
GetFileReadVersionString 370
GetFront 357
GetGlobalSimulationOrder 294
GetIndexedConValue 262
GetIndexedConValue2 263
GetIntermediateBlocks 263
GetItem 160
GetLibraryContents 294
GetLibraryInfo 294
GetLibraryPathName 294
GetLibraryStringInfo 294
GetLibraryVersion 294
GetLibraryVersionByName 294
GetModelName 294
GetModelPath 294
GetModelSimulationOrder 295
GetModifierKey 249
GetMouseX 197, 249, 305
GetMouseXActiveWindow 249
GetMouseY 197, 249, 305
GetMouseYActiveWindow 249
GetMsgSendingBlock 291
GetNumCons 263
GetObjectHandle 119, 124
GetPassedArray 104, 106, 342
GetPlotterValue 311
GetPreference 304
GetRear 357
GetRecentFilePath 304
GetReportType 308

GetRightClickedCon 263
GetRunParameter 295
GetRunParameterLong 295
GetSerialNumber 295
Get-Set Dialog Variable model 98
GetSignalName 311
GetSignalValue 311
GetSimulateMsgs 291
GetSimulationPhase 295
GetStaticNames 259
getStaticVariable 271
GetSystemColor 272
GetTickCount 312
GetTimeUnits 362
GetUserPath 304
GetVariableNumeric 272
GetWindowsHndl 296
GetWorksheetFrame 305
GetY1Y2Axis 312
global arrays 67
  AttribType 152
  AttributeList 152
  AttribValues 153
  functions 342
  working with 109
global numbers 256
global variables 76
  definition 30
  for passing messages 112
  lists 191
  reserved 192
  scope 62
  SysGlobals 160
  use during CheckData or InitSim 164
  use during Simulate 161
GlobalProofStr 191
GlobalToLocal 259
Go To Function/Message Handler command 81
Go To Line button 79, 81
Go To Line command 81
goal seeking 118
GOTO statement 70, 72
green location arrow 174, 180, 185
groups
  for radio buttons 15

H

HBlockClicked 249
HBlockClose 198
HBlockFromLibrary 198
HBlockHelpButton 199
HBlockMove 199
HBlockOpen 199
HBlockSaveToLibrary 199
HBlockUnlinkFromLibrary 305
HBlockUpdate 199
header files 81
heapCheck 368
help 369
  functions 368
  getting technical support 4
Help tab 14
HelpButton 199
HideDialogItem 272
HideDialogItem2 272
hiding dialog items 97
hierarchical blocks
  animation 131
HSV 130, 366

I

icon
  alternate views 9
  animating in 2D 53
  block part 8
  creating 9
  functions for view 293
  red border around block 173
  showing a picture on 134
  tools 21
  views 9
icon positioner 301
Icon tab 7
  exploring 9
Icon toolbar 21, 46
  description 21
icon view
  functions 293
Icon views popup 9, 10
IconBody 259
IconGetClass 293
IconGetView 293

IconGetViewName 293
IconSetViewByIndex 293
IconSetViewByPartialName 293
IconViewChange 199
IDE 2
identifier
  definition 30
Identity 215
IdentityC 215
IEEE floating point numbers 60
IF statement 70, 71
ifdef 82
If-Else statement 70, 71
ifndef 83
Ignore comparison and always break 187
Ignore conditions and always break 188
Imagine That, Inc. 4
Import 223
Import function 110
ImportText 224
Include additional block information option 14
include files
  definition 12
  MouseClick 100
  New Include File command 82
  referencing 81
  Save Include File command 82
IncludeFileEditor 222
Includes popup menu 185
indentation guides 79
index of items 150
indexing
  table by data source type 94
indexing (data source) 94
indicators 185
INetCloseHandle 227
INetConnect 227
INetFileImportText 227
INetFindNextFile 227
INetFTPCreateDirectory 227
INetFTPDeleteFile 227
INetFTPExport 227
INetFTPExportGA 227
INetFTPExportText 228
INetFTPFindFirstFile 228
INetFTPGetCurrentDirectory 228
INetFTPGetFile 228
INetFTPImport 228
INetFTPImportGA 228
INetFTPImportText 228
INetFTPPutFile 228
INetFTPRemoveDirectory 229
INetFTPRenameFile 229
INetFTPSetCurrentDirectory 229
INetGetFindFileInfo 229
INetGetFindFileName 229
INetOpenSession 229
INetOpenURL 229
initializing connectors 102
InitSim 140, 195
Inner 215
InnerC 215
InstallArray 312
InstallAxis 312
InstallFunction 312
Int 208
integer
  division of an 69
  function return 207
  maximum value 60
  to real 64
  to string 65
integer (data type) 33
  definition 60
integer array for SysGlobal 151
Integerabs 208
IntegerParameter 248
IntegerParameter2 248
integrated development environment (IDE) 2
IntegrateEuler 214
IntegrateInit 214
IntegrateTrap 214
integration functions 214
interface 125
internet access functions 226
Interprocess communication (IPC)
  functions 229
IPCAdvise 229
IPCCheckConversation 230
IPCConnect 230
IPCDisconnect 230
IPCExecute 230
IPCGetDocName 230
IPCLaunch 230

IPCOpenFile 231
IPCPoke 231
IPCPokeArray 231
IPCRequest 231
IPCRequestArray 231
IPCSendCalcReceive 231
IPCServerAsync 232
IPCSetTimeOut 232
IPCSpreadSheetName 232
IPCStopAdvise 232
IsBlockSelected 305
IsConVisible 263
IsFirstReport 308
isKeyDown 249
IsLibEnabled 305
IsMenuItemOn 305
IsMetric 305
IsSimulationPaused 296
item
    array 151
    priority 151
    quantity 151
item attributes 152
item index 150
Item library
    programming 146
itemArray arrays 150
itemArrayC
    passed by SysGlobal9 151
itemArrayI
    passed by SysGlobal4 151
itemArrayI2 152
itemArrayR
    passed by SysGlobal3 151
items
    flow is blocked 156
    pulling 155
    pushing 154

J

Java 29

L

label 15
label functions 256
labels for dialog items 18
labels on connectors 23
LastBlockPlaced 305
Lastkeypressed 250
LastSetDialogVariableString 272
left to right order 191
libraries
    functions 293
    number of blocks per 380
    protecting 90
    size 380
    source folder 84
LibrariesOpen 296
LibraryGetInfoByName 305
LibraryUsed 296
limits of ExtendSim 380
line numbers 79
Link Alerts 114
Link Contents 114
Link Structure 114
LinkContent 203
linked list functions 349
linked lists
    sorting 67
LinkStructure 203
list of tables 319, 327, 328
ListAddElement 350
ListAddString63s 350
ListCopyElement 350
ListCreate 351
ListCreateElement 351
ListDeleteElement 351
ListDispose 351
ListDisposeAll 351
ListElementMinMax 351
ListGetCount 352
ListGetDouble 352
ListGetElements 352
ListGetIndex 352
ListGetInfo 352
ListGetLong 352
ListGetName 352
ListGetString 353
ListLastElementIndex 353
ListLocked 353
ListSearch 353
ListSearchCount 353
ListSearchCountLongs 353

ListSearchLongs 354
ListSetDouble 354
ListSetLong 354
ListSetName 354
ListSetSort 354
ListSetSort2 354
ListSetString 354
literal
    definition 31
local numbers 256
local variables 34, 62, 76
    definition 31
LocalNumBlocks 259
LocalToGlobal 259
LocalToGlobal2 259
location arrow 174, 175
Log 208
Log10 208
Log2 208
logical operators 69
LogNormal distribution function 210
long 207
long (data type) 33
    definition 60
LUdecomp 216
LUdecompC 216

M

magnitude operators 69
Mailslot functions 240
MailSlotClose 240
MailSlotCreate 240
MailSlotRead 240
MailSlotReceive 199, 240
MailSlotSend 241
MaintenanceSupportPlanExpired 370
Make Your Own block 160, 165
Make Your Own blocks 146
MakeArray 66, 111, 341
MakeArray2 66, 341
MakeBlockInvisible 305
MakeConnection 305
MakeDialogModal 289
MakeFeedbackBlock 263
MakeOptimizerBlock 259
MakeScatter 318
MakeSelectionHierarchical 199
malloc() 342
Mandelbrot model 136
MatAdd 216
MatAddC 216
matching
    #ifdef, #endif, and #else 78
    braces 78
    strings 78
MatCopy 216
MatCopyC 216
math functions 207
math operators 68
MatInvert 216
MatInvertC 216
MatMatProd 216
MatMatProdC 216
matrix functions 214
MatScalarProd 217
MatScalarProdC 217
MatSub 217
MatSubC 217
MatVectorProd 217
MatVectorProdC 217
Max2 208
Mean 210
memory usage when programming 103
message emulation 158
message handlers
    block status 197
    block to block 202
    categories 111
    categories and types 194
    connector messages 201
    constants 292
    CreateBlock 50
    definition 31, 33
    dialog messages 36, 199
    dynamic data link 203
    example 33, 75
    format 75
    model status 196
    OLE 203
    overriding 34, 75
    purpose during DE initialization loop 144
    purpose during initialization loop 141
    Simulation Messages 194
    types 111

use during DE initialization loop 144
use during initialization loop 141
message name 36
messages 112
AbortDialogMessage 200
ActivateModel 196
AdviseReceive 203
AnimationStatus 196
AttribInfo 202
block status 197
block to block 202
BlockClick 197
BlockIdentify 197
BlockLabel 197
BlockMove 197
BlockRead 197
BlockReceive 202
BlockReport 196
BlockRightClick 197
BlockSelect 198
BlockTableInfo 202
BlockUndelete 198
BlockUnselect 198
Button 201
Cancel 200
CellAccept 200
CheckData 195
ClearStatistics 202
CloneInit 198
CloseModel 196
ConArrayChanged 201
ConArrayChangedComplete 201
ConArrayCollapseChanged 201
ConnectionBreak 201
ConnectionClick 201
ConnectionMake 198, 202
connector messages 201
ConnectorName 202
ConnectorRightClick 198, 202
ConnectorShowHide 198
ConnectorToolTip 202
ContinueSim 195
CopyBlock 198
CreateBlock 198
DataTableHover 200
DataTableResize 200
DataTableScrolled 200
DEExecutiveArrayResize 202
DeleteBlock 198
DeleteBlock2 198
DialogClick 200
DialogClose 200
DialogItem 201
DialogItemRefresh 200
DialogItemToolTip 200
DialogOpen 200
DragCloneToBlock 100, 198
dynamic data link 203
EndSim 196
EquationCompilePlatform 198
FinalCalc 195
FlowBlockReceiveN 203
functions for communicating 290
GetDraggedCloneList 100
HBlockClose 198
HBlockFromLibrary 198
HBlockHelpButton 199
HBlockMove 199
HBlockOpen 199
HBlockSaveToLibrary 199
HBlockUpdate 199
HelpButton 199
IconViewChange 199
InitSim 195
LinkContent 203
LinkStructure 203
list 194
MailSlotReceive 199
MakeSelectionHierarchical 199
model status 196
ModelSave 196
ModifyRunParameter 194
netlist 291
OK 200
OldFileUpdate 196
OLE 203
OLEAutomation 203
OpenModel 196
OpenModel2 196
PasteBlock 199
PasteBlock2 199
PauseSimulation 196
PlotterClose 199
PreCheckData 194, 195
ProofAnimation 202
QueueFunction 202
ResumeSim 197
ResumeSimAllBlocks 197
sent by application 112
sent during user interaction 112

ShiftSchedule 202
SimFinish 196
SimOrderChanged 197
SimSetup 197
SimStart 194
Simulate 195
simulation 194
StepSize 195
TabSwitch 201
TimerTick 199
types of 111
UpdateStatistics 202
UserMsg0-9 202

meter 15, 41

methods
for automation 119

Miles block 44

Min2 208

MOD operator 69

model
artificial intelligence 118
debugging a 170
discrete event (running) 142
functions 293
goal-seeking 118
how discrete event models work 146
profiling 170
programming 293
self-modifying 118
size 380
status messages 196
text blocks as commands 115
timing for discrete event 146

ModelLock 296

ModelSave 196

ModelSettingsGet 306

ModelSettingsSet 306

ModernRandom 191

ModifyDate 364

ModifyRunParameter 194

ModL
case sensitivity 60
code conventions 49
code example 47
code part of block 7
compared to C++ 27
compared to Java, Visual Basic, FORTRAN 29
compiled to machine code 12
conditional compilation 76, 82
connector names as variables 34
constant definitions 33, 60
data types 33, 60
external source code 42, 83
functions and message handlers 33
language terminology 30
layout 32
overview of the language structure 27
preprocessor directives 76, 82
type declarations 33, 60

ModL feature overview 30

modulo 209

MouseClick.h 100

MoveBlock 306

MoveBlockTo 306

MovieOn 191

movies 253

MsgEmulationOptimize 292

MultC 210

multiple simulations 191

multiple statements 70

MyBlockNumber 135, 259

MyLocalBlockNumber 259

N

name
dialog items 17

name functions 256

names
constants 60
functions 60
reserved 60
variables 60

names of blocks
length of name 380

NearlyEqual 208

NearlyGreaterThan 208

NearlyLessThan 208

netlist messages 291

New Include File command 82

NextTimes 149

No resize bar 21

no such parent 318

no such record 318

NodeGetCurrentValue 263

NodeGetIDIndex 263

normal connectors 22, 35

not linked error 319
not unique error 318
not unique index 318
notebook functions 293
NotebookClose 296
NotebookIsOpen 297
NotebookItemInfo 297
NotebookItemInfoString 297
NotebookItemRect 296
NotebookItems 296
notebooks
  functions 293
NotebooksInfo 297
notify message 157
NoValue 63, 69, 208
  converted to integer 64
Now 364
number of periods 213
numbers 60
NumBlocks 260
numeric conversion 64
NumericParameter 248
NumericParameter2 248
NumMultiLaunchesAllowed 370
NumPlotPoints 313
NumScenarios 191
NumSims 138, 140, 142, 143, 191
NumSteps 138, 141, 191
NumToFormat 359

O

object
  for Automation 119
ObjectIDNext 260
objects
  animation 128
ODBC functions 241
ODBCBindColumn 242
ODBCColAttribute 242
ODBCColumns 242
ODBCColumns2 242
ODBCConfigDataSource 242
ODBCConnect 242
ODBCConnectName 242
ODBCCountRows 243
ODBCCreateTable 243
ODBCDisconnect 243
ODBCDriverConnect 243
ODBCExecuteArray 243
ODBCExecuteQuery 243
ODBCFetchRows 243
ODBCFreeStatement 243
ODBCInsertRow 243
ODBCKeyword 243
ODBCNumResultCols 244
ODBCSetRows 244
ODBCSetRowsType 244
ODBCSuccessInfo 244
ODBCTables 244
OK 200
OldFileUpdate 196
OLE
  automation client 119
  automation server 119
  BlockMsg 123
  C++ examples 120
  Execute 121
  GetObjectHandle 124
  IDispatch interface 120
  messages 203
  methods 119
  Poke 122
  Request 122
  VB.net DLL example 125
  Visual Basic 125
  Visual Basic examples 125
OLE functions 232
OLEActivate 233
OLEAddRef 233
OLEArrayParam 233
OLEArrayParamVariableColumns 233
OLEArrayResult 233
OLEArrayResultVariableColumns 233
OLEAutomation 203
OLECreateObject 233
OLEDBParam 234
OLEDBResult 234
OLEDeactivate 234
OLEDispatchGetCLSID 234
OLEDispatchGetDispatchName 234
OLEDispatchGetDispID 234
OLEDispatchGetDoc 234, 236
OLEDispatchGetFuncIndex 234

OLEDispatchGetFuncInfo 234
OLEDispatchGetHelpContext 235
OLEDispatchGetNames 235
OLEDispatchInvoke 235
OLEDispatchParam 235
OLEDispatchPropertyGet 235
OLEDispatchPropertyPut 235
OLEDispatchresult 235
OLEGAParam 235
OLEGAResult 235
OLEGetCLSID 235
OLEGetDispatchName 236
OLEGetDispID 236
OLEGetDoc 236
OLEGetFuncIndex 236
OLEGetFuncInfo 237
OLEGetGUID 237
OLEGetHelpContext 233
OLEGetInterface 237
OLEGetNames 237
OLEGetRefCount 237
OleGlobal 191
OleGlobalInt 191
OleGlobalStr 191
OLEInsertLicensedObject 238
OLEInsertObject 238
OLEInsertObjectFromFile 238
OLEInvoke 238
OLELongParam 238
OLELongResult 238
OLEObjectIsRegistered 238
OLEPropertyGet 239
OLEPropertyPut 239
OLERealParam 239
OLERealResult 239
OLERelease 239
OLEReleaseInterface 239
OLERemoveObject 239
OLERequestLicKey 239
OLESetNamedParam 239
OLEStringParam 239
OLEStringResult 239
OLESupressInvokeErrors 240
OLEVariantParam 240
OLEVariantResult 240
Open Block Structure command 7
OpenAndSelectDialogItem 272
OpenAndSelectDialogItem2 272
OpenBlockDialogBox 289
OpenDialogBox 289
OpenDialogBoxToTabName 290
OpenEnclosingHBlock 289
OpenEnclosingHBlock2 289
OpenExtendFile 306
OpenModel 196
OpenModel2 196
OpenNotebook 297
OpenNotebook2 297
OpenURL 369
operations
    between any type and a string 65
    between reals and integers 64
operators 68
Optimizer block 94
Option key 249
Outer 217
OuterC 217
overridin
    message handlers 34
overriding
    functions 34
    message handlers 75
    user-defined functions 74

P

panes
    connectors 8
parameter
    add to right click menu 19
    adding to a dialog 49
    changing globally from a block 98
    changing through text on the model 115
    dialog item 14, 36
    display only 19, 36
    format 17
    formats for numbers 17
    tags 284
    visible in all tabs 19
parameter fields 14
parameter tag functionss 284
Pascal distribution function 210
Pascal to C string function 61
pass by value or reference 104

PassArray 104, 342
passing array functions 341
passing arrays 105, 341
  connectors 105
  precautions 106
  using globals 106
Passing Arrays model 104
passing blocks 148
passing messages 112
PassItem 160
PasteBlock 199
PasteBlock2 199
PasteBlock3 199
PauseSim() 368
PauseSimForSave 298
PauseSimulation 196
payment 213
PI 63
Pick Screen Color button 366
PictureList 256
pictures 90, 134, 254
  naming conventions 90
PieChart 316
PieChartSlice 316
pixels
  animating 136
  coordinates 253
  measurement 250
PlaceBlock 306, 307
PlaceBlockInHBlock 306
PlaceDotBlock 307
PlaceTextBlock 307
PlaceTextBlockInHBlock 307
Planet Dance model 129
Platform_Macintosh_Defined_Symbol 83
Platform_Windows_Defined_Symbol 83
PlaySound 248
PlotNewBarPoint 313
PlotNewPoint 313
PlotNewScatter 318
PlotPropertyChange 199
PlotSignalFormat 313
plotter functions 308
PlotterAutoscaleLimits 313
PlotterBackground 313
PlotterClose 199
PlotterNameGet 313
PlotterNameSet 313
PlotterSignalColorSet 313
PlotterSignalEColorSet 314
PlotterSignalValueGet 314
PlotterSignalValueSet 314
PlotterSquare 314
PlotterValueGet 314
PlotterValueSet 314
PlotterXAxisCalendar 315
PlotterXAxisTime 315
pointer functions 342
PointerDispose 342
PointerFromDynamicArray 342
pointers 104
PointerToDynamicArray 342
pointertype (data type) 33, 246
  definition 60
Poisson distribution function 210
Poke 119, 122
popup menus
  creating 48
  dialog item 14, 39
  in block dialogs 48
PopupCanceled 272
PopupItemParse 273
PopupMenuAppendArray 273
PopupMenuArray 273
posting events 149
PostInitSim 140
Pow 209
PreCheckData 140, 194, 195
PrecisionTimer 364
PrecisionTimerScale 364
pre-defined constants 63
preprocessor directives 29, 76, 82
preprocessor symbols 83
present value 213
procedures 72
Profile Block Code command 170
ProfileBlockGet 368
profiling 170
programming
  2D animation 128
  arrays 103
  changing parameters globally 98
  code search and replace 80
  debugging 171

dialogs 95
discrete event blocks 146
discrete rate blocks 165
DLLs 86
extensions 85
include files 81
profiling 170
scripting 118
sounds 89
techniques 78
text as commands 115
Trace 170
viewing debugging data 171
viewing intermediate results 171
programming languages
C++ 27
C++ example for Automation 120
Fortran 29
Java 29
used for OLE and ActiveX 119
VBA for Automation 125
Visual Basic 29
Visual Basic for Automation 125
prompts 248, 300
Proof Animation
numerical format 359
ProofAnimation 202
ProofEncode 256
ProofEncodeReset 256
Properties dialog for animation objects 129
protecting libraries 90
pseudocode 138, 142
pulling items 155
pushing items 154
PushPlotPic 315
PutFront 357
PutRear 357
Python 89

Q

QueGetN 357
QueInit 357
QueLength 357
QueLookN 357
Query Equation block 94
Query Equation(I) block 94
query message 156
QueSetAlloc 357
QueSetN 357
Queue Equation block 94
queue functions 356
QueueFunction 202
QuickTime 253
QuickTimeAvailable 307

R

radians 209
radio button
defining in a dialog 48
dialog item 15
groups 15
programming for 37
radio group ID 37
radio control 37
Radio Group ID 37
Random 211
Random distribution function 211
random number generator 191
RandomCalculate 211
RandomCheckParam 212
RandomGetModelSeedUsed 212
RandomGetSeed 212
RandomReal 212
RandomSeed 191
RandomSetSeed 212
RandomString 359
rate 213
Rate library
programming 165
Read/Write Index Checking 318
real
function return 207
to integer 64
to string 65
real (data type) 33
definition 60
Real (uniform) function 212
real array 151
Realabs 209
Realmod 209
RealToStr 359
RealToStrShortest 359
recursive functions 73
red border 173
red circle 179, 185

red name of a block 45
RefreshDatatableCells 278
RefreshPlotter 315
RegisterBlockInLeftClickDB 101
registered blocks 113
registering blocks 55, 280
regular expression (searching) 80
RemoveAttribute 361
RemoveSignal 315
RenamePlotter 315
reporting
    programming 170
Request 119, 122
reserved database 114
    _character 114
    blocks that use 114
residence blocks 148
ResizeDTDuringRead 197, 278
resizing a tab 13
Resource Order ID 152
ResourcePoolAllocate 300
ResourcePoolAvailable 300
RestrictConnectorMsgs 292
ResumeSim 117, 197
ResumeSimAllBlocks 117, 197
ResumeSimulation 298
RetimeAxis 315
RetimeAxisNStep 315
Return statement 70, 72
returns
    function 207
RGB 366
Roots 217
Round 209
row index 94
rows and columns
    setting for text tables and data tables 18
runs
    multiple 190, 191
RunSetup 298
RunSimulation 298

S

Save Block command 49
Save Include File As command 82
SaveModel 298
SaveModelAs 298
SaveTopDocAs 298
scatter plots 317
scientific notation
    definition 30
scope 62
script editor
    miscellaneous features 81
    syntax styling 78
Script tab 7, 12
    syntax highlighting 78
    syntax styling 78
scripting 118
scripting functions 300
ScrollDTTo 278
searching and replacing 80
searching text 80
SeedListClear 212
SeedListRegister 212
Select Color window 366
SelectBlock 368
SelectBlock2 368
SelectConnection 307
self-modifying 118
SendConnectorMsgToBlock 292
SendItem 160
SendMsg 154, 160
SendMsgToAllCons 292
SendMsgToBlock 292
SendMsgToHBlock 292
SendMsgToInputs 154, 155, 158, 165, 292
SendMsgToOutputs 154, 155, 165, 292
sensitivity analysis
    CurrentSense variable 190
sensor connector 158
serial I/O functions 244
serial ports 244
SerialRead 245
SerialReset 245
SerialWrite 245
ServerOpenPort 232
Set Block Category command 56
Set Breakpoints window 185
SetAttribute 361
SetAxisName 315
SetBlockLabel 260
SetBlockSimulationOrder 299

SetConnectionColor 263
SetConnectionEColor 264
SetConnectionThickness 264
SetConVisibility 264
SetDataTableCornerLabel 278
SetDataTableLabels 279
SetDataTableSelection 279
SetDefaultTabName 290
SetDialogColors 273
SetDialogItemColor 273
SetDialogItemEColor 273
SetDialogVariable 98, 273
SetDialogVariableNoMsg 274
SetDirty 307
SetDTColumnWidth 279
SetDTRowStart 279
SetIndexedConValue 264
SetIndexedConValue2 264
SetModelSimulationOrder 299
SetPopupLabels 274
SetRunParameter 299
SetRunParameters 299
SetSelectedConnectionColor 264
SetSelectedConnectionEColor 264
SetSignalName 316
SetTickCounts 316
SetTimeConstants 362
SetTimeUnits 362
SetVariableNumeric 274
SetVisibilityMonitoring 274
Shift key 249
Shift Selected Code Left command 81
Shift Selected Code Right command 81
Shift-click 99
    _leftClickDB 102
    code for a custom stand-alone block 100
    code in blocks used remotely 100
ShiftSchedule 202
Show 2D Animation button 128
Show 2D Animation command 128
Show Animation command 250
    AnimationOn variable 190
Show Reserved Databases command 114
ShowBlockLabel 260
ShowFunctionHelp 222
ShowHelp 369
showing dialog items 97
ShowPlot 316
ShowPlot2 316
SimDelay 191
SimFinish 196
SimMode 191
SimOrderChanged 197
SimSetup 197
SimStart 194
Simulate 141, 195
SimulateConnectorMsgs 293
simulation messages 194
simulation order 191
simulation pseudocode 142
Simulation Setup command 138
simulations
    automating 118
    changing data while running 117
    internals 138
    messages 194
    methods 138
    multiple 190, 191
    number of runs (maximum) 380
    order when running 191
    paused 140
    pseudocode 138, 142
    running 138
    stopping multiple 142, 146
    time 146, 190
Sin 209
Sinh 209
sink connectors 103
sizes 103
SLClear 355
SLCreate 355
SLDelete 355
SLFlagGet 355
SLFlagRealGet 355
SLFlagRealSet 355
SLFlagSet 355
SLGetCount 355
SLGetCountStrings 356
slider 15, 41
SLIs 356
SLPopupMenu 356
SLSort 356
SLStringAppend 356
SLStringGet 356

SLStringGetIndex 356
SLStringInsert 356
SLStringRemove 356
smart highlighting 78
SortArray 341
SortArrayVariableColumns 279
sorting
  linked lists 67
sounds 89, 248
source code debugger (see "debugger") 172
source code editor 78
source connectors 103
source folder 84
Source pane 174
Speak 248
Speech Manager 248
SpinCursor 299
SpinCursorStart 299
SpinCursorStop() 300
Sqrt 209
StartTime 138, 140, 142, 191
StartTimer 365
StartTimerID 365
statements
  control 70
  definition 31
  multiple 70
static data limits 62
static text 15, 39
static variables 32, 34, 62, 76
  definition 31
  limits on data size 62
  scope 62
  uninitialized 62
statistics functions 210
Status block 291
StdDevPop 212
StdDevSample 213
Step Into tool 184
step loop 141
Step Out tool 184
Step Over tool 184
steps
  number of 190
StepSize 138, 140, 195
Stop and Go To tool 184
Stop Debugging and Edit Code 182
Stop tool 184
StopDataTableEditing 279
Stoptimer 365
StopTimerID 365
Str127 33, 60
Str15 33, 60
Str255 33, 60
Str31 33, 60
Str63 33, 60
StrFind 359
StrFindDynamic 284
StrFindDynamicStartPoint 284
StrGetAscii 359
string 207
  concatenation 68, 69, 249
  declaring 61
  function return 207
  functions 358
  literals 61
  to integer 65
  to real 65
String data type 33
  definition 60
string functions 358
StringCase 359
StringCompare 359
StringTrim 360
StripLFs 226
StripPathIfLocal 226
StrLen 360
StrPart 360
StrPartDynamic 361
StrPutAscii 360
StrReplace 360
StrReplaceDynamic 284
StrToReal 39, 360
structure
  open command 7
structure of a block 7
structures 67, 103
  using passed arrays 107
structures (pointers) 342
style of text for dialog items 20
SubC 210
submenus for blocks 55
subscripts of an array 65
SuppressWorksheetRedraw 307

Switch 72
switch 15, 41
Switch statement 70
SwitchPlotterRedraw 316
symbols 82
 Compiled_Debug 83
 ExtendSim_10 83
 Platform_Macintosh_Defined_Symbol 83
 Platform_Windows_Defined_Symbol 83
 pre-defined preprocessor 83
syntax colorization 78
syntax styling 78
SysDBNGlobalInt0-19 167
SysFlowGlobal0 165
SysFlowGlobal1 165
SysFlowGlobalInt0-21 166
SysFlowGlobalStr0 165
SysGlobal 76
SysGlobal variables 160, 192
 during CheckData or InitSim 164
 during Simulate message 161
SysGlobal0 147
SysGlobal0-22 161
SysGlobal1 170
SysGlobal12 150
SysGlobal13 147, 148
SysGlobal2 170
SysGlobal23 167
SysGlobal24 167
SysGlobal25 167
SysGlobal26 167
SysGlobal27 167
SysGlobal28 167
SysGlobal3 150
 passing itemArrayR 151
SysGlobal4 150
 passing itemArrayI 151
SysGlobal6 150
SysGlobal7 147, 148
SysGlobal9 150
 passing itemArrayC 151
SysGlobalInt0 147, 154, 155, 157, 164
SysGlobalInt0-40 162
SysGlobalInt1 164
SysGlobalInt11 165
SysGlobalInt3 154, 155, 157
SysGlobalInt41 165
SysGlobalInt42 164
SysGlobalInt43 164
SysGlobalInt44-52 167
SysGlobalInt53 167
SysGlobalInt54-56 167
SysGlobalInt57 167
SysGlobalInt58 167
SysGlobalInt59 167
SysGlobalInt6 157
SysGlobalInt60-79 167
SysGlobalInt8 150, 164
SysGlobalStr0-2 162
SysGlobalStr1 165
SysGlobalStr3 167
SysGlobalStr4-9 162
system variables
 definition 31
 list 190

T

tab character 223
tab delimited files 223
tab number 18
tab resizing 13
table list 319, 327, 328
tabs 8, 13
TabSwitch 201
Tan 209
Tanh 209
Target the DLL by name 89
technical support
 information to provide 4
terminology 30
text 115
 changing programmatically 95
 tables 15, 38
text as commands 115
text files
 functions 222, 224
 include files 81
 numerical data 223
 programming 223
 string data 223
text frame 15, 40
text tables 15, 38
 row and column 18
TextWidth 360

TickCount 365
time
  current time 190
time functions 364
time functions (legacy) 364
time in simulation 190
time units
  functions 361
TimeArray 147, 149
TimeBlocks 147
TimeEventMsgType 147
timer functions 364
TimerID 365
TimerTick 199
TimeToString
  see EDdateToString 363
tool tips
  connector 267
  dialog item 288
  on block dialogs 20
  on dialog tab 20
tools
  animation 21
  connectors 21
  in the Debugger window 184
tooltips 23
topic and item 120
TraceModeEnableDisable 368
transparency 366
transparent text 136
Transpose 218
TransposeC 218
Trapezoidal integration 214
trigonometry functions 209
TRUE 63
TStatisticValue 213
tutorial
  source code debugger 173
type conversion 64
  function arguments 207
type declarations 32
  definition 31
type of dialog item 17

U

uninitialized static variables 62
UnRegisterBlockInLeftClickDB 101
UnselectAll 307
UpdatePublishers 232
UpdateStatistics 202
upper limits 380
Use 361
UseRandomizedSeed 213
user-defined functions 72
  ADO functions 371
  exiting 73
  include files 81
  overriding 74
  recursive 73
user-defined procedures 72
UserError 171, 248
UserMsg0-9 202
UserParameter 248
UserPrompt 248
userPromptCustomButtons 248

V

value connector messages 158
variable connectors 21, 22, 35, 102
  convert to normal 21
  no resize bar 21
variable name 36
VariableNameToTabName 290
variables
  data consumption 61
  global 192
  list 190
  local 31, 62
  memory usage 103
  names 60
  pane 174
  scope 62
  static 31, 62
  SysGlobal 192
  system 190
Variables pane 174
VB.net 89, 125
VB.net COM DLL example 125
VBA example 125
version control 42, 83
Visible checkbox 19, 97
visible in all tabs 19
Visible option for dialog items 19
Visual Basic 29, 125

void 207
void functions (procedures) 30

W

W (width) and H (height) coordinates 98
WaitNTicks 130, 365
Watch(A) 188
WatchPoint condition 188
web functions 368
WhichDialogItem 274
WhichDialogItemClicked 200, 250, 268
WhichDTCell 279
WhichDTCellClicked 250
While loop 71
While statement 70
white circle 185
white space 78
WhoInvoked 274
WinRegSvr32 240
WinSetForegroundWindow 307
WinShellExecute 308
wizards 118
WorksheetRefresh 308
worksheetSettingGet 308
worksheetSettingSet 308

X

X and Y coordinates 17, 97

Y

yellow location arrow 175, 176, 185

Z

zero time event list 146
Zoom In button 81
zOrder 17, 54
zOrderanimation objects
    zOrder 129