

MATRIXx™

SystemBuild™ User's Guide

The MATRIXx products and related items have been purchased from Wind River Systems, Inc. (formerly Integrated Systems, Inc.). These reformatted user materials may contain references to those entities. Any trademark or copyright notices to those entities are no longer valid and any references to those entities as the licensor to the MATRIXx products and related items should now be considered as referring to National Instruments Corporation.

National Instruments did not acquire RealSim hardware (AC-1000, AC-104, PCI Pro) and does not plan to further develop or support RealSim software.

NI is directing users who wish to continue to use RealSim software and hardware to third parties. The list of NI Alliance Members (third parties) that can provide RealSim support and the parts list for RealSim hardware are available in our online KnowledgeBase. You can access the KnowledgeBase at www.ni.com/support.

NI plans to make it easy for customers to target NI software and hardware, including LabVIEW real-time and PXI, with MATRIXx in the future. For information regarding NI real-time products, please visit www.ni.com/realtime or contact us at matrixx@ni.com.

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599,
Canada (Calgary) 403 274 9391, Canada (Montreal) 514 288 5722, Canada (Ottawa) 613 233 5949,
Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530,
China 86 21 6555 7838, Czech Republic 420 2 2423 5774, Denmark 45 45 76 26 00,
Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, Greece 30 2 10 42 96 427,
India 91 80 51190000, Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970,
Korea 82 02 3451 3400, Malaysia 603 9131 0918, Mexico 001 800 010 0793, Netherlands 31 0 348 433 466,
New Zealand 1800 300 800, Norway 47 0 66 90 76 60, Poland 48 0 22 3390 150, Portugal 351 210 311 210,
Russia 7 095 238 7139, Singapore 65 6226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197,
Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 2 2528 7227,
Thailand 662 992 7519, United Kingdom 44 0 1635 523545

For further support information, refer to the [Technical Support Resources and Professional Services](#) appendix.
To comment on the documentation, send email to techpubs@ni.com.

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THEREFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

AutoCode™, DocumentIt™, LabVIEW™, MATRIXx™, National Instruments™, NI™, ni.com™, RealSim™, SystemBuild™, and Xmath™ are trademarks of National Instruments Corporation.

Product and company names mentioned herein are trademarks or trade names of their respective companies.

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or `ni.com/patents`.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Contents

Contents	iii
Using This Manual	xxiii
Welcome to SystemBuild	xxiii
Scope	xxiii
Conventions	xxiv
Support	xxvi
1 Introduction	
<hr/>	
1.1 Overview	1-1
2 Starting SystemBuild and Using the Catalog Browser	
<hr/>	
2.1 Starting and Exiting SystemBuild	2-1
2.2 Loading Data	2-2
2.2.1 Xmath Load Command.	2-2
2.2.2 Catalog Browser Load Dialog	2-2
2.3 Understanding Catalogs	2-4
2.4 Browsing a Catalog	2-5
2.5 Saving Data	2-8

	Xmath SAVE Command	2-8
	Catalog Browser Save Dialog	2-8
	SystemBuild AutoSave Feature	2-9
2.6	Accessing the SystemBuild Editor	2-10
2.6.1	Using the Catalog Browser Quick Access Menu	2-10
2.6.2	Opening a SuperBlock in the Editor	2-11
2.6.3	Dragging and Dropping in the Catalog Browser	2-12
2.6.4	Creating a New SuperBlock	2-13
2.7	Updating Catalog Browser Data	2-13
2.8	Editing Catalogs Using the Catalog Browser	2-14
2.8.1	Using the Tools Menu	2-14
2.8.2	Modifying a Catalog	2-15

3 Editing SuperBlocks

3.1	SuperBlock Hierarchies	3-1
3.1.1	SuperBlocks and Functional Blocks	3-2
3.2	Creating SuperBlocks	3-3
3.2.1	Creating a New SuperBlock from the Catalog Browser	3-4
3.2.2	Making a New SuperBlock from Existing Blocks	3-5
3.3	SuperBlock Dialogs	3-6
3.3.1	SuperBlock Properties Dialog	3-6
	Attributes Tab	3-7
	Code Tab	3-7
	Inputs Tab	3-8
	Outputs Tab	3-9

	Document Tab	3-9
	Comment Tab	3-9
3.3.2	SuperBlock Block Dialog	3-9
3.4	Creating a SuperBlock Reference	3-11
	Creating a Reference from the Catalog Browser	3-12
	Creating a Reference from the Editor	3-12
3.4.1	Creating a Copy of a SuperBlock	3-14
	Creating a Copy with Copy and Paste	3-14
	Creating a Copy by Modifying the SuperBlock Properties	3-14
3.4.2	Using Rename	3-15
	Replacing a SuperBlock with Catalog Browser Rename	3-15
	Creating a Copy with Rename	3-16

4 Editing Blocks

4.1	Creating Blocks	4-1
4.2	Block Dialog Elements	4-3
4.3	Block Dialog Fields	4-5
4.3.1	Parameters	4-7
4.3.2	Code	4-7
4.3.3	Inputs	4-7
4.3.4	Outputs	4-8
4.3.5	States	4-9
4.3.6	Document	4-9
4.3.7	Comment	4-9
4.3.8	Icon	4-10

4.3.9	Display	4-10
	Input Pins/Output Pins Display Mode	4-11
	Show Labels	4-11
	Propagate Labels	4-11
	Icon Color	4-12
	Icon Type	4-12
4.4	Using Block Dialogs	4-12
4.4.1	Using the Matrix Editor	4-13
	Entering a Matrix	4-13
	Editing a Matrix	4-14
4.4.2	Specifying Labels or Names	4-14
	SuperBlock External Input Labels	4-15
	Creating Sequential Names for Vectors or Matrices	4-15
	Shortcuts for Editing Labels or Names	4-18
4.5	DataTypes	4-19
4.5.1	Traditional Datatypes	4-21
4.5.2	Datatype and Typecheck Example	4-23
4.5.3	Traditional Datotyping Example	4-24
4.5.4	Datotyping Rules	4-25
4.6	Connecting Blocks	4-28
4.6.1	Connection Rules	4-29
4.6.2	Creating Connections	4-30
	Creating a Simple Connection	4-30
	Using the Connect Menu	4-30
	Using the Connection Editor	4-31
4.6.3	Automatic and Manual Connection Routing	4-33

4.7	Modifying Block Diagram Appearance	4-34
4.8	Example	4-36
4.9	Special Blocks	4-40
4.9.1	Conditional Execution (Condition, IfThenElse Blocks)	4-40
4.9.2	Repetitive Execution (While, Break Blocks)	4-41
4.9.3	Terminating Execution (Stop Block)	4-41
4.9.4	Execution Ordering (Sequencer Block)	4-41

5 SuperBlocks and SuperBlock Transformations

5.1	Continuous SuperBlocks	5-1
5.2	Discrete SuperBlocks	5-1
5.3	Triggered SuperBlocks	5-2
5.4	Procedure SuperBlocks	5-3
5.4.1	Standard Procedures	5-4
5.4.2	Macro Procedure	5-4
5.4.3	Inline Procedure	5-5
5.4.4	Background Procedure	5-6
5.4.5	Startup Procedure	5-6
5.4.6	Interrupt Procedure	5-7
5.4.7	Asynchronous Procedure Execution	5-8
5.4.8	Limitations of Asynchronous Procedures	5-10
5.5	File SuperBlocks	5-10
5.6	Timing Considerations	5-11
5.7	Simulation Timing Properties	5-12
5.7.1	At Next Trigger (ANT)	5-13
5.7.2	At Timing Requirement (ATR)	5-13

5.7.3	As Soon As Finished (SAF)	5-13
5.7.4	Asynchronous (ASYNC)	5-14
5.8	Subsystem Priorities	5-19
5.9	AutoCode Timing Properties	5-20
5.10	SuperBlock Transformation	5-23
5.10.1	Transformation Limitations and Implications	5-24
	Limitations	5-24
	Dynamic Blocks	5-24
	Gain Block	5-25
	Integrators and PID Controller	5-25
5.11	Transformation Methods	5-26
5.11.1	Transformation Using the Transform SuperBlock Dialog	5-26
5.11.2	Transformation Using the SuperBlock Properties Dialog Box	5-27
5.11.3	Initial Condition Transformations	5-28
5.11.4	Undoing a Transformation	5-28

6 SystemBuild Customization

6.1	user.ini File Format	6-1
6.2	Printer Settings (UNIX)	6-4
6.3	Default Text Editor	6-4
6.4	Comment Editor	6-5
6.5	Custom Menus	6-5
6.5.1	A Sample user.ini that Calls MSCs	6-6
6.5.2	A Typical Template for User Menus	6-7

6.6	SystemBuild Resource File (UNIX)	6-8
6.6.1	Controlling Colors	6-8
	Foreground and Background	6-8
	SystemBuild and ISIM Color Settings	6-9
6.6.2	Resizing, and Repositioning the Display	6-9

7

Simulator Basics

7.1	Introduction	7-1
7.1.1	Simulation User Interfaces	7-1
7.1.2	SystemBuild Editor Simulation Interface	7-1
7.1.3	Xmath Simulation Interface	7-2
	Sim Function Syntax	7-2
	Background Simulation	7-4
	Simulator Termination	7-5
7.2	Parameterization	7-5
7.2.1	The Simulation vars Keyword	7-6
7.2.2	Parameter Variable Scoping	7-6
7.3	SystemBuild Keyword Default Options	7-8
7.4	Operating System Command Line Simulation Interface	7-8
7.5	Analyze Function	7-10
7.5.1	Analyze from the Xmath Command Area	7-10
7.5.2	Analyze from the SystemBuild Editor	7-12
7.5.3	Automatic Analyze	7-13
7.6	Algebraic Loops	7-13
7.7	lin Function	7-16
7.8	simout Function	7-17
7.9	creatertf Command	7-18

7.10	Selecting an Integration Algorithm	7-18
7.10.1	Integration Algorithms	7-19
7.10.2	Integration Algorithm Recommendations	7-19
7.11	Simulation Errors	7-21
7.11.1	Simulation Software Errors	7-21
7.11.2	Hardware Errors	7-22
7.11.3	Operating System Errors	7-22
7.12	Scheduler	7-23
7.12.1	Subsystems	7-23
	Processor Group ID	7-23
	Subsystem Scheduling	7-24
	Continuous Subsystem Scheduling	7-25
	Discrete Subsystem Scheduling	7-25
	Properties of Discrete Scheduled Subsystems	7-26
	Simulation actiming Option Scheduling	7-27
7.13	Simulation Time Lines, Inputs, and Outputs	7-27
7.13.1	The Input Time Line	7-27
7.13.2	The Internal Time Line	7-28
7.13.3	Computing External Input Values	7-28
7.13.4	The Output Time Line	7-29

8 Interactive Simulation

8.1	The Interactive Simulation Process	8-3
8.2	The ISIM Window	8-4
8.3	Special Notes on ISIM	8-5
8.4	sim Keywords for ISIM	8-6
8.5	Standard Animation Icons	8-7

8.5.1	Strip Chart Display Icon	8-9
8.5.2	Multiple Line (Bar Graph) Icon	8-10
8.5.3	LED Digital Monitoring Display Icon	8-10
8.5.4	Numeric Display Icon	8-10
8.5.5	In/Out Pushbutton Switch	8-10
8.5.6	Text Icon	8-11
8.5.7	Slide Output Controller Icon	8-11
8.6	Using ISIM	8-12
8.6.1	Building the ISIM Car Model	8-12
8.6.2	Simulating the Car Model	8-14
8.7	Run-time Variable Editor	8-15
8.7.1	RVE and ISIM	8-15
8.7.2	RVE Commands and Functions	8-19
8.7.3	RVE-Compatible Blocks	8-21

9

Linearization

9.1	Linearization	9-1
9.2	Linearizing Single-Rate Systems About an Initial Operating Point	9-2
9.2.1	Continuous Systems	9-2
	Explicit Form	9-3
	Implicit Form	9-3
9.2.2	Discrete Systems	9-4
9.3	Exact vs. Finite Difference Linearization	9-5
9.4	Special Linear Models	9-5
9.4.1	Continuous Time Delay	9-5
9.4.2	State Transition Diagrams	9-5
9.4.3	FuzzyLogic Block	9-6

9.4.4	Integrator Block (Resettable)	9-6
9.4.5	UserCode Blocks	9-6
9.4.6	Procedure SuperBlocks Referenced from Condition Blocks	9-6
9.5	Linearizing About a Final Operating Point	9-6
9.6	Multirate Linearization	9-7
9.6.1	Interpretation of Multirate lin Results	9-8
9.6.2	Linearizing Fixed-point Blocks	9-10
9.6.3	References	9-10
9.7	Trim	9-11
9.7.1	trim Syntax	9-12
9.7.2	trim Algorithm	9-14
9.7.3	trim Behavior	9-15
	Stability	9-15
	Free Integrators	9-15
	Algebraic Loops and trim	9-16
9.7.4	trim Examples	9-16

10 Classical Analysis

10.1	Classical Analysis Tools	10-1
10.2	Classical Analysis Tools Process	10-2
10.3	Open-Loop Frequency Response	10-3
10.4	Time Response	10-7
10.5	Point-to-Point Frequency Response	10-9
10.6	Root Locus	10-11
10.7	Parameter Root Locus	10-13

11 Advanced Simulation

11.1	Explicit vs. Implicit Models	11-1
11.1.1	Explicit Models	11-2
11.1.2	Implicit Models	11-2
	Implicit Model Constraints	11-2
	Implicit States and Implicit Outputs	11-3
	Implicit Outputs	11-4
	Initialization	11-4
11.1.3	Implicit Model Examples	11-4
11.2	Operating Points	11-10
11.2.1	Continuous Subsystem	11-10
11.2.2	Discrete Subsystems	11-10
11.3	Inserting Initial Conditions	11-11
11.4	Use of sim, lin, simout for Implicit UCBs	11-13
11.5	Matrix Blocks in the Simulator	11-13
11.6	Sim Integration Algorithms	11-14
11.6.1	Comparing Integration Algorithms	11-15
11.6.2	Overview of the Algorithms	11-16
	Euler Integration Method	11-16
	Second Order Runge-Kutta (Modified Euler) Method	11-17
	Fourth Order Runge-Kutta Method:	11-18
	Fixed-Step Kutta-Merson Method	11-18
	Variable-Step Kutta-Merson Method	11-19
	Stiff System Solver (DASSL)	11-19
	Variable-Step Adams-Bashforth-Moulton Method	11-21
	QuickSim Method	11-22

	Over-determined Differential Algebraic System Solver (ODASSL)	11-24
	Gear's Method	11-30
11.7	Absolute and Relative Tolerances	11-31
11.7.1	Variable-Step Kutta-Merson Method	11-31
11.7.2	Stiff System Solvers (DASSL and ODASSL)	11-32
11.7.3	Variable-Step Adams-Bashforth-Moulton Method	11-32
11.7.4	Computing the Maximum Integration Step size in Variable-Step Integration Algorithms	11-33
11.8	Sample Simulation	11-34
11.9	State Events	11-42
11.9.1	ZeroCrossing Block	11-43
11.9.2	Continuous UserCode Blocks	11-44
	Restrictions and Limitations	11-46
11.9.3	Example	11-46

12 BlockScript

12.1	Introduction	12-1
12.1.1	The Block Paradigm	12-1
12.1.2	BlockScript Program Structure	12-2
12.2	BlockScript Variables	12-3
12.2.1	Block Variable Declarations	12-4
12.2.2	DataTypes and Dimensions	12-5
	Wildcard Dimensions and Dialog Imported Information	12-7
	Method for Implied Datatyping	12-8
	BlockScript Datatypes and Code Generation	12-9

12.3	The BlockScript Language	12-9
12.3.1	Operators and Precedence	12-9
12.3.2	Assignment Statements and Expressions	12-9
12.3.3	Looping Constructs	12-12
12.3.4	Functions	12-15
12.3.5	Environment Variables	12-19
12.4	Debugging Tips	12-20
12.5	Compiling BlockScript Blocks	12-20
12.6	Examples	12-22
12.6.1	Bessel Equation BlockScript Block	12-22
12.6.2	Discrete PID Controller BlockScript Block	12-23
12.6.3	Three-Cycle Delay Script	12-24
12.6.4	Linear Interpolation Algorithm Script	12-25
12.6.5	Hysteresis Script	12-27

13 SystemBuild Access (SBA)

13.1	Overview	13-1
13.2	Xmath Syntax Review	13-3
13.2.1	Command Syntax	13-3
13.2.2	Function Syntax	13-3
13.2.3	Inputs, Optional Inputs, and Keywords	13-3
13.3	Basic SBA Tasks	13-4
13.3.1	Create	13-4
13.3.2	Query	13-4
13.3.3	Modify	13-6
13.3.4	Display	13-6

13.3.5	Delete	13-6
13.3.6	Sample Scripts	13-6
13.4	Using SBA	13-7
13.4.1	Keyword Ordering	13-7
13.4.2	Block Parameters	13-8
13.4.3	Error Handling	13-8
13.4.4	Input Formats	13-8
13.4.5	SuperBlock Editor Coordinate System	13-10
13.5	Tutorial	13-11
13.5.1	Building the Predator-Prey Model	13-11
13.5.2	Simulating the Predator-Prey Model	13-12

14 UserCode Blocks

14.1	Introduction	14-1
14.2	The Structure of UCBs	14-2
14.2.1	Explicit UCBs	14-2
14.2.2	Implicit UCBs	14-2
14.2.3	Implicit UCB Implementation	14-3
14.2.4	Implicit UCBs and <code>sim</code> , <code>lin</code> , or <code>simout</code>	14-8
14.2.5	Input Direct Terms	14-10
14.2.6	State Events	14-13
14.2.7	The Simulation API	14-14
14.2.8	Using the SIMAPI to Gather UCB Reference Information	14-15
14.2.9	Using the SIMAPI to Access and Modify Variables	14-16
14.2.10	Using the SIMAPI to Access Simulation Debugging Information	14-18
14.2.11	Using and Managing the Debugging Access Functions	14-20

14.2.12	Integration Algorithm Updates	14-23
	Other Issues and Notes	14-27
14.2.13	SIMAPI Debug UserCode Block Example	14-28
14.2.14	USR01 and IUSR01 Template	14-30
14.2.15	Explicit UserCode Function Calling Arguments	14-32
14.2.16	UserCode Function Arguments	14-33
14.3	Variable Interface UserCode Blocks	14-37
14.3.1	Overview	14-37
14.3.2	Using a Wrapper so that SystemBuild Can Simulate Code Written for AutoCode	14-38
14.3.3	Writing a Wrapper	14-40
	Converting Data From the SIM Interface	14-40
	Converting Data Back to the SIM Interface	14-41
14.3.4	Specifying the Variable Interface	14-41
	Setting Variable Interface Parameters	14-41
	Specifying Datatypes	14-41
	Specifying Input Shapes	14-42
	Specifying Output Shapes	14-42
14.3.5	Running a Variable Interface Example	14-42
14.4	UCB Programming Considerations	14-43
14.5	Building, Linking and Debugging UCBs	14-43
14.5.1	Collecting UserCode Files	14-44
	UCB Block Parameter Form	14-44
	CSOURCE and FSOURCE	14-44
	Specifying Sources in the makefile	14-45
	Reusing Sources from the Previous Simulation	14-45
	Specifying Another Location for UCB Code	14-45

14.5.2	Compiling and Linking User Code	14-46
14.5.3	Debugging User Code	14-47
14.6	Posting Error Indications	14-48

15 Fixed-point Arithmetic

15.1	Introduction to Fixed-point Arithmetic	15-3
15.1.1	Fixed-point Number Representation	15-3
15.1.2	Conversion Between Fixed-point Numbers	15-6
15.1.3	Addition and Subtraction	15-6
15.1.4	Multiplication	15-7
15.1.5	Division	15-8
15.1.6	Relational Operations	15-9
15.1.7	Overflow	15-10
15.2	SystemBuild Fixed-point	15-11
15.2.1	User Interface	15-11
15.2.2	Simulator	15-12
15.2.3	Building a Model and Demonstrating Overflow	15-13
15.2.4	Comparing Fixed- and Floating-Point Numbers	15-16
15.2.5	Comparing the Effects of Different Conversion Sequences	15-20
15.3	Fixed-point Blocks and I/O Datatype Rules	15-23
15.3.1	Advanced Simulation Topics	15-25
	Intermediate Types	15-25
	Simulation Issues	15-29
	32-bit Operation Issues	15-32
	Gain Block: A Special Case	15-32
15.3.2	Radix Calculations	15-33

15.4	MinMax Data Logging	15-37
15.4.1	Activating MinMax Logging	15-38
	Simulating with the minmax Keyword	15-38
	Saving MinMax Datasets to a File	15-38
15.4.2	MinMax Display Tool	15-38
15.5	User-Defined Data Types (Usertypes)	15-40
15.5.1	Usertype Editor	15-40
15.5.2	Usertype MathScript Commands	15-42
15.5.3	Using Usertypes in SystemBuild	15-42
15.5.4	Storing Usertypes	15-43
15.6	SystemBuild Functions in Fixed-point	15-44
15.6.1	Linearization Function	15-44
15.6.2	Simout Function	15-44
15.7	Scaling Aid Blocks	15-45

16 Building Custom Icons

16.1	IA Basics	16-1
16.1.1	Adding a Custom Icon to a Block Diagram	16-2
16.1.2	Sample Icon Source	16-2
16.2	Defining Custom SystemBuild Icons	16-3
16.2.1	Importing or Referencing an External Bitmap	16-3
16.2.2	Creating or Attaching an IA Source Icon	16-4
16.3	An Icon Source File	16-8
16.3.1	Icon Identification	16-10
16.3.2	Types	16-10
	Integer Types	16-10

- Real Types 16-10
- String Types 16-11
- 16.3.3 General Control and Calculation Statements 16-11
- 16.3.4 General Graphic Statements and Coordinate System 16-13
- 16.3.5 General Graphic Characteristic Statements 16-17
- 16.3.6 Animation Statements 16-19
- 16.3.7 Pointer Action Statements 16-20
- 16.3.8 Palette Definition 16-21
- 16.4 animation.cfg 16-21
 - 16.4.1 Important animation.cfg Keywords for Customized Icons 16-24
 - 16.4.2 icon.src Field for Customized Icons and New Palettes 16-24
- 16.5 Procedure for Building Your Own IA Custom Icons 16-25

17 Components

- 17.1 Introduction 17-1
 - 17.1.1 Component Scope 17-2
 - 17.1.2 Component Interface 17-2
 - 17.1.3 Component Parameter Sets 17-3
 - 17.1.4 Component References 17-3
 - 17.1.5 Component Access 17-3
 - Open Components 17-4
 - Encrypted Components 17-4
 - Licensed Components 17-4
- 17.2 Using Components in SystemBuild Models 17-5
 - 17.2.1 Viewing Components 17-5
 - 17.2.2 Creating References to Components 17-6
 - 17.2.3 Controlling Component Parameters 17-7

17.2.4	Loading Component Parameter Sets	17-7
17.2.5	Changing Scope into a Component Catalog	17-8
17.2.6	Simulating Models with Components	17-8
17.3	Creating Components	17-9
17.3.1	Restrictions on Component SystemBuild Hierarchies	17-9
17.3.2	Understanding Parameterization of Components	17-9
17.3.3	Understanding the Component Scope.	17-10
17.3.4	Mapping Exported Variables.	17-11
17.3.5	Customizing the Component Dialog	17-11
17.3.6	Documenting the Component	17-12
17.3.7	Creating Components Using the Component Wizard.	17-12
17.3.8	Modifying Components.	17-13
17.3.9	Unmaking a Component.	17-14
17.4	Creating and Using Parameter Sets	17-14
17.5	Using SBA with Components	17-16
17.6	Distributing SystemBuild Components	17-16
17.7	Examples	17-17
17.7.1	Encapsulating a SuperBlock Hierarchy	17-17
17.7.2	Exporting Component Parameters	17-18
17.7.3	Using the Parameter Set Interface	17-19
17.7.4	Interface Mapping	17-21
17.7.5	Using a Custom Dialog.	17-23

18 Palettes and Custom Blocks

18.1	Custom Palettes	18-1
18.1.1	Creating Palette Files	18-2
18.1.2	Palette File Syntax	18-3



Using This Manual



Welcome to SystemBuild

Congratulations on purchasing SystemBuild™, the industry's most powerful system modeling and simulation package. For your convenience, please read the brief sections that follow, so you can make the most efficient use of the product and its documentation.

Scope

This guide provides extensive user information on nearly every aspect of SystemBuild, with emphasis on the SystemBuild Catalog Browser, SuperBlock Editor, and simulator. Exceptions are as follows:

- Although the SystemBuild Blocks and SystemBuild functions and commands are introduced here, they are discussed in detail in the online help.
- The STD Editor is covered in the SystemBuild State Transition Diagram Block User's Guide.
- The Xmath Basics explains how to use the Xmath® analysis and design package. SystemBuild is closely tied to Xmath. This manual assumes you have knowledge of basic Xmath capabilities such as plotting, printing, Xmath command and function syntax, and MathScript programming.

Conventions

This sections describes the visual conventions used in this document.

Font Conventions

This sentence is set in the default text font, Bookman Light. Bookman Light is used for general text, menu selections, window names, and program names. Fonts other than the standard text default have the following significance:

Courier:	<code>Courier</code> is used for command and function names, file names, directory paths, environment variables, messages and other system output, code and program examples, system calls, prompt responses, and syntax examples.
bold Courier:	bold Courier is used for input (anything you are expected to type in).
italic:	<i>Italics</i> are used in conjunction with the default font for emphasis and publication titles. Italics are also used in conjunction with <i>Courier</i> or <i>bold Courier</i> to denote placeholders in syntax examples.
Bold Helvetica narrow:	Bold Helvetica narrow font is used for buttons, fields, and icons in a graphical user interface. Keyboard keys are also set in this font.

Format Conventions

This manual uses special formatting conventions to present sample procedures (syntax, code or programming examples) and sample input/output.

Sample Procedures

[Example 1 on page xxv](#) shows the formatting convention for sample procedures, code, and programming examples.

EXAMPLE 1: Code Sample

```
command RENUMBER
[ids=blocklist]=querysuperblock();len=length(ids);
[sids,idx]=sort(ids',{increasing});

for i=1:len
    blk=sids(i);
    modifyblock blk,{id=i}
SBADISPLAY, {refresh}
endfor
endcommand
```

Mouse Conventions

This document assumes you have a standard, right-handed three-button mouse. From left to right, the buttons are referred to as left, middle, and right. All instructions assume the left mouse button unless otherwise noted.

- click** Press and quickly release a mouse button. The left button is assumed if click is used without a button designation. For example, “click **OK**”.
- double-click** Click a button twice in quick succession.
- drag** Place the cursor over an object, then hold down the appropriate mouse button while moving the mouse. Release the button to complete the operation.

Note and Caution Conventions

Within the text of this manual, you may find notes, cautions, and warnings. These statements are used for the purposes described below.

NOTE: Notes provide special considerations or details which are important to the procedures or explanations presented.

CAUTION: **Cautions indicate actions that may result in possible loss of work performed and associated data. An example might be a system crash that results in the loss of data for that given session.**

Support

You can contact MATRIXX Technical Support in any of the three ways listed below. When Technical Support responds, you will be given a Call ID specific to the problem you have reported. Please record the Call ID and use it whenever you contact Technical Support regarding the issue.

- Submit a problem report via the ISI web site using the following URL:

```
http://www.isi.com/Support/MATRIXx
```

This is the preferred method, as it is the most traceable; your problem report will be automatically entered into our support database.

- Send an e-mail to `mx_support@isi.com`. We can serve you better if you mail us details on your configuration and the circumstances under which your problem occurred. We provide an ASCII file that you can use as a template for your e-mail to support; it can be found in:

```
$MATRIXX/version/v6support.txt
```

For example, if you have Xmath running you can use the following Xmath function call to copy the template to the current working directory:

```
copyfile("$MATRIXX/version/v6support.txt")
```

- Call 800-958-8885 (where 1-800 service is available) or 408-542-1930. Telephone support hours are 7:00 a.m. through 5:30 p.m. PST, Monday through Friday. We can respond more efficiently if you are ready to provide the information requested in `$MATRIXX/version/v6support.txt` at the time you call.

Using the ISI FTP Site

If your problem involves scripts or model file(s), Technical Support may ask you to FTP your files to us for further examination.

1. Connect to the ISI FTP site:

```
ftp ftp.isi.com
```

2. Log on as anonymous, and supply your e-mail address as the password.
3. Change to the `/incoming` directory:

```
cd /incoming
```

4. Use `put` or `mput` to specify the file(s) you are transferring. When the transfer is complete, quit.
5. Send an e-mail message to Technical Support that states the Call ID (if available), the exact name(s) of the file(s) you put in `/incoming`, and the approximate time you made the transfer; alternatively, call 800-958-8885 (where 1-800 service is available) or 408-542-1930 and provide this information. It will be a minimum of 15 to 20 minutes before the transferred file(s) will pass through the firewall.



1

Introduction

1.1 Overview

The SystemBuild™ design environment plays a central role in the MATRIX_x® Product Family (Figure 1-1).

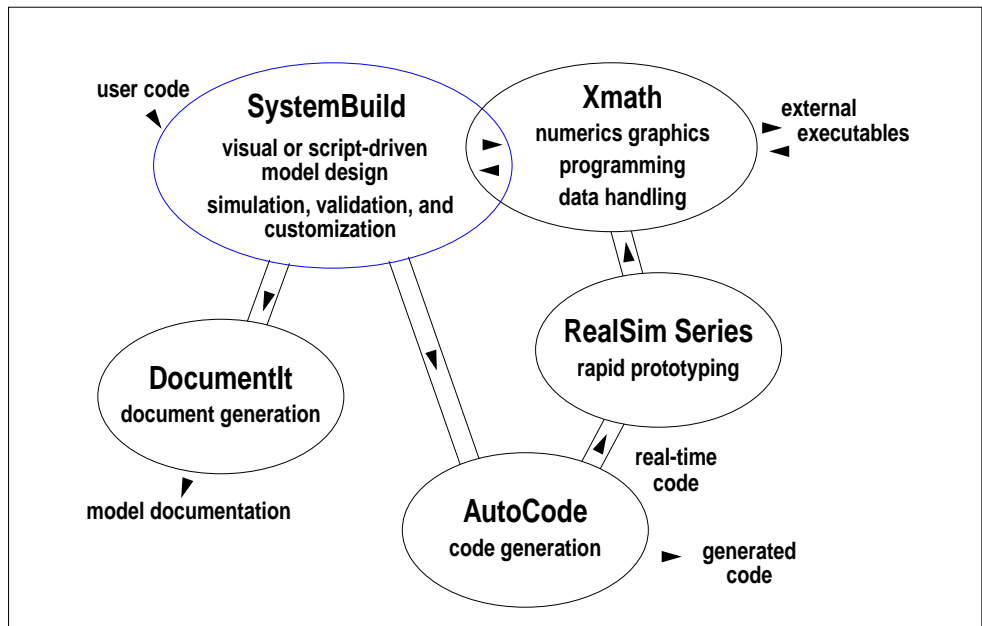


FIGURE 1-1 The MATRIX_x Product Family

SystemBuild provides a hierarchical catalog browser and other organization tools to help manage your models and model data. You can easily reuse or share models you create.

Whether your design is simple or complex, SystemBuild's user-friendly graphical interface and extensive library of blocks, utility functions, and tools make it the fastest way to express an engineering concept as a graphical model; a model design can be validated immediately using SystemBuild analysis and simulation tools.

Although SystemBuild capabilities are extensive, the interface is purposely open and flexible, so that you can do even more. You can implement your own commands and functions, create your own blocks, and link in external code. You can also customize the simulation environment.

This manual introduces you to SystemBuild's many features. It focuses primarily on model building and editing tasks, object relationships, and conceptual descriptions of complex topics such as analysis and simulation. For convenient descriptions of individual blocks, block dialogs, and the user interface, see the context-sensitive online help. For information on using the help, type **help help** in the Xmath command area.

2

Starting SystemBuild and Using the Catalog Browser

Xmath is the entry point to the MATRIX_x product family. SystemBuild makes use of Xmath's numerical analysis, graphics, data handling, and file management capabilities. SystemBuild is launched from Xmath. The first window to appear is the SystemBuild Catalog Browser.

A SystemBuild catalog is a complex storage structure that can contain models, model data, SuperBlock and block parameters, and other objects unique to SystemBuild. The Catalog Browser handles communication between SystemBuild, Xmath, and the operating system; it also manages the interrelationship of objects within the catalog. The Catalog Browser provides an interactive way to create, edit, view, and organize catalog objects. The browser is the entry point to the SystemBuild Editor.

This chapter guides you through basic Catalog Browser tasks.

2.1 Starting and Exiting SystemBuild

You must start SystemBuild from the Xmath command window.

Starting SystemBuild

In Xmath, select Window→SystemBuild, or type the following command in the command area:

```
build
```

The Catalog Browser will appear. The BUILD command also allows you to specify a SuperBlock name. This syntax is:

```
build "SB_Name"
```

SB_Name is the name of a SuperBlock in string form.

- If SystemBuild is not running, it will be launched, and a new continuous SuperBlock of the specified name will be launched.
- If SystemBuild is running, and a SuperBlock with the specified name is in the current catalog, the SuperBlock will be displayed in the editor.
- If SystemBuild is running, but the specified SuperBlock doesn't exist, a new continuous SuperBlock by that name will be created and displayed.

Exiting SystemBuild

To exit SystemBuild from the Catalog Browser, select File→Exit. You are asked if you want to save your work before exiting. If you answer yes, the Catalog Browser Save dialog will appear. Push the Help button on the dialog if you need assistance.

Exiting from the Catalog Browser will leave Xmath running. To exit all MATRIX_x processes at once, go to the Xmath command Window and select File→Quit from the Xmath Command Window, or, type Quit in the Xmath Command Window command area. Note, you will be prompted to save, but only a full save to `save.xmd` will be performed in this circumstance.

2.2 Loading Data

This section discusses Xmath and SystemBuild load methods, and uses an example to explain how to selectively load data using the Advanced Load dialog. The data loaded in [Example 2-1 on page 2-3](#) will be used throughout this chapter.

2.2.1 Xmath Load Command

The LOAD command can load an entire file, or selectively load Xmath, SystemBuild, or Usertype information. The basic syntax is:

```
LOAD {xmath, build, usertype} "fileName"
```

To see the online help for the load command, type **help load** in the Xmath command area.

2.2.2 Catalog Browser Load Dialog

In the Catalog Browser, select File→Load. Click the **Help** button for information on using this dialog.

EXAMPLE 2-1: Loading Portions of a Catalog

1. Copy a data file from the SystemBuild examples directory to your current working directory. In the Xmath command area, type:

```
copyfile "$SYSBLD/examples/manual/cb_ex1.cat"
```

This file will be referred to throughout the chapter.

2. To start SystemBuild from Xmath, select Windows→SystemBuild. The Catalog Browser and the SystemBuild Editor are launched.
3. In the Catalog Browser, select File→Load. When the Load dialog appears, select the file `cb_ex1.cat`. This catalog file contains the folders SuperBlocks, State Diagrams, DataStores, and Components; these objects are described in full elsewhere in this manual. The most common action is to load an entire catalog, however, in this example we'll load what we need a piece at a time.
4. To view the contents of the catalog before loading it, Click the **Advanced** button. The Advanced Load dialog appears; by default, an alphabetical listing of all SuperBlocks in the catalog is displayed on the right.

In the Table of Contents, each folder contains a listing of different object types. Click on each different folder to see what it contains.

5. Check the **Load Hierarchy** box. SuperBlocks can contain other SuperBlocks, forming a SuperBlock hierarchy. Checking this option ensures that if you load a single SuperBlock, all SuperBlock elements included in its hierarchy, including State Diagrams or DataStores, will be loaded as well (it does not affect objects other than SuperBlocks, such as components, libraries, STDs, or DataStores that are not included in diagrams).
 - a. Select "continuous automobile."
 - b. To select an additional SuperBlock, hold down the **Control** key and click on Cruise Control System. Note that multiple SuperBlocks are selected because the Hierarchy button is pushed. Continue selecting SuperBlocks until all SuperBlocks are selected.
 - c. Select **Apply** to load all selected SuperBlocks.
6. Select the Components folder. Select the component "nlinteg" and click **Apply** to load it.
7. Click **OK** to complete the advanced load.

Now the default appearance of the Catalog Browser (with SuperBlocks selected) is as shown in [Figure 2-1](#).

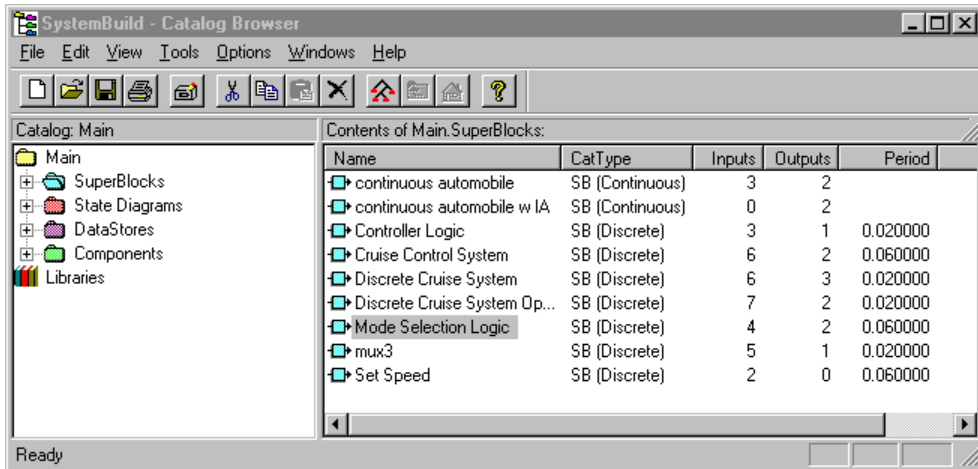


FIGURE 2-1 Catalog Browser View of Data from `cb_ex1.cat`

2.3 Understanding Catalogs

All data is loaded into the Main catalog (the Catalog name is shown directly above the left pane; see [Figure 2-1](#)). The Main catalog has four folders: SuperBlocks, State Diagrams, DataStores, and Components. These objects must all have discrete names. Libraries are shown as a separate catalog.

SuperBlocks

SuperBlocks are the most commonly used objects and will be examined throughout this chapter. A SuperBlock is a collection of primitive blocks that can include State Transition Diagrams (STDS), DataStores, and references to other SuperBlocks. A specific SuperBlock is only defined once in a catalog; references can then be made to it in multiple locations; these references can be thought of as instances of the SuperBlock.

When a SuperBlock contains references, they are referred to as “children”, and the SuperBlock itself is called the “parent”. This parent/child nesting forms a SuperBlock hierarchy. A SuperBlock at the top of a hierarchy is called a top-level SuperBlock; it has no parent. [Chapter 3](#) describes how to create SuperBlocks and

SuperBlock References; [Chapter 5](#) describes how the different types of SuperBlocks influence a model's timing.

State Diagrams

STDS graphically implement finite state machines. These objects are documented in the *State Transition Diagram User's Guide*.

DataStores

DataStores provide global data storage.

Components

A component is an encapsulated SuperBlock hierarchy. In the Catalog Browser, each component forms a separate catalog; the encapsulation of a SuperBlock into a component circumvents the restriction that all catalog elements must have a unique name. Although the component must have a unique name, objects encapsulated within it are not visible to other catalogs because a component has its own scope. Components are discussed in detail in [Chapter 17](#).

Libraries

Libraries are SystemBuild model files that are not loaded into SystemBuild.

2.4 Browsing a Catalog

This section introduces some of the ways you can view and organize catalog items. Load the model as described in [Example 2-1 on page 2-3](#). As seen in [Figure 2-1 on page 2-4](#), the Catalog Browser has two panes. The left pane displays the Catalog view, and the right pane shows the Contents view. There are two display modes using the Catalog and Contents views. When you select one of the labels in the Catalog view (SuperBlocks, State Diagrams, DataStores, etc.), a listing of all currently defined catalog items of the selected type will be displayed in the Contents view.

EXAMPLE 2-2: Working with Catalog Views

1. List all SuperBlocks:
 - Click on the label **SuperBlocks** in the Catalog view. The Contents view will display all currently defined SuperBlocks in the Catalog.
 - Click on the other catalog labels to view all currently defined Datastores, STDs, and Components in the catalog.


2. List all top-level SuperBlocks:

Each label in the Catalog view is preceded by a + symbol. To list all top-level SuperBlocks, click on the + symbol next to the label **SuperBlocks**.

Select one of the top-level SuperBlock names to view the SuperBlock contents in the Contents view. SuperBlocks are containers that organize and describe blocks. Blocks perform the actual work in a system; they are referred to as “functional blocks” or “primitives”. For primitives the Contents view displays a simple block icon and the block type.

3. Navigate a SuperBlock hierarchy:

Each top-level SuperBlock (shown as a folder preceded by a +) has child items you can view in the Contents view (the right window pane). Click on the + symbol to expand a subhierarchy. The symbol will change to a minus sign (-); clicking on the minus sign compresses the hierarchy.

- Select a folder and its contents will be listed in the Contents view.
- To expand all levels a hierarchy, select a SuperBlock and choose Edit→Hierarchy Select Mode. Alternatively, click the  icon.

4. Open the State Diagrams and DataStores folders.

DataStores and State Diagrams are primitive blocks, therefore the contents are displayed directly in the Catalog view; no information is displayed in the Contents view.

5. Expand the Components hierarchy.

Select `nlinteg`, then select View→Component Catalog, or select a component, right-click, then select Component Catalog. The Catalog view changes from Main to `nlinteg`, and the Contents view changes accordingly. To return to the main catalog, select View→Main Catalog, or right-click then select Main Catalog.

6. Adjust the sizes of the panes to accommodate your data:

- a. To change the width of the panes, widen the window or adjust the pane width. This task is slightly different on UNIX and Windows:

UNIX Click on the small square button towards the bottom of the dividing line between the Catalog and Contents view, and drag horizontally.

Windows Click directly on the dividing line between the panes and drag horizontally.

- b. In the Contents view, you can change the width of each column. Click directly on the dividing line and drag left or right.

7. Sort the Contents view:

By default, items in the Contents view are displayed in alphabetical order by name. Click on the heading of any column to reverse the sort order. For example, display the Controller Logic SuperBlock, and click on the ID header; the blocks are sorted by ID number, from lowest to highest.

Figure 2-2 shows the Catalog Browser with expanded Catalog hierarchies and modified contents view.

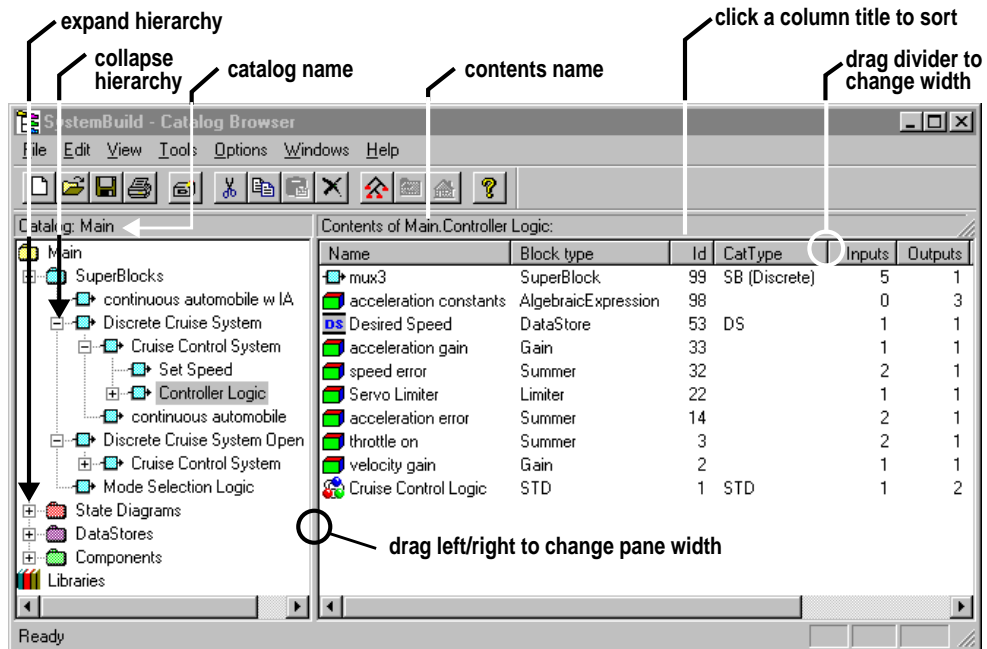


FIGURE 2-2 Expanded Hierarchies and Adjusted Contents View (Windows)

2.5 Saving Data

This section describes how to save the current model. You can save your catalog from both Xmath or the Catalog Browser. In addition, SystemBuild has an automatic save feature, AutoSave, that automatically saves your catalog data at a specified interval.

Xmath SAVE Command

You can save any combination of Xmath and SystemBuild data in ASCII or binary format using the Xmath SAVE command.

The general syntax is:

```
SAVE "fname" {xmath, build, usertype, superblock="name", hierarchy}
```

For a full description, see the online help for SAVE. Briefly, each keyword specifies a subset of data to be save. The default behavior is inclusive, as demonstrated in the following syntaxes:

```
SAVE {ascii}           Saves all to save.xmd (default) in ASCII. (The default
                        Xmath SAVE format is binary; you must specify ascii if
                        you need to share saved catalog files across platforms.)
```

```
SAVE "fname"          Saves all data to a file you specify.
```

```
SAVE "fname" {build}  Saves all SB data, including and usertypes (see
                        Section 15.5).
```

```
SAVE "fname" {superblock="sbname", hierarchy}

                        Saves the specified SuperBlock, all SuperBlocks in its
                        subhierarchy, and all Xmath data.
```

Catalog Browser Save Dialog

In the Catalog Browser, select File→Save As. This will raise the SaveAs dialog; the basic functionality is identical to that of the SAVE command, with the exception that the default save format is ASCII. Push the Save dialog Help button for a detailed explanation.

Save Selected

One capability is extended. The Xmath SAVE command allows you to specify a single SuperBlock hierarchy to be saved. In the Catalog Browser you can selectively save multiple SuperBlock hierarchies. Note, the selection(s) must be made *before* invoking

ing the Save dialog; when the dialog is raised you must specify the **Selected** selection mode. [Example 2-3](#) demonstrates this process:

EXAMPLE 2-3: Saving Data

In this example you will save the file you loaded earlier ([Figure 2-2 on page 2-7](#)) in two ways: saving all catalog data, and saving selected catalog items.

1. Save everything.

With nothing selected (click on the Main catalog to deselect any selected items), select File→SaveAs. Make sure that every option is set to **All** and the selected format is ASCII. Specify the following filename: `cb_ex1_sel.cat`. Click **OK**.

2. Perform a hierarchical selective save.

Start from the Catalog Browser.

- a. Select Edit→Hierarchy Select Mode.
- b. Click on Cruise Control System; all objects in this system hierarchy will be highlighted.
- c. Select File→SaveAs. In the Save dialog, go to the SuperBlock option and choose **Selected**. (If you don't enable this option, your preselections will be ignored and all data will be saved.) Specify the name `cb_ex1_ccs.cat`. Click **OK**.

The preselected objects will be saved in the specified file.

SystemBuild AutoSave Feature

SystemBuild has an automatic save capability that can be activated by resetting the SystemBuild defaults from the Xmath command area using the `SETSBDEFAULT` command. Two parameters, `autosavefile` and `autosavetime`, must be specified, and both must have value (they cannot be null/zero) for the autosave to take place. `autosavetime` is the number of seconds between saves; `autosavefile` specifies the name of the file in which the catalog data will be saved.

The AutoSave feature saves catalog data only; Xmath data or Usertypes ([Section 15.5](#)) are not supported in AutoSave.

Autosave is disabled by default; to enable it every time you start SystemBuild, add the `SETSBDEFAULT` command to your `startup.ms` file. For more on this feature, go

to the Xmath command area and type `help SETSBDEFAULT` then look at the help on the `autosavefile` and `autosavetime` keywords.

EXAMPLE 2-4: Using AutoSave

1. The following Xmath command will start AutoSave:

```
SETSBDEFAULT {autosavefile="autosave.cat",autosavetime=3600}
```

After entering the command, go to the Catalog Browser and select View→Refresh to update the new values. AutoSave is now set to save all Catalog Data to a file named `autosave.cat` every 60 minutes.

2. To change the save time interval to every two hours, type:

```
SETSBDEFAULT {autosavetime=7200}
```

Select update in the Catalog Browser to activate the new interval.

3. To check the current AutoSave settings type the following.

```
SHOWSBDEFAULT {autosavefile,autosavetime}
```

4. To turn off AutoSave, set `autosavetime` to 0 and refresh the Catalog Browser, type:

```
SETSBDEFAULT {autosavetime=0}
```

2.6 Accessing the SystemBuild Editor

2.6.1 Using the Catalog Browser Quick Access Menu

The Quick Access menu is a special feature to speed up your SuperBlock editing tasks; this menu has the same functionality as the Edit menu on the menu bar. To raise the Quick Access menu, do the following:

UNIX Right-click on an object; drag to select a menu item.

Windows Right-click on an object and release; the menu appears. Select a menu item.

Note that menu items may be disabled depending on the preselected object(s). In particular, if the hierarchy select mode is enabled, most Quick Access menu items will be disabled, as the selections are only valid if a single object is selected.

2.6.2 Opening a SuperBlock in the Editor

SuperBlocks listed in the Catalog Browser may be opened for editing using the Quick Access menu.

EXAMPLE 2-5: Opening a SuperBlock

1. In the Catalog Browser Catalog View, click on the SuperBlocks folder. This will list all currently defined SuperBlocks in the Contents view.
2. In the Contents View, select the first SuperBlock, "continuous automobile," To open this SuperBlock in an editor window, select Open in the quick Access menu. (Note, you cannot open a SuperBlock from the Contents view unless the SuperBlocks folder is selected in the Catalog view.)
3. SuperBlocks can also be opened directly from the Catalog view. Expand the SuperBlock hierarchy in the Catalog View, and use the quick access menu to open the SuperBlock Cruise Control System to a new editor window.

Edited SuperBlocks

In the Contents view, observe that the icon for the Edited SuperBlocks has a red check, indicating it is open in the editor. Each SuperBlock opened appears in a separate editor window.

- You can edit up to 20 SuperBlocks simultaneously.
 - Each editor window title bar will display the scope of the currently edited object. The scope specifies the object's position within its catalog. It also makes it possible to differentiate between SuperBlock or Component references that appear in the multiple editors.
 - The Catalog Browser Window menu provides the ability to iconify, deiconify, or close all editors. The bottom of this menu displays a list of the currently edited SuperBlocks. Because the name includes the scope, it may be truncated. To view the scope information, select Window→Catalog Object Properties.
 - If a catalog object (a SuperBlock, STD, etcetera) is currently open in an editor window, attempting to open the object will merely raise the editor that contains it. You cannot edit the same object in multiple windows.
-

2.6.3 Dragging and Dropping in the Catalog Browser

You can “drag and drop” objects from the Catalog Browser into the editor. This process is slightly different across platforms:

UNIX Middle-click on a SuperBlock icon and drag it into an existing editor window.

Windows Left-click on a SuperBlock icon and drag it into an existing editor window.

Make sure the editor window is visible before you start dragging.

Drag from the Catalog View

Dragging a SuperBlock from the Catalog view into an existing editor window saves and closes the previous SuperBlock. The dragged object becomes the currently edited SuperBlock.

Drag from the Contents View

Dragging an object from the Contents view copies it into the currently edited SuperBlock. If you drag a SuperBlock definition into the editor, a SuperBlock reference to the definition will be created instead.

EXAMPLE 2-6: Catalog Browser Drag and Drop Features

1. Select the top-level SuperBlock Discrete Cruise System in the Catalog view. Using the process defined for your operating system, drag the SuperBlock Discrete Cruise System to any editor window.
 2. You have loaded the SuperBlock into the editor; it has 3 blocks.
 3. Select Discrete Cruise System Open Loop. From the right hand pane, drag the SuperBlock reference Cruise Control System into the editor. Note, the SuperBlock does not open. A reference is created and it becomes part of the diagram.
-


2.6.4 Creating a New SuperBlock

New SuperBlocks can only be created from the Catalog Browser. To create a new SuperBlock, select File→New→SuperBlock. The SuperBlock Properties dialog will appear.

1. You must give the SuperBlock a unique name. Click **OK**.
2. To force an update from the editor, Select File→Update in the editor window.

2.7 Updating Catalog Browser Data

When you edit a catalog item from the Catalog Browser (using rename, cut, copy, etc.) the changes are immediately displayed in the Catalog. This is *not* true when you modify a catalog item from the SystemBuild Editor. By default, editor changes are not written to the catalog unless you change view to another object, load a file, or analyze a model. There are several ways to force an update:

- In the editor, type **Control-U**, select File→Update, or click the Update icon .
- In the Catalog Browser, type **F5**, or select View→Update.

Only items updated to the catalog will be preserved with the Save As or AutoSave features (see [Section 2.5](#)).

The remainder of this chapter consists of examples that modify the catalog used in previous examples. It emphasizes using two special Catalog Browser capabilities; the Quick Access menu (described in [Section 2.6.1](#)) and Drag and Drop. For details on performing the same tasks from the menu or icon bar, see the online help for this window (Help→Topics).

EXAMPLE 2-7: Creating New SuperBlocks and the Update Process

1. Create a new SuperBlock. In the Catalog Browser, select File→New→SuperBlock. The SuperBlock Properties form will appear.
2. Name the SuperBlock "NewSB", change the type to Discrete, and click **OK**. A new blank SuperBlock will appear in the editor window.
3. In the Catalog View, click on the SuperBlock label to display all currently defined SuperBlocks; note that NewSB does not appear.
4. Type **F5**. The SuperBlock NewSB will now appear in the Catalog Browser SuperBlock folder.

2.8 Editing Catalogs Using the Catalog Browser

The Catalog Browser is the gateway to the SystemBuild Editor. In its role as a data manager, the Catalog Browser can operate on catalog elements individually, or as hierarchies.

The editor is launched at the same time as the Catalog Browser, but it is not active unless a SuperBlock (or STD) is being edited. To activate the editor you can create a new SuperBlock or edit a SuperBlock that exists in the current catalog. Both tasks are performed from the Catalog Browser.

2.8.1 Using the Tools Menu

Catalog Browser tools, as listed in the Tools menu, operate on the model hierarchy. These tools are discussed in depth elsewhere, but you can try some of them using the current Catalog:

- SuperBlock transformations are fully explained in [Chapter 5](#).
- Component Tools (Make, Edit, Unmake) are fully explained in [Chapter 17](#).
- HyperBuild is discussed in the *HyperBuild User's Guide*.
- AutoCode, the AutoCode Code Generation tool is documented in the *AutoCode User's Guide* and *AutoCode Reference*.
- DocumentIt, the document generation tool is explained in the *DocumentIt User's Guide*.

EXAMPLE 2-8: Transforming a SuperBlock

This example uses the editing techniques explained in [Section 2.8.2 on page 2-15](#).

1. In the Catalog Browser catalog view, copy the SuperBlock “continuous automobile”, and paste it into the catalog.
2. Rename the copy “discrete automobile”.
3. Select discrete automobile, then select Tools→Transform. The Transform SuperBlock dialog will appear.
4. From the Type drop-down menu, select Discrete. Select Transform Initial Conditions. Click **OK**.

The new SuperBlock appears in the SuperBlock hierarchy.

2.8.2 Modifying a Catalog

EXAMPLE 2-9: Modifying Catalogs

1. Select the Controller Logic folder. From the Quick Access menu, select Copy. The object is copied to the clipboard.
 2. From the Quick Access menu, select Paste to paste the contents of the clipboard into the catalog. A folder named "Copy of Controller Logic" will be placed at the top level.
 3. Select Copy of Controller Logic, then select Rename from the Quick Access menu. The Rename dialog will appear; supply the name "Modified Controller Logic". Make sure **Rename All References** is checked, and click **OK**.
 4. Open the new block in the editor. Modify the SuperBlock as follows:
 - a. From the editor menu bar, select File→SuperBlock Properties. The Super-Block properties dialog will appear.
 - b. Change the Sample Period to 0.01, and the Sample Skew to 0.01. Click **OK**.
 5. Go to the Catalog Browser (from the editor you can select Window→Catalog Browser); select the SuperBlock hierarchy to view the block information for top-level SuperBlocks. Controller Logic and Modified Controller Logic still have the same Period and Skew Values.
 6. Because we are still editing this SuperBlock, the changes haven't been written to the Catalog Browser. In the Catalog Browser, select View→Update or type F5 to force the update.
-

3

Editing SuperBlocks

The SystemBuild SuperBlock editor, referred to as simply “the editor”, is an interactive design environment for creating and editing block diagrams. All block diagrams start with a SuperBlock that contains one or more blocks. SystemBuild can edit up to 20 SuperBlocks simultaneously, each displayed in a separate Editor window. Each editor window instance can display up to 199 blocks. In addition, the SuperBlock editor provides easy access to analysis and simulation tools.

As mentioned in [Chapter 2](#), the Catalog Browser manages the status and interrelationships of SuperBlocks, blocks, and all other objects in the catalog. The Catalog Browser initiates each editing session; from it you can create and edit a new SuperBlock, or open an existing SuperBlock and modify it, as described in [Section 2.6 on page 2-10](#). This chapter focuses on how to create and use SuperBlocks.

3.1 SuperBlock Hierarchies

Up to 199 blocks may be displayed in a SuperBlock editor window; although you can view the entire diagram by scrolling the editor window, it may be inconvenient. SystemBuild has a number of special features for creating a modular hierarchical system.

SuperBlocks provide a way to simplify large block diagrams or to group blocks that have a common purpose or common properties. The SuperBlock capability makes hierarchical systems and subsystems possible.

In the context of the SuperBlock hierarchy there are two types of SuperBlocks: a top-level SuperBlock, and a SuperBlock reference. A top-level SuperBlock has no parent; any SuperBlocks it contains are its children. A single SuperBlock can be used in multiple models within a catalog; each instance is called a *reference*; a change made in the original top-level SuperBlock will propagate to all references.

3.1.1 SuperBlocks and Functional Blocks

SuperBlocks do not perform direct functional actions. They are hierarchical entities that define the timing properties for functional blocks (and optionally other SuperBlocks) below them in the hierarchy. SuperBlock timing methods are: Continuous, Discrete, Enabled, Trigger, or Procedure. All non-continuous blocks have type-specific timing attributes. SuperBlocks also define the timing attributes of SystemBuild subsystems. Primitive blocks derive their timings from the parent SuperBlock. See [Chapter 5](#) for more on SuperBlocks and timing.

The operation of SuperBlocks in hierarchies is illustrated in [Figure 3-1](#), which shows how SuperBlocks may be nested, one within another, each containing functional blocks.

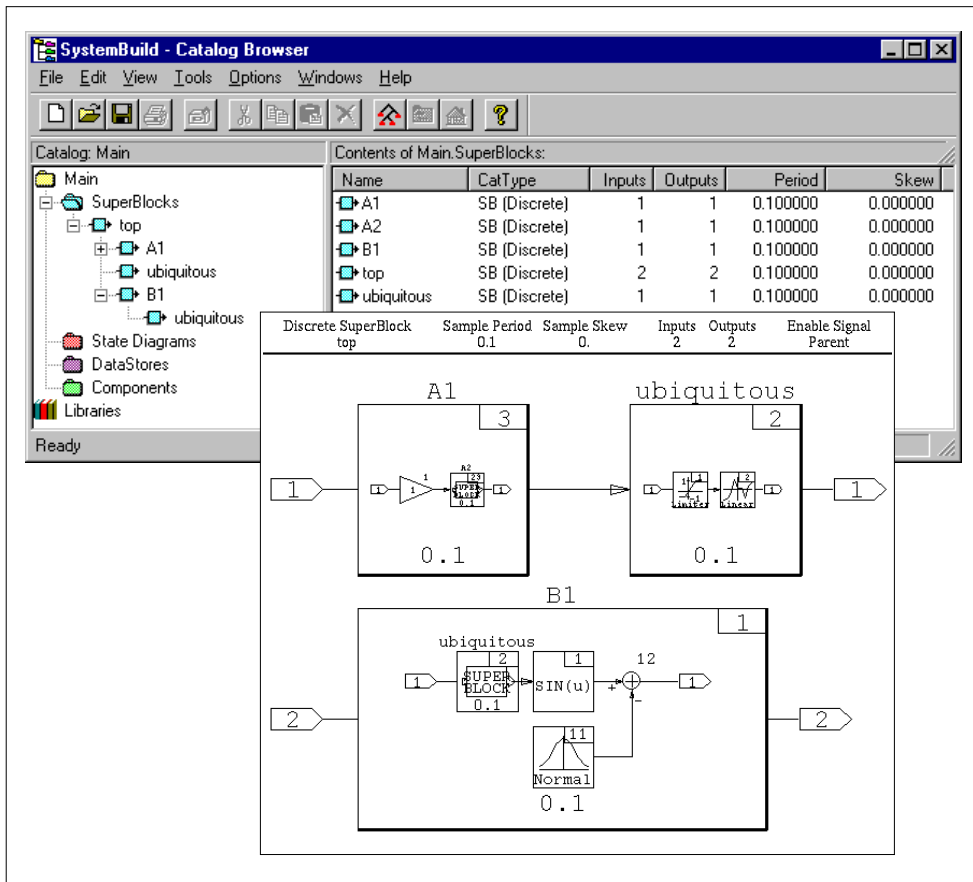


FIGURE 3-1 SuperBlock References in a SuperBlock Hierarchy

When a SuperBlock icon appears in a block diagram, the icon represents an instance of (or a call to) a SuperBlock with that name. Thus, any change in a SuperBlock goes into effect in all the places it is referenced. [Figure 3-1 on page 3-2](#) shows the SuperBlock “ubiquitous” occurring at different levels in a hierarchy.

3.2 Creating SuperBlocks

All SuperBlocks have the following properties:

- A SuperBlock must have a unique name (within the current catalog) that starts with an alpha character, is less than 32 characters long, and contains no punctuation characters (such as semicolons, periods, etcetera). A SuperBlock’s name is its sole method of identification.
- A SuperBlock must contain at least one block. The blocks within a SuperBlock can be basic functional blocks, or references to different SuperBlocks, components, DataStores, or State Transition Diagrams (STDs).
- A SuperBlock hierarchy is a global object (within a catalog), therefore its definition can be reused by calling it from within other SuperBlocks in your catalog; a block that calls a SuperBlock’s definition is called a SuperBlock reference to, or instance of, that SuperBlock hierarchy.
- A SuperBlock definition that is not referenced elsewhere in the catalog is called a top-level SuperBlock.
- A SuperBlock’s outputs cannot be directly connected to its inputs; the output signal must first pass through another block, see [Figure 3-2](#):

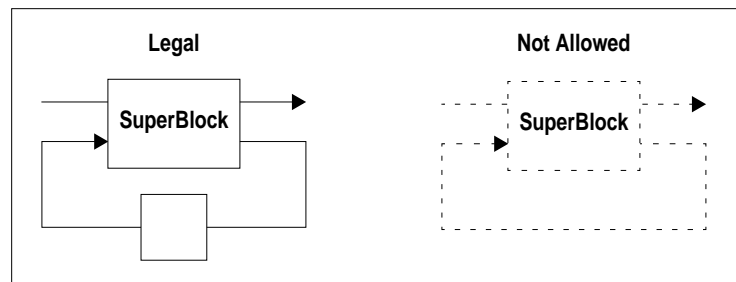


FIGURE 3-2 Legal and Illegal Connections

The methods for creating SuperBlocks are discussed in the following subsections.

3.2.1 Creating a New SuperBlock from the Catalog Browser

To create a new SuperBlock, go to the Catalog Browser and select File→New→SuperBlock. The SuperBlock Properties dialog (Figure 3-3 on page 3-4) will be raised. You must provide a name. A legal SuperBlock name is a string of no more than 32 characters. It must start with an alpha character, but it can also contain numbers, spaces, and underscores. Punctuation characters, such as period, comma, percent, or slash, are not allowed; if they are used they will be replaced with underscores. Because SuperBlock names must be unique within a catalog, a fatal error will be issued if an existing name is specified.

Click the **Help** button for a full description of each field. Click **OK** to finish.

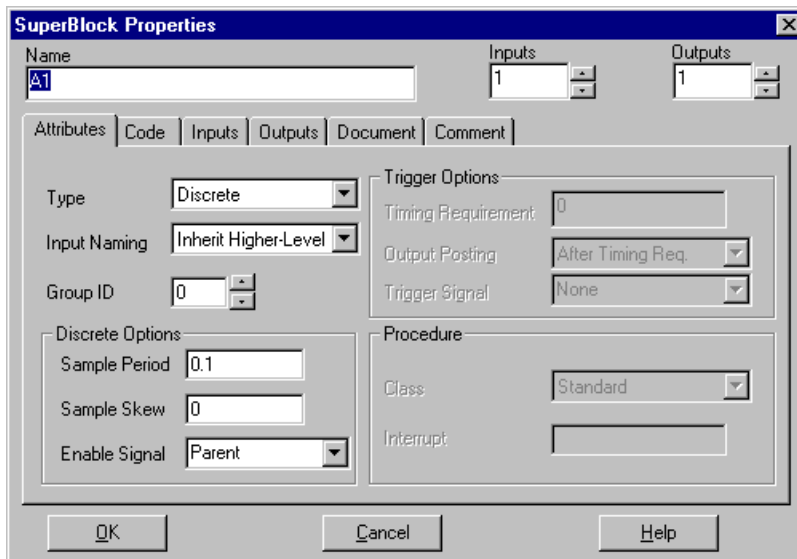


FIGURE 3-3 SuperBlock Properties Dialog

The new SuperBlock will be opened in the SuperBlock editor; it will initially appear as a top-level SuperBlock in the Catalog view. You must create at least one block in the SuperBlock. To force an update of changes from the editor to the Catalog Browser, type **Control-U**, select File→Update, or click the Update icon.

3.2.2 Making a New SuperBlock from Existing Blocks

You can create a new SuperBlock by grouping existing functional blocks and SuperBlocks in the SystemBuild Editor.

To create a SuperBlock, select the blocks that you want to move to a subhierarchy (the children of the SuperBlock), then select Edit→Make SuperBlock. The new SuperBlock will be given the default name makesb#, where # is a system-supplied digit that ensures a unique name. It will have the timing properties of the current parent SuperBlock, and will initially appear in the Catalog view as a top-level SuperBlock. To ungroup the blocks, select the SuperBlock then select Edit→Expand SuperBlock. Note, you cannot use Expand SuperBlock within a container block; for a description of container blocks, type `help container` in the Xmath command area.

Figure 3-4 on page 3-5 illustrates this process.

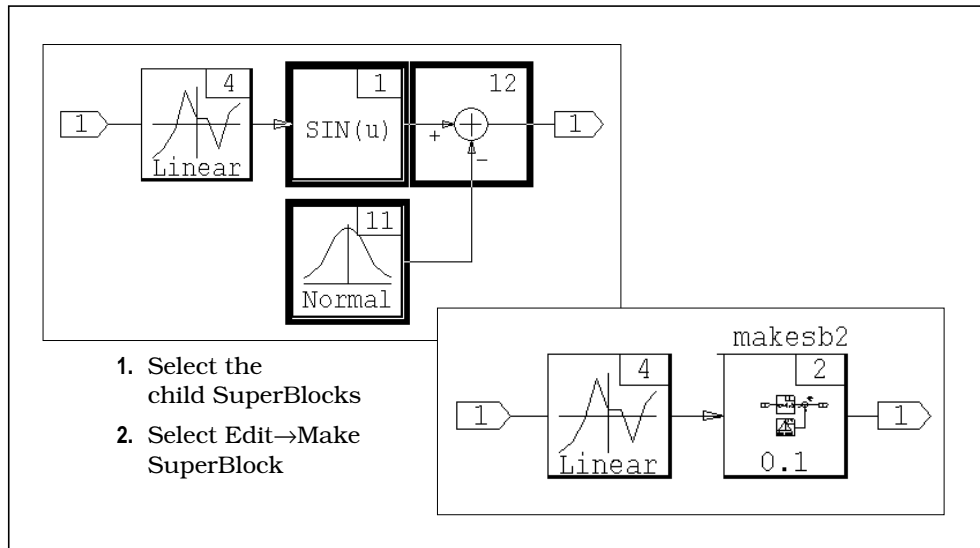


FIGURE 3-4 Making a SuperBlock from Existing Blocks

To display the contents of the SuperBlock in the icon, place the cursor on the SuperBlock and press **s** until the User or Alternate icon is displayed.

3.3 SuperBlock Dialogs

All the information to define a SuperBlock is entered through its block dialogs. Unlike other blocks, a SuperBlock has two kinds of block dialogs associated with it, the SuperBlock Properties dialog and the SuperBlock Block dialog.

3.3.1 SuperBlock Properties Dialog

This dialog first appears when a SuperBlock is being created from the Catalog browser. When a SuperBlock is being edited, you can raise this dialog with a single click on the SuperBlock ID bar (the strip of SuperBlock information that appears below the editor's icon bars and above the diagram work area), or by selecting File→SuperBlock Properties. [Figure 3-3](#) shows a SuperBlock Properties dialog.

The **Name**, **Inputs**, and **Outputs** fields are always visible. A SuperBlock must have a unique name. It must start with an alpha character and it cannot have more than 32 characters. Names can contain spaces and underscores, but spaces may be replaced by underscores. Punctuation characters such as period, comma, percent, or slash are not allowed, and will be replaced with the underscore character if they are used.

A SuperBlock is not required to have inputs and outputs. When they exist, however, each represents a signal, sometimes called a data channel, for the current block diagram. The number of inputs and the number of outputs are independent; they do not have to match.

The **OK**, **Cancel**, and **Help** buttons appear at the bottom of the form. **OK** accepts all changes and closes the dialog; **Cancel** disregards all changes and closes the dialog. Clicking **Help** raises the online help for the SuperBlock Properties dialog.

The SuperBlock Properties dialog has the following six tabs: Attributes, Code, Inputs, Outputs, Document and Comment. An overview of the contents of each tab is summarized in the remainder of this section.

Attributes Tab

The Attributes tab allows you to set the timing attributes of the SuperBlock; these attributes can be inherited by SuperBlocks lower in the hierarchy.

Type	The default type is continuous.
Continuous	Continuous modeling is specified for the processes of the SuperBlock (the sample rate is 0). This is the default. See Section 5.1 .
Discrete	SuperBlocks may run in discrete time, being either free-running or enabled. If discrete is selected, the Sample Period, Sample Skew and Enable Signal fields are active; if an Enable Signal is specified the block will execute only when the Enable Signal is asserted. See Section 5.2 .
Trigger	The SuperBlock is triggered for execution (one-shot) by a specified trigger signal. See Section 5.3 .
Procedure	Procedure SuperBlocks allow users to implement stand-alone procedures. They may inherit their timings from a parent (Standard Procedures) or run untimed (Asynchronous). Possible procedure classes are standard, startup, background, interrupt, macro, and inline. See Section 5.4 .
Input Naming	Controls the display of labels in the block diagrams in the SuperBlock subhierarchy. By default all labels are inherited from the parent SuperBlock. If Enter Local Label Names is specified, local names are used. If you are specifying a top-level SuperBlock you must select Enter Local Label Names if you wish to specify labels in the Inputs tab.
Group ID	Optional processor group ID for the current SuperBlock hierarchy. Disabled if the type is continuous, enabled otherwise. Default is 0. See Section 7.12.1 for an in-depth discussion.

Code Tab

The Code tab is enabled when the SuperBlock type is Procedure and the procedure class is Macro. Your macro is specified in the Macro Name editing area. You can type directly in the editing area, or, you can push the **Editor** button to raise the current editor. When you exit the editor, the contents will appear in the Macro Name editing area.

The default editor for UNIX is vi. The default editor for Windows is Notepad. To change the default editor, you must set the environment variable `EDIT_COMMENT`

from the operating system command line, or within a startup script. See the following examples:

UNIX `setenv EDIT_COMMENT emacs`

Windows `set EDIT_COMMENT=C:\Program Files\Accessories\wordpad.exe`

The new text editor will not be attached until you restart Xmath. See [Section 6.3](#) for further details.

Inputs Tab

An input is a data channel or signal. The Inputs tab gives you the opportunity to add a label or name the signal and specify its data type.

Input Label The text you enter in the Input Label field can appear in the block diagram if Show Labels (on the Display tab) is activated for the functional block(s) that receive the inputs. If a SuperBlock receives the inputs, Show Labels will display the labels, and if Propagate Labels is enabled (available on the SuperBlock Block Display tab only) the parent labels will be propagated down the hierarchy.

Labels also appear in the analyze output listing and the DocumentIt documentation. The label you specify on the Inputs tab is also displayed on the Document tab.

Input Name You can specify a name for the input signal. This name will be associated with the signal in the AutoCode code listing; it has no impact on the block diagram. See the AutoCode User's Guide.

**Input
DataType** This field allows you to assign a datatype for each input. Because datatypes are normally set in functional blocks, these settings may be ignored or overridden in the analysis phase. They will be used if this is a top-level SuperBlock or if the SuperBlock type is Standard Procedure. To assign a datatype on UNIX, click in the field; a menu of types will appear. On Windows the types are in a drop-down box; press the triangle beside the field to display the menu. See [Section 4.5 on page 4-19](#).

Input Radix This field is used solely for fixed-point data. If the Input DataType is set to a signed or unsigned type, you can specify an input radix. See [Chapter 15](#).

- Input UserType** If you want to specify a User Defined Type (usertype) for this input, specify it in this field. See [Section 15.5](#). As with DataTypes, the type may not be relevant for functional blocks within the diagram. Procedure SuperBlocks lower in the hierarchy can inherit these values, however.
- Input Scope** The input scope can be set to either local (the default) or global. It is only pertinent when the current SuperBlock type is Procedure and you are generating code. The scope determines whether data in Procedure SuperBlocks will be global or local in the generated code. See the *Auto-Code Reference* for details on signal scoping.

Outputs Tab

The Outputs Tab is read-only. SuperBlock output labels appear in output order, and the name assigned is the name (if any) of the functional block the signal last passed through, followed by the number of the signal (if there were multiple outputs from that block).

Document Tab

The document tab displays a spreadsheet that allows you to describe the input signal by assigning text or values to each field. The Input Label and Input Name fields are linked to the Inputs tab; a change in this location will occur there as well. Information in this tab has no simulation or code generation effect; it is extracted to create documentation when the DocumentIt document generation tool is used.

Comment Tab

The comment tab is an editing field. You can attach a comment that applies to this SuperBlock and its subhierarchy, if applicable. The contents of this field may also be accessed by DocumentIt. For an explanation of the Comment tab, click Help in the SuperBlock properties dialog, then follow the hypertext link to the Comment tab discussion. To change the default comment editor, see [Section 6.4](#).

3.3.2 SuperBlock Block Dialog

The SuperBlock Block dialog is also referred to as the SuperBlock Reference or Instance dialog. It controls information specific to an instance of a SuperBlock within a block diagram; a reference can be made using either of the methods in [Section 3.4 on page 3-11](#).

To view the SuperBlock Block dialog, select a SuperBlock icon and press Return. See [Figure 3-5 on page 3-11](#) for an example of a SuperBlock Block dialog.

Name	<p>The name field contains the name of the SuperBlock definition. If this is a new instance (a SuperBlock icon pulled from the palette browser) the default name is <code>_SB</code>. Spaces or underscore characters may be used as word separators in SuperBlock names. Caution should be used in employing spaces, however, because processes other than <code>MATRIX_x</code> may remove or replace space characters arbitrarily.</p> <p>When you enter a name, the system checks to see if the named SuperBlock exists in the catalog. If it does, a reference is made and the undefined SuperBlock block icon becomes an instance of the named SuperBlock; when the Catalog is updated, the reference will appear in the hierarchy. If it does not exist, the block is given the name, but it remains undefined and does not appear in the hierarchy.</p>
Inputs and Outputs	<p>If the name field is the default, or specifies the name of a SuperBlock that does not exist in the current catalog, you can specify inputs and outputs.</p> <p>If this is a reference to an existing SuperBlock, the number of inputs and outputs cannot be altered.</p>
ID	The block ID of the SuperBlock block icon.
Instance Name	You can supply a local name for this reference. If an instance name is used, it is appended to the SuperBlock definition field and surrounded by parentheses. For example, if a reference to a SuperBlock named System is given the instance name "variation1", the name above the SuperBlock reference will be "System (variation1)".
Xmath Partition	Use this field to specify the name of an Xmath partition this SuperBlock instance will use for loading and saving data.

The SuperBlock Block diagram has Parameters, Inputs, Outputs, Comment, Icon, and Display tabs. The Parameters tab is disabled. Fields in the other tabs behave the same as the equivalent tabs on functional blocks (see [Section 4.3 on page 4-5](#)). The single exception is found on the Display tab.

The Propagate Label option can only be seen on the SuperBlock block, the Condition block, and the BlockScript block. If enabled, this option dictates that labels from the parent SuperBlock will be propagated to child SuperBlocks in the Subhierarchy, and that signals passing through the SuperBlocks will pass the reference's labels to the following blocks.

3.4 Creating a SuperBlock Reference

A SuperBlock definition is saved in the Catalog hierarchy. References are local instances of a SuperBlock. If the SuperBlock is currently open in the editor you can edit the SuperBlock properties; any changes are global and will affect the model at every instance, SuperBlock instances are represented by a block; because the definition of the SuperBlock is elsewhere, you can only edit parameters local to the block, such as the name, labeling, and icon appearance. The SuperBlock reference dialog is shown in [Figure 3-5](#); compare this to the SuperBlock Properties dialog in [Figure 3-3 on page 3-4](#).

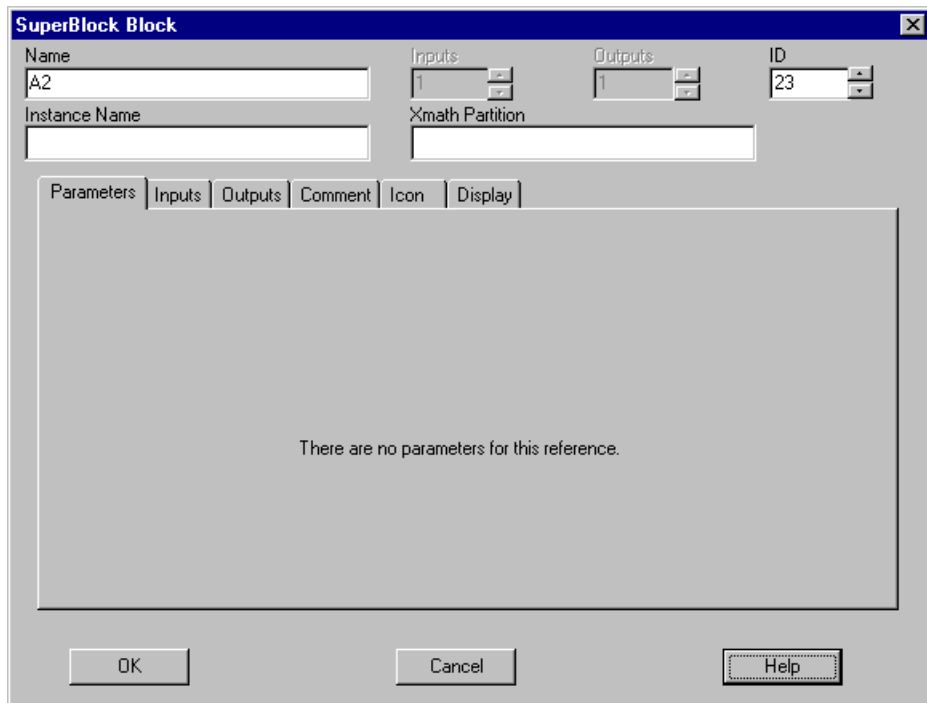


FIGURE 3-5 SuperBlock Block Dialog that References an Existing SuperBlock

You can form a SuperBlock reference from either the Catalog Browser or the Editor.

CAUTION: If a SuperBlock definition is removed from the catalog, references to it will remain in the block diagram, connected as before, but all information provided by the deleted definition is lost. This means the SuperBlock reference reverts to the default state

(a continuous SuperBlock with default values). This is also true if a definition is renamed (Section 3.4.2 on page 3-15) but references to it are not. To create valid references in this situation, you must create a new SuperBlock definition with the same name, or, rename the references to refer to valid SuperBlocks.

Creating a Reference from the Catalog Browser

If a SuperBlock exists in the catalog, you can create an instance by dragging the SuperBlock from the Catalog Browser Contents view and dropping it in the editor:

1. To view all the SuperBlocks in the current catalog, click on the SuperBlocks folder.
2. From the Contents view (the right side) select a SuperBlock icon and drag it into the currently edited SuperBlock; on UNIX the select and drag are done with the middle mouse button; Windows platforms use the left mouse button.

When you drop the reference a SuperBlock block appears in the editor, as shown in [Figure 3-6](#).

Creating a Reference from the Editor

1. Double-click in empty space to raise the Palette Browser.
2. From the SuperBlocks hierarchy, drag a SuperBlock icon into the Editor (on UNIX use the middle button; on Windows use the left button). The SuperBlock block icon will show Undefined by default.
3. To edit the block dialog, select the block and press Return.
 - To reference an existing SuperBlock, supply the name of a SuperBlock present in the catalog and any other parameters, and press return. The block's timing attributes are taken from the referenced SuperBlock; the SuperBlock reference name will be the name of the SuperBlock, followed by the name of the instance (if any) in parentheses.
 - You can leave the SuperBlock reference undefined, but you must put at least one primitive block in it as a placeholder block (it doesn't have to be connected to anything).

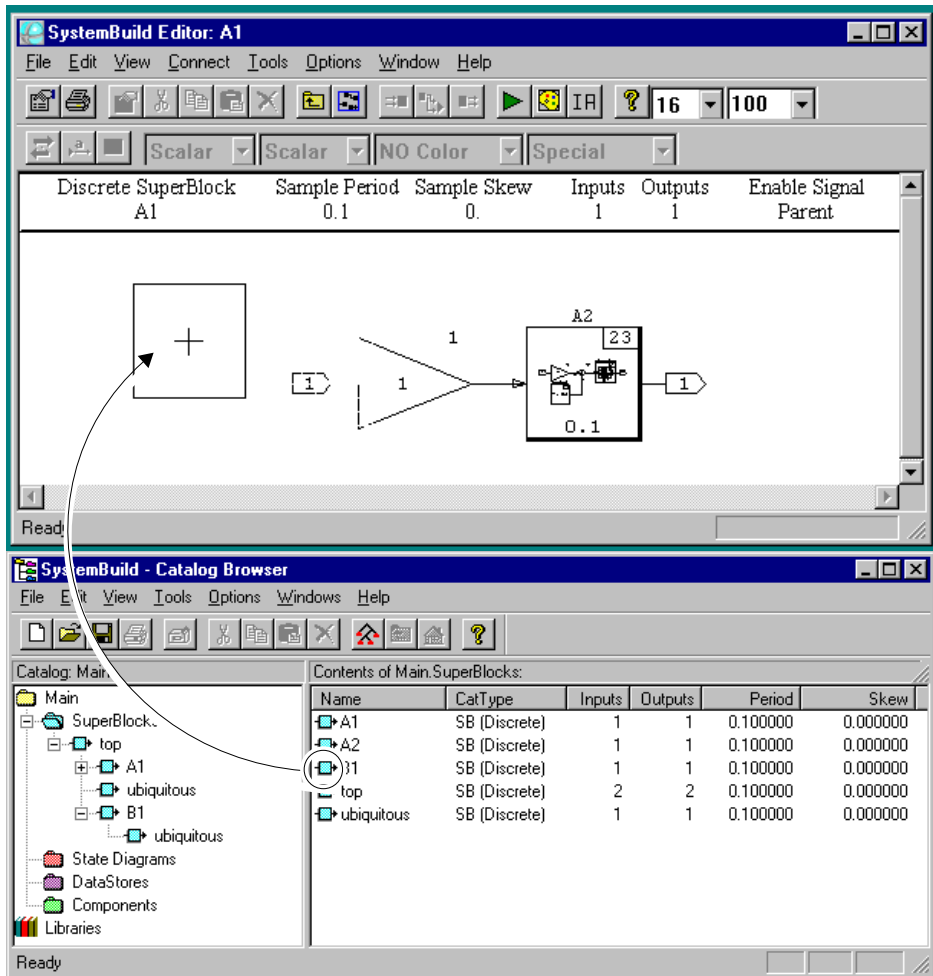


FIGURE 3-6 Drag from the Catalog Browser Contents View, Drop in the Editor

3.4.1 Creating a Copy of a SuperBlock

You can create a copy of a SuperBlock in the SuperBlock Catalog from the Catalog Browser or the Editor.

Creating a Copy with Copy and Paste

From the Catalog Browser, you can copy an existing SuperBlock, paste it into the current catalog, and rename it.

1. Select an existing SuperBlock.
2. Right-click to raise the Quick Access menu, and select Copy.
3. Right-click to raise the menu again, and select Paste.

The new SuperBlock is named “Copy of <sbName>”, where sbName is the name of the selected SuperBlock; the original SuperBlock remains in the hierarchy and the copy initially appears as a top-level SuperBlock at the end of the SuperBlock hierarchy. This process is illustrated in [Figure 3-7 on page 3-14](#). To rename the copy from the Catalog Browser, select it then select Edit→Rename.

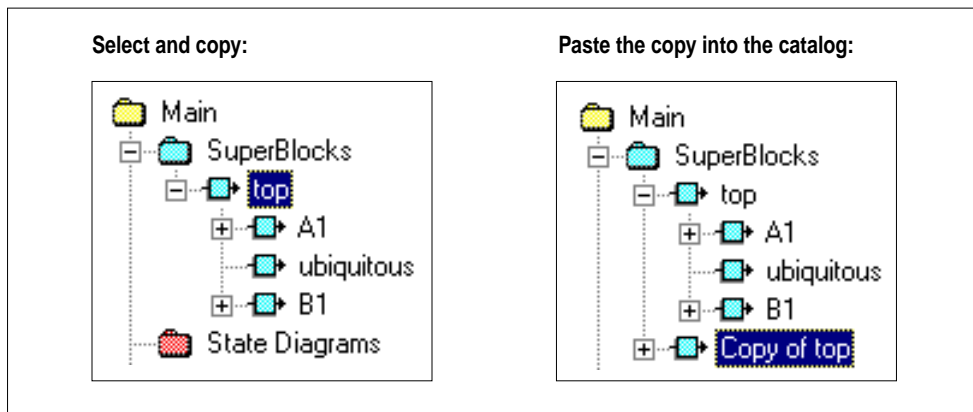


FIGURE 3-7 Copying a SuperBlock

Creating a Copy by Modifying the SuperBlock Properties

You can edit a SuperBlock’s properties if the SuperBlock is currently displayed in the editor. Single-click on the SuperBlock ID bar to raise the SuperBlock Properties dialog ([Figure 3-3 on page 3-4](#)). If you change the SuperBlock **Name** field from this dialog, a copy of the current SuperBlock with the new name will appear in the Catalog. The original definition and any references to it are unaffected.

3.4.2 Using Rename

Replacing a SuperBlock with Catalog Browser Rename

Because the Catalog Browser operates on a global level, renaming a SuperBlock from the catalog browser will result in the destruction of the original.

- If you rename a SuperBlock and specify **Rename All References**, the SuperBlock definition and all references to it will be given the specified name. The original is removed from the catalog. This is demonstrated in [Figure 3-8 on page 3-15](#), where the SuperBlock “ubiquitous” has been renamed to “U1”.
- If you rename a SuperBlock and do not rename references, a copy of the original SuperBlock with the new name appears at the top-level of the SuperBlock hierarchy. All references to the original SuperBlock retain the old name but become undefined; they will not appear in the catalog. [Figure 3-9 on page 3-16](#) shows the effect of “ubiquitous” being renamed to “Z1”. Although the undefined SuperBlocks do not appear in the catalog, they will remain in the block diagram, and they will still be named “ubiquitous”.

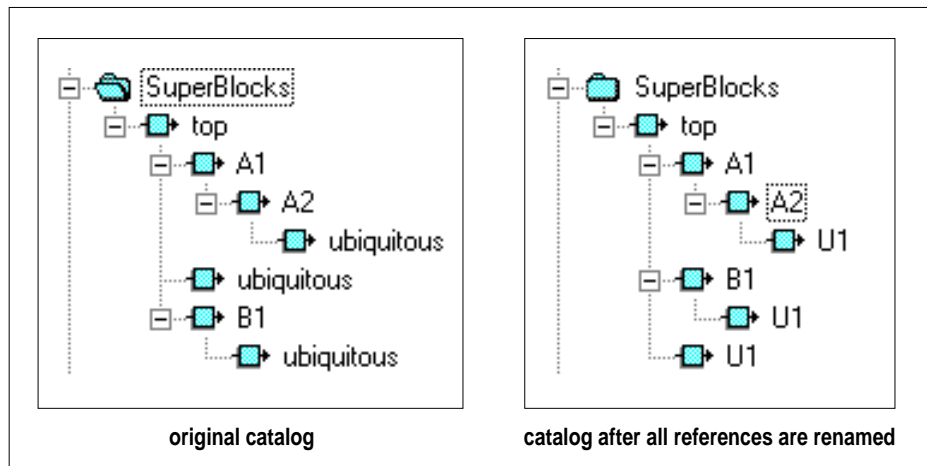


FIGURE 3-8 Renaming a SuperBlock and All References

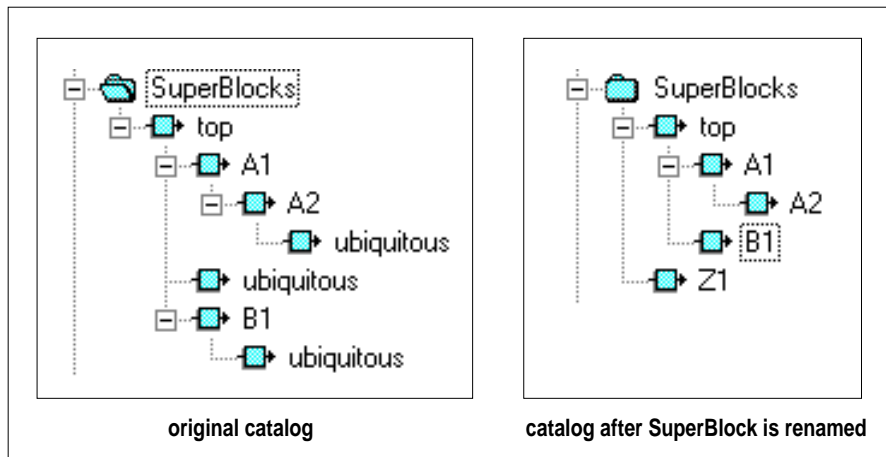


FIGURE 3-9 Renaming a SuperBlock Without Renaming References

Creating a Copy with Rename

- When you change the **Name** field from a reference's SuperBlock block dialog (Figure 3-5 on page 3-11) only that block is renamed.
- If you rename a SuperBlock from the Editor, by simply opening its field in the SuperBlock Block dialog, a call to an undefined empty SuperBlock with the new name appears at the same spot in the hierarchy. The original SuperBlock becomes a top-level SuperBlock.
- If you rename a SuperBlock from the Catalog Browser (select a SuperBlock, then select Edit→Rename) you will be given the option to rename all references to the new name.

4

Editing Blocks

The SuperBlock editor operates on a SuperBlock that has been loaded into the Catalog Browser ([Section 2.2 on page 2-2](#)) or created using one of the methods in [Section 3.2 on page 3-3](#). While SuperBlocks control the timing attributes of subsystems, functional blocks (also called primitive blocks) operate on a signal's value. A block diagram can include functional blocks, SuperBlock references ([Section 3.4 on page 3-11](#)), connections between blocks, external connections between blocks and the SuperBlock's inputs and outputs. This chapter contains simple examples that demonstrate how to define, connect, and modify blocks and the block diagram.

Individual blocks are described in the online help. In the Xmath command area, type `help blocks` to see a list of blocks organized alphabetically and by palette.

4.1 Creating Blocks

Once a SuperBlock is created, you can access the SuperBlock Editor ([Section 2.6 on page 2-10](#)), then create a model by placing blocks within the SuperBlock. The most common way to create a block is to drag a block icon from the Palette Browser into the SuperBlock Editor. (You can also create blocks using SystemBuild Access functions and commands, as explained in [Chapter 13](#) and the online help.)

A single palette browser supports all Editor windows. It can be accessed from the Editor by any of the following methods:

1. Double-click the left mouse button in an open area of the Editor window.
2. Click on the **Palette** icon in the [Edit Toolbar](#).
3. Select Window→ Palette Browser.
4. Move the cursor to empty space and press `d`.

On UNIX systems, drag the block from the palette to the workspace with the middle mouse button. On Windows systems, use the left mouse button.

By default the Palette Browser displays the ISI Main palette; it can also display custom blocks and palettes, as described in [Chapter 18](#). For a full description of Palette capabilities, including loading and deleting palettes, see the Palette browser online help.

Under some circumstances you will not be able to drag a given block into the editor. Typically this occurs when you try to instantiate a strictly continuous or strictly discrete block in a SuperBlock with conflicting timing. You must change the SuperBlock type or select another block.

Once a block is instantiated, it is defined from the block dialog. There are three ways to raise the block dialog:

- position the cursor over a block then press **Return**
- select a block and click the Block Properties icon on the Editor toolbar
- select a block, then select **Edit**→**Block Properties**

Only one block dialog can be open at any given time, regardless of the number of editors. The available tabs and fields vary from block to block, but in general, the settings and values used to operate on an input signal are found on the parameters tab. [Figure 4-1](#) illustrates some block properties that can be displayed in the editor.

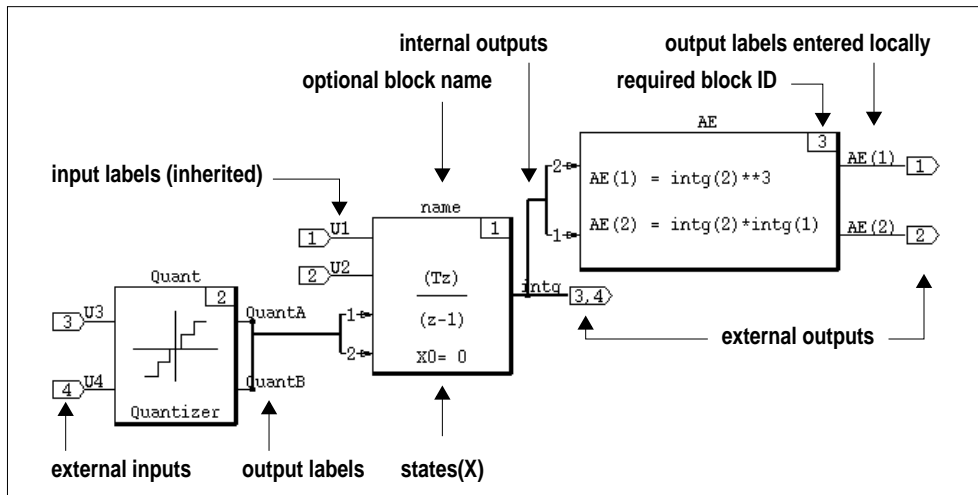


FIGURE 4-1 Block Properties Visible in the Editor

For block-specific information, see the online help. This chapter focuses on elements common to all blocks, e.g., tabs, fields, and how to use them.

4.2 Block Dialog Elements

This section discusses graphical elements (controls) found on SuperBlock and block dialogs. All interactive dialogs have common graphical elements that provide clues about how data should be entered. Data entry can take many forms. Depending on the block type, you can type in strings and numeric values, Xmath variable names that represent values, or Xmath statements that will calculate a value. You can also choose settings by selecting dialog-specific menu items, or enabling or disabling check boxes.

In SystemBuild, dialogs adapt to the block environment and settings whenever possible. For example, if a field is not applicable in the current environment, there will be a visual indication, as shown in [Figure 4-2 on page 4-4](#).

[Figure 4-2](#) also demonstrates the principle of consistency between fields. Here, nine inputs have been specified, so the dialog supplies nine fields for each column on the Inputs tab. SystemBuild also synchronizes the Outputs tab fields and the Document tab fields, to match the inputs and outputs specified. Each block has its own parameters and its own dependencies between fields; these are discussed in the online help for each individual block.

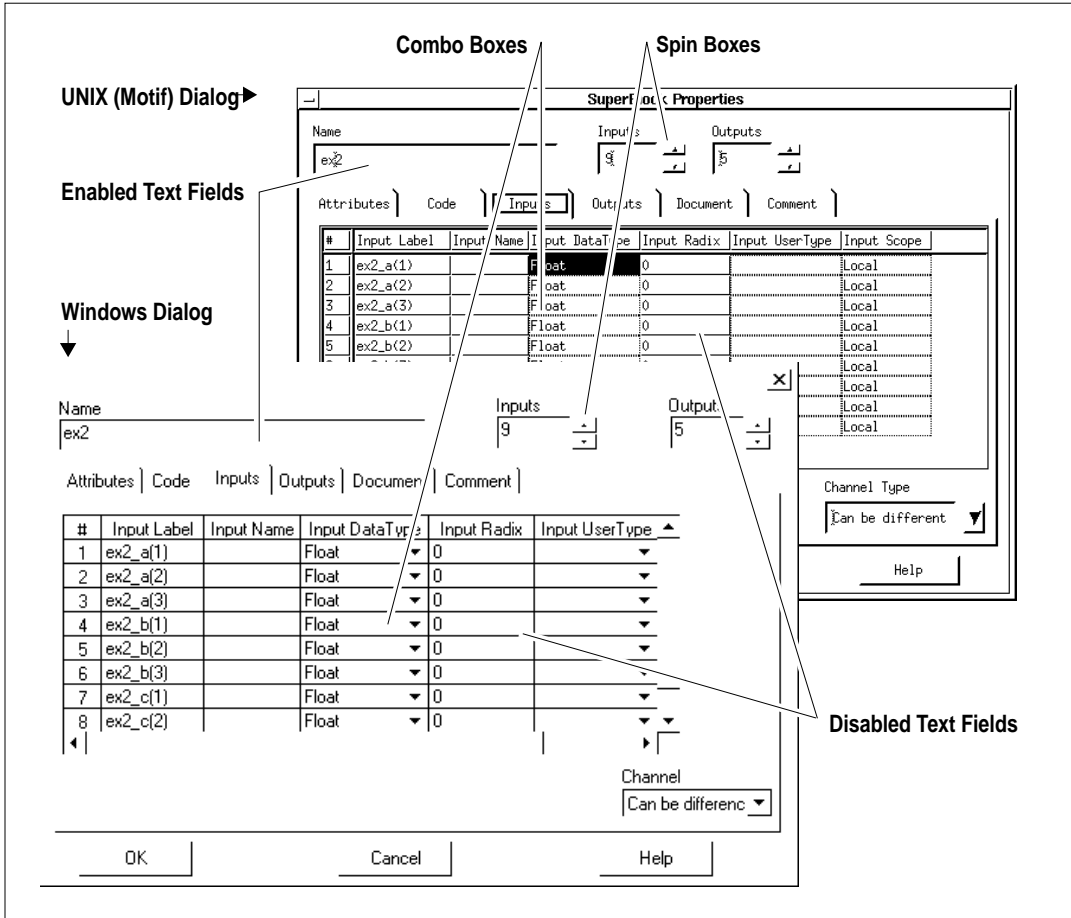


FIGURE 4-2 Motif and Windows Versions of the SuperBlock Dialog.

Dialog contents are the same across platforms, but there are small differences in the implementation. These differences are summarized in [Table 4-1](#).

TABLE 4-1 Cross Platform Widget Appearance

	UNIX	Windows
Enabled Text Fields	Title is black. Type directly into the field.	Field is white. Type directly into the field.
Disabled Text Fields	Title is grayed out.	Field is grayed out.
Combo Box (drop-down box)	Click on the field to raise a menu, drag to highlight an item, then release.	Click on the down arrow to raise a menu. Click on a menu item to select it.
Spin Box	Click the up arrow to increment the value; click the down arrow to decrement.	Click the up arrow to increment the value; click the down arrow to decrement. To rapidly advance the counter, click down on the arrow and hold; release when the desired value is reached.

The online help documents dialog box navigation and shortcuts. To see this information, type **help shortcuts** in the Xmath command area, and view the Block Dialog Shortcuts.

4.3 Block Dialog Fields

To access a block dialog, position the mouse over a block then press **Return**. Most block dialogs have the fields **Name**, **Inputs**, **Outputs**, **States**, and **ID** across the top. The block type is displayed at the top of the dialog window frame.

Name A name is optional for functional blocks. A legal name is an alphanumeric string that starts with a alpha character and contains no more than 32 characters. The default is blank (no name).

Inputs The number of data flows or signals serving as inputs to the block.

Outputs The number of data flows output by the block.

- States** Dynamic blocks and certain other blocks have a number of “memory” elements, referred to as *states*; in dynamic blocks the number of states is determined by the order of the dynamics.
- ID** This block number will be given a default value by the system, but can be changed to any unused number in the range [1: 199].

Fields that do not apply may be missing or grayed out, as shown in [Figure 4-3](#).

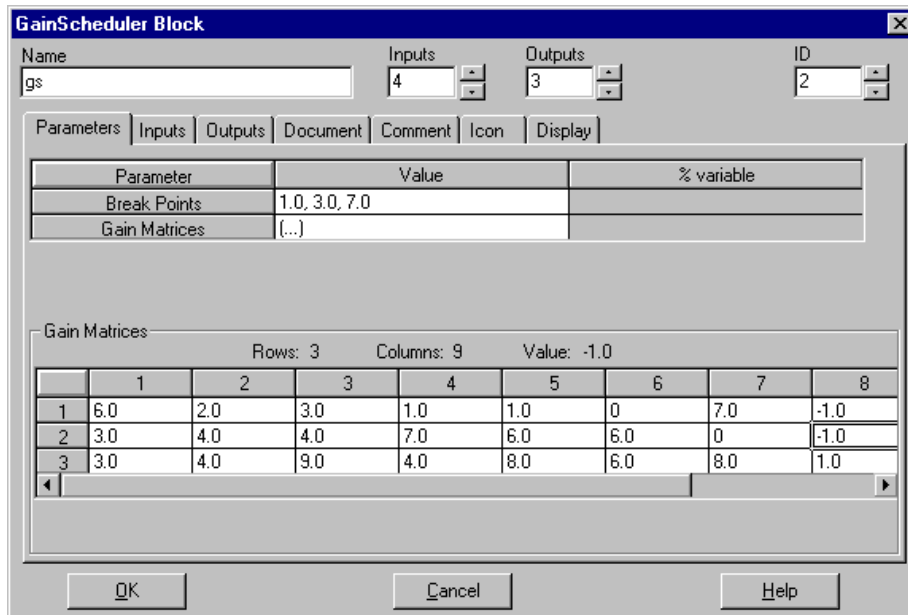


FIGURE 4-3 Gain Scheduler Block Dialog, Parameter Tab View

All blocks have **OK**, **Cancel**, and **Help** buttons across the bottom. Clicking the **Help** button raises the help for the current block in the `MATRIXx` Help window. The block help focuses on the Parameters tab fields.

The behavior of all other tabs is consistent among blocks. The nine possible tabs, [Parameters](#), [Code](#), [Inputs](#), [Outputs](#), [States](#), [Document](#), [Comment](#), [Icon](#), and [Display](#), are discussed in this section. To view a tab, click on the tab name.

4.3.1 Parameters

The Parameters tab fields vary widely among blocks. Data or settings information for the block's internal function are supplied here. Depending on the block, Parameters tab fields may require strings, vectors, or matrices. Numeric input may have specific datatype requirements. [Section 4.2 on page 4-3](#) reviews the different data entry methods. It is common to have block-specific interdependencies between fields. For example, in [Figure 4-3 on page 4-6](#), the number of elements in the Break-points field (three) determines the number of rows in the gain matrix.

4.3.2 Code

When present, the Code tab is used to accommodate multiple lines of text input. Typical uses for the Code tab are equations, expressions, or logical statements. Examples of blocks that have a Code tab are: AlgebraicExpression, LogicalExpression, Condition, BlockScript, IfThenElse and While, and the FuzzyLogic block. The syntax for instructions placed in the Code tab is block-specific, so be sure to view the on-line help for each block before using the Code tab. To change the editor in the code tab, see [Section 6.3](#).

4.3.3 Inputs

For primitive blocks, this tab displays the Input Name and Input Signal labels, if any, inherited from previous blocks.

For SuperBlocks, this tab has Input Labels (which can be displayed in the Editor), and Input Names (which cannot be shown in the editor). The Label is attached to the signal, which enters the diagram as an external input.

Input names are optional, and used exclusively for generated code (see the *Auto-Code Reference*). Input names are specified at the SuperBlock level as Input Labels, then propagated to SuperBlocks or blocks further down the hierarchy. Output labels, by contrast, are specified within each elementary block.

Examples of these fields are shown in [Figure 4-2 on page 4-4](#). See [Section 4.4.2 on page 4-14](#) for tips on entering labels or names.

Input Name A legal name has up to 32 characters, starts with a letter, and can contain the characters A-z, the numbers 0-9, and underscores. The Input Name will appear in the AutoCode-generated code for the block or SuperBlock.

Input Signal For primitive blocks, this read-only field shows the names of the external inputs to the block. The numbers are the pin numbers of the inputs, numbered from the top of the block.

If the block is a SuperBlock, you can make the field writable as explained in [page 4-15](#).

4.3.4 Outputs

The Outputs tab is used to label and format block output data. You can display and modify several output-related fields, including labels, names, and datatypes. Also, if you choose a fixed-point data type, a **Radix** field and related fixed-point information (read-only) are displayed near the bottom of the dialog.

Note that while input labels are inherited, output labels and names must be specified anew for each block. Like input names, output names do not appear in the block diagram; they are of importance for code generation only.

#	This read-only field displays the number of this output pin on the block icon, counting from the top.
Output Label	Enter a label name in this field. If Show Labels is enabled in the Display tab, the label will be displayed at the corresponding output pin location in the block diagram. This label will also appear in DocumentIt documentation.
Output Name	To increase traceability in generated code, AutoCode users can specify a name for the output signal. This name will never appear in the editor.
Output Scope	Select Local or Global scope for the output channel. This choice impacts how the output is declared in the generated code; it has no simulation effect. See the <i>AutoCode Reference</i> manual for more information.
Output Address	A text field, no more than 32 characters in length, that can contain the memory address of a Global scoped channel. This address has no simulation effect. See the <i>AutoCode Reference</i> manual for more information.
Output DataType	This field allows you to assign a datatype for each output. For more detail on datatypes, see Section 4.5 and Chapter 15 , "Fixed-point Arithmetic".
Radix	If the Output DataType is Fixed-Point, this field may be enabled. See Chapter 15 .

- Overflow Protection** The overflow protection setting is ignored during simulation (it is forced to On) but, it significantly impacts code generation. If overflow protection is off, AutoCode C will not generate overflow protection code for the current block (assuming the output datatype is fixed point). This option is not supported for Ada.
- Output UserType** If you want to specify a User Defined Type (usertype) for this output, specify it in this field. See [Section 15.5 on page 15-40](#).

4.3.5 States

The States tab is only present in Dynamic blocks.

- State Name** Optional. If entered, this name will appear in the AutoCode generated code.
- State Comment** Optional text string.

4.3.6 Document

The Document tab is only present in blocks with outputs. Its fields are used to annotate generated code, or are extracted by DocumentIt. The # (output number), **Output Label**, and **Output Name** fields are tied to the corresponding fields in the Outputs tab; a change to these fields on either tab updates both locations.

The **Output Min**, **Output Max**, **Output Accuracy**, **Output Unit** and **Output Comment** fields are strictly for documentation; any values entered will appear in the AutoCode generated code.

4.3.7 Comment

The Comment tab is available on almost all blocks; it allows you to include a multi-line comment in text form.

The main feature of this tab is a scrolling text area. This area has two purposes:

- editing and displaying comments
- defining and displaying User Parameter values.

Edit Comments

When **Comments** is selected, the scrolling text area is used to edit and display comments. The user parameter list shown in the lower right pane is grayed out.

The default editing mode is text, so, you can simply type text straight into the text area. (To change the default text editor, see [Section 6.3](#).)

Alternatively, you can create a comment in a supported document editor. By default, the Editor combo box shows the text editors vi and xemacs, the Word binary format, and the Word rich text format (rtf). Word formats are only supported on Windows platforms. To add or remove applications from the Editor combo box, see [Section 6.4](#).

To enter a comment in an editor, select an editor from the Editor combo box, then push the **Launch** button to raise it. Enter the comment, then close the editor when you are finished. The text you created will be displayed in the comment tab. If the editor is Word, the actual binary or RTF markup is displayed.

CAUTION: The | and ~ characters are reserved. If your editor is Word, you cannot use the | or ~ characters in your comment text, not even in RTF format. If you enter these characters, they will be dropped.

Edit User Parameters

When **User Parameters** is selected, the text area is used to display or change User Parameters. User parameters are created in several ways:

- Create a new user parameter from the Editing User Parameters dialog. To view this dialog, press the **Advanced** button.
- Define a new user parameter using the SETSBDEFAULTS userparameters keyword. These parameters are shown in the Default pane.

The text area is used to display or define the value of a userparameter.

Double-click on a user parameter to display it in the editor in which it was created. You can redefine the value using the appropriate datatype. If you need to add, delete, or rename a user parameter, press the **Advanced** button.

4.3.8 Icon

The icon tab is present for all blocks. [Chapter 16](#) explains how you can define or reference an icon using this tab.

4.3.9 Display

The Display tab appears for every block.

Input Pins/Output Pins Display Mode

These fields determine how connections will be displayed for the current block. Three modes are possible: Scalar, Vector, and Bundle.

- Scalar** The default; displays each pin separately.
- Vector** Groups all consecutively labeled signals of the same type (see [Section 4.4.2 on page 4-14](#)), displaying one line per group. If labels are enabled, this option displays the root label for each group of signals.
- Bundle** Uses a single line to represent all inputs/outputs. The number of signals in each bundle is displayed near each input/output bundle.

The above behaviors also have special notation within the Connection Editor. See [Section 4.6 on page 4-28](#). Note that you can also change the display mode via pull-downs in the SuperBlock Editor tool bar.

Show Labels

If **Show Labels** is checked, any inherited signal labels will be shown on the external inputs. If the receiving block is a SuperBlock reference or a Condition block, any input label information is passed from external or internal outputs to the input signal field.

Propagate Labels

Reference blocks (BlockScript, Condition, or SuperBlock blocks) have a checkbox named **Propagate Labels**. This option determines whether output labels from blocks contained in a Reference block are propagated into the Reference block's output labels. Only contained blocks that are connected to the Reference block's external outputs can propagate labels into the Reference block.

When **Propagate Labels** is checked, the contained block's output labels are immediately propagated, overwriting all output labels in the Reference block. Any change to the contained block's output labels will appear in the Reference block's output labels. You cannot make modifications to Reference block output labels when **Propagate Labels** is checked.

When **Propagate Labels** is not checked, no propagation occurs. Turning off propagation does not delete the reference block's output labels. you can modify the reference block's output labels.

Icon Color

The Icon Color field controls the color of the icon. Color is a positive or negative integer from 1-14. 0 indicates No Color. An unsigned or positive integer fills the block or bubble with the specified color. A negative integer uses the specified color to draw the object outline and the block name (the object is not filled).

Fourteen colors can be used, as listed in [Table 1 on page 4-12](#).

TABLE 1 Integer Values and Approximate Colors

Integer	Color	Integer	Color
1	red	8	pink
2	green	9	yellowgreen
3	yellow	10	bluegreen
4	blue	11	ltblue
5	magenta	12	purple
6	cyan	13	brown
7	orange	14	gray

For UNIX, these colors are approximate, as their values are based on the values specified in your Sysbld resource file (see [Section 6.6 on page 6-8](#)).

Icon Type

The Icon Type field controls the icon appearance; you can select **Special**, **Alternate**, **User**, **Simple**, or **Custom**; note, not all blocks have different views for each option. **Special** is the default selection, and usually displays a descriptive word or picture. You can also change the icon type at the block diagram level. When you select a block, its icon type will be displayed on the SuperBlock Editor toolbar. With the block selected, press **s** repeatedly to cycle through icon types. Alternatively, use the drop-down menu on the tool bar to select a new type.

4.4 Using Block Dialogs

For most blocks, numerical data is entered via a block's **Parameters** tab; depending on the block, a scalar, vector, or matrix may be entered. For the most part, all other tabs accept strings to define labels or comments, or specify code or equations pertinent to the block.

4.4.1 Using the Matrix Editor

When a block requires data in the form of a vector or matrix, the dimension input is determined by the block type and the numbers of inputs, outputs, or states, or other block-specific considerations. As applicable, the block dialog provides a simple matrix editor. Figure 4-4 shows a dialog with a two matrix editor fields: Input Points, and Output Values. Note that both show the placeholder (...) in the input field. To invoke the matrix editor, click on the field name, or click in the field itself. The dimension of the matrix displayed below will vary according to the selected field.

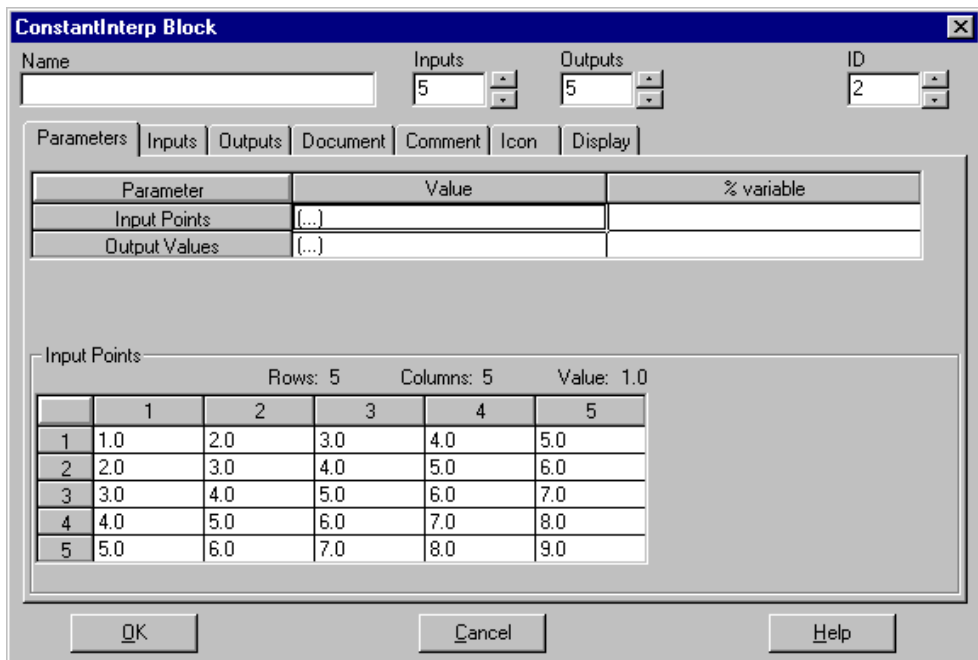


FIGURE 4-4 Matrix Editor Field in Dialog

Entering a Matrix

Data can be entered in the Parameter field or directly into the matrix editor. Matrices are specified in the same manner as Xmath data: they are enclosed in square brackets, with commas separating column elements, and semicolons separating rows, e.g., [11,12;21,22].

To enter data, specify a matrix in the parameter field, or, type the values into the matrix itself. When using the parameter field, note that you erase or overwrite the (...); if the matrix input is accepted, the placeholder will reappear; if there are prob-

lems with the input, your expression will remain in the field, giving you an opportunity to edit it. In addition to vector/matrix notation, you can use Xmath expressions to supply matrix input.

For example, given a ConstantInterp block with 5 inputs and outputs, as shown in [Figure 4-4 on page 4-13](#), the following are acceptable inputs that can be typed in either the parameter field or the 1,1 cell in the matrix editor (note, this block expects the Input Points to be increasing).

matrix using vector notation: `[1:5;6:10;11:15;16:20;21:25]`
 transposed matrix: `[1:5;6:10;11:15;16:20;21:25]'`
 expression that results in a legal matrix: `sort(rand(5,5),{incre})'`

- The matrix editor does not allow illegal matrices to be entered. In the case above, a decreasing matrix would be refused by the ConstantInterp block.
- Sometimes the input meets the block criteria, but the *dimension* of the input is incompatible. Depending on the block, the dialog may attempt to use the input by changing the inputs, outputs, or states to match the input matrix. It may also attempt to resolve the incongruity by cropping the matrix to fit the default or current matrix dimensions.

Editing a Matrix

After a matrix has been created, you may alter it from within the matrix editor by simply typing new values or Xmath expressions in either the parameter field or in matrix cells. Given a 5x5 matrix, you can do the following:

Alter a row: In 3,1, type: `1:5`
 Alter a column: In 1,5 type: `[95:99]'`
 Change the entire matrix: In 1,1 of the Output Values matrix, type:
`kron(1:5,[1:5])'`

4.4.2 Specifying Labels or Names

As shown [Figure 4-7 on page 4-29](#), block dialogs allow you to enter optional output labels for each block output. If output labels are inherited from a previous block, they can be displayed in the block diagram. To do this, go to the receiving block's Display tab and push the Show Labels button. Alternatively, select the block and

press l (lower-case L), or select the block and click the Labels On/Off icon on the tool bar.

The output labels are propagated to any blocks that receive the signal. The output label on the sending block becomes an input signal label on the receiving block (note this distinction: a *label* is defined at the source of a signal, and may be changed locally, but a *signal* is assumed to originate elsewhere, and can only be changed at the origin).

External inputs to a SuperBlock represent the first place where a signal is visible, and may be labeled in the SuperBlock. To change an input signal label, in an elementary block, you must change the output label in the block that generates the signal.

SuperBlock External Input Labels

By default SuperBlocks have the Input Naming field set to Inherit Higher-Level Names; this disables the Inputs tab **Input Label** field, prohibiting local changes.

You can replace the **Input Label** values with labels that are local to this SuperBlock and (optionally) the hierarchy below it as follows:

1. Raise the SuperBlock properties dialog; in the Attributes tab **Input Naming** field, select Enter Local Label Names.
2. Go to the Inputs tab and specify the local input labels.
3. To propagate the local signals down the hierarchy, open each SuperBlock. Go to the Display tab and select Show Labels and Propagate Label.

The BlockScript block and the Condition block can also propagate their names to blocks that receive their signals.

Creating Sequential Names for Vectors or Matrices

Although labels or names can be entered separately into each enabled field in the Input or Output tab, SystemBuild vectoring provides a way to automatically generate unique labels for each element in a vector. You can also create matrix labels by assigning a unique name for each element in a matrix. For AutoCode users, labels or names determine the structure of the output code; see the *AutoCode Reference*.

Vectoring Signal Labels or Names

Vectoring can be used to generate unique labels or names for vectors. The syntax is:

```
string(s:f)
```

`String` is the constant part of the name (must be entered first), `s` is the starting number, and `f` is the finishing number for the vector. The parentheses are required. For example, `lab(1:8)` will produce `lab1`, `lab2`, ... `lab8`.

- Descending sequences are not supported.
- If a vector longer than the list is specified, a vector of labels will be assigned, up to the last (highest numbered) label field; that is, the numbering does not wrap around to the top of the list. See [Example 4-1](#).

EXAMPLE 4-1: Vectoring Labels

In the Catalog Browser, select `File`→`New`→`SuperBlock`. The `SuperBlock Properties` dialog appears.

1. Name the `SuperBlock` `ex2` and select `Discrete`.
2. Specify 9 inputs and 5 outputs.
3. Because this is a top-level `SuperBlock`, it has no parent to inherit labels from. In the `Input Naming` field, select `Enter Local Label Names`.
4. Select the `Inputs` tab. Click into the first row of the `Input Label` field and type `ex2_a(1:3)` then press return. Click into the fourth row of the `Input Label` column and type `ex_2b(1:3)`. In the seventh row, type `ex_c(1:3)`.

Your dialog should now resemble one of the dialogs in [Figure 4-2 on page 4-4](#).

Signal Labels or Names for Matrices

The following matrix naming syntax will automatically create one label for each element in a matrix:

```
string(sr:fr,sc:fc)
```

`string` is the constant part of the name (it must be entered first), `sr` and `sc` are the starting row and column number, and `fr` and `fc` are the finishing row and column for the matrix. The parenthesis are required, and the entries produced will be emitted in **row-major** order. The total number of entries produced automatically will be

$(fr-sr+1)*(fc-sc+1)$. For example, `foo(1:2,1:2)` produces the entries `foo(1,1)`, `foo(1,2)`, `foo(2,1)`, and `foo(2,2)`.

The same result can be produced with the following alternate syntax:

```
string[ROWSxCOLS]
```

In the above the syntax string is the root of the name and it must appear first. ROWS and COLS are integers giving the number of rows and columns in the matrix. x indicates dimension, so `foo[2x3]` would create a 2x3 matrix whose entries are `foo(1,1)`, `foo(1,2)`, `foo(1,3)`, `foo(2,1)`, and `foo(2,2)`, and `foo(2,3)`. In this case the matrix dimension [2x3] will be displayed in the editor (assuming labels are on). The individual element coordinates can be viewed in the block dialog, or from the connection editor.

Matrix Labelling Must Start at 1

Either syntax is only reliable when $sr = sc = 1$ (generate starting at the first row and column positions). If you start at a location other than (1,1), the automatically generated labels will be “mangled”. You will recognize a mangled label by the presence of underscores. For example, the mangled version of `foo(3,1)` would be `foo_3_1`. These labels will not be recognized as matrix elements. One exception exists; if the dimension of the matrix was properly specified at an earlier time, the matrix structure will be recognized; the labels will still be mangled, however.

For example, suppose you have a block with eight outputs. If you delete all the output labels, go to the fifth output label box and type `foo(3:4,1:2)`, the four output labels will be mangled—they will not represent matrix elements. (You will get `foo_3_1`, `foo_3_2`, `foo_4_1`, `foo_4_2`.) However, if you delete all the output labels (again), go to the first output label box and type `foo(1:2,1:2)`, then when you go to the fifth output label box and type `foo(3:4,1:2)` you will get the expected (true) matrix element entries.

- Descending sequences ($sr > fr$ or $sc > fc$) are not supported.
- If a matrix with more elements than there are label fields (boxes) remaining is specified, matrix element labels will be assigned, up to the last (highest numbered) label field; i.e., the numbering does not wrap around to the top of the list.

Example

1. In the Catalog browser, select File→New→SuperBlock.
2. In the SuperBlock Properties dialog, name the SuperBlock `matrixExample` and make the type `Discrete`.

3. Specify 12 inputs and 5 outputs.
4. Because this is a top-level SuperBlock, it has no parent from which to inherit labels. In the Input Naming field, select Enter Local Label Names.
5. Select the Inputs tab. Click into the first row of the Input Label field and type a(1:3,1:2) then press return. Click into the seventh row and type b(1:5,1:1).

The resulting input tab is shown in [Figure 4-5](#).

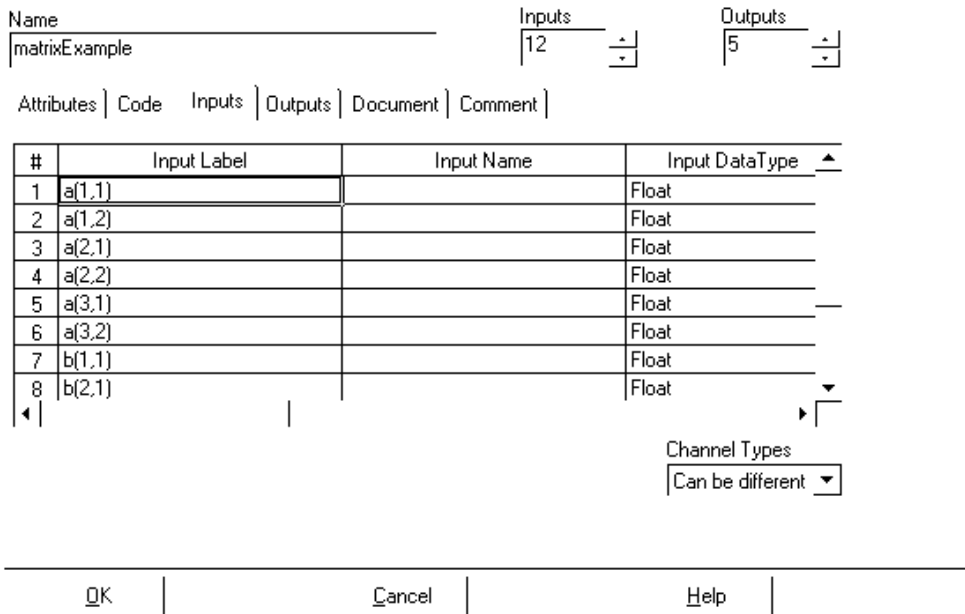


FIGURE 4-5

Shortcuts for Editing Labels or Names

This section contains shortcuts for editing existing labels. Wherever fields are writable, you can change existing labels or names individually or use the vectoring matrix syntax to overwrite multiple existing labels. The following table summarizes label editing syntaxes.

NOTE: Erase the current contents of the field before entering these commands:

string(s:f)	Starting from the cursor location, create unique names; existing names will be overwritten.
string(s:f)&	Insert new labels starting at this location, and displace ensuing labels; some labels may drop off the bottom of the list.
string(sr:fr,sc:fc)	Starting from the cursor location, create unique names; existing names will be overwritten. If results are not part of a complete matrix, "string" will be mangled and array notation not used.
string(sr:fr,sc:fc)&	Insert new labels starting at this location, and displace ensuing labels; some labels may drop off the bottom of the list. If results are not part of a complete matrix, "string" will be mangled and array notation will not be used.
(1:n)	Starting at the cursor location, erase consecutive labels, where n is the number of fields you want erased.
(1:n)&	Starting at the cursor location, insert n empty fields; ensuing labels are displaced, but preserved.
(s:f)	With your cursor in cell s, erase all fields through cell f.
(s:f)&	With your cursor in cell s, erase all fields through cell f and displace all ensuing labels.
&d	Put the cursor at the end of the line or delete the empty field before issuing this command. It deletes the current label and advances all following labels by one to fill the gap.
&dn	n is a number indicating how many consecutive labels to remove. For example, &d5 will delete five labels including the current one. The remaining labels will be advanced to fill up the space.

4.5 DataTypes

The default output and parameter datatype for SuperBlocks and functional blocks is Float. In SystemBuild, you can select the datatype for external inputs and block outputs from the Inputs and Outputs tabs, respectively.

The type associated with an input is inherited from the signal attached to the block input pin and can only be modified at the source. Input signals originate from either external SuperBlock inputs or outputs of other blocks.

Types associated with block parameters, on the other hand, are derived from the input/output datatypes and therefore cannot be modified. The rules that describe such type derivations are listed in [Table 15-1 on page 15-23](#). The radix position of a gain block that is used in a fixed-point context is an exception, because you can set this value.

For a detailed description of the fixed-point arithmetic feature, see [Chapter 15](#).

The SystemBuild datatype feature provides allows you to:

1. Create models that may require a mix of different datatypes. The user specifies datatypes in the editor while defining individual blocks and SuperBlocks.
2. Perform an automatic block by block check of the compatibility of output datatypes with input datatypes in a model. To do this use the `typecheck` keyword in the `analyze`, `sim`, `simout`, `creatertf`, `autocode`, or `documentit` commands.
3. Simulate models using the appropriate arithmetic behavior when mixed datatypes are present. The keyword `fixpt` is provided as an option for the `sim` command for this specific purpose.
4. Automatically generate code with proper type declarations for the model.

Datatypes can be monitored and modeled in simulation; the `typecheck` and `fixpt` `sim` keywords are provided for this purpose. The `typecheck` keyword performs a consistency check between input and output datatypes during the simulation analysis phase. The `fixpt` keyword enables fixed-point arithmetic, which supports mixed datatypes. By combining 8, 16, and 32-bit types with signed and unsigned properties, more than 300 datatypes can be created. The powerful `fixpt` keyword propagates and performs checking on all datatypes encountered in the model. [Table 4-1](#) summarizes the simulation behavior when different combinations of `typecheck` and `fixpt` are used. See [Section 7.3 on page 7-8](#) and the `sim` online

help for more about these keywords and how to use them. Note that both are off by default, and the default simulation behavior is to treat all datatypes as float.

TABLE 4-2 typecheck and fixpt in sim

sim options		propagated if present?		
typecheck	fixpt	float	integer	fixpt
true	true	yes	yes	yes
true	false	yes	yes	no
false	false	yes	no	no
false	true	yes	yes	yes

4.5.1 Traditional Datatypes

SystemBuild and AutoCode provide the user with a rich collection of built-in datatypes for modeling, simulation, and code generation purposes. The available set of datatypes is composed of the three traditional types, Float, Integer, and Logical, as well as fixed-point datatypes. The set of fixed-point datatypes includes more than 300 distinct members. Each fixed-point type is uniquely specified as a signed/unsigned type with two additional attributes: wordlength, and radix position.

Following the common microprocessor architectures, a fixed-point datatype may have a wordlength of 8, 16, or 32 bits. The radix position is restricted to a value between -16 and 48).

The rest of this chapter deals strictly with the non-fixed-point method of datatype checking. Depending on what is specified in the SystemBuild model, AutoCode will declare variables with one of the following datatypes:

Floating point — real-valued data is the default type for inputs and outputs. In Ada and C code this type is referred to as RT_FLOAT.

Integer — allows operation with whole numbers. In Ada and C code, this type is referred to as RT_INTEGER. SystemBuild model data which is declared RT_INTEGER is rounded.

Logical — two-valued data is referred to as BOOLEAN or RT_BOOLEAN in Ada and C.

Attempting to produce efficient code, AutoCode takes advantage of the datatype characteristics whenever possible. More appropriate algorithms may be generated according to the datatypes specified.

To generate the desired declarations, the SystemBuild Editor sets datatypes for four classes of data: SuperBlock external inputs, block outputs, block states, and block parameters. The block parameters and states are specified by internal datatype rules and require no intervention by the user. The datatypes for the SuperBlock external inputs and primitive block outputs are set explicitly in the SystemBuild editor via the Inputs tab view in the SuperBlock Attributes dialog, and the Outputs tab view in the primitive block dialog.

For each SuperBlock, the Inputs tab view in the Attributes dialog allows you to specify the datatypes and input pin label names for the external inputs, and the Outputs tab view is a read-only display, echoing the output labels on the primitive blocks that define the external outputs from the SuperBlock.

In a symmetrical manner, for each primitive block you can use the Output tab view to specify the datatypes and output pin labels for each block output. Specifically, the primitive block output pin that is connected to an external output pin furnishes the output label that appears in the Output tab view of the SuperBlock Attributes dialog. (The SystemBuild Editor does not allow multiple primitive block outputs to be connected to an external output pin.) Also, the Input tab view furnishes a read-only display, echoing the input labels on the input pins, as specified in the Inputs tab view of the SuperBlock Attributes dialog (if the input pin is connected to an external input), or the output of the previous primitive block (if the input pin is driven from another primitive block).

The block parameter or state datatype rules usually depend on the datatypes of the inputs or outputs of the block. Thus, these datatypes may be affected by what you set in the SuperBlock Input and Block Output dialogs. For example, the rule for the quantization block requires that the datatype of the Resolution parameter be the same as the output. So if in the Output dialog, Integer was specified for the output, then the parameter Resolution must also be an integer. This rule would also apply to any parameter variables (%Variables) that are used for Resolution.

To check for consistency of the three traditional datatypes among the four classes of data, you must set the simulator `typecheck` keyword option in simulation or Auto-Code. Before the RTF is generated, type checking is performed for the entire model. Any inconsistencies detected in the model will produce error messages pointing to the block or blocks that contain the conflicting datatypes.

NOTE: Full datatype checking is also performed if the `fixpt` (fixed-point arithmetic) keyword is set.

All datatype conflicts must be resolved before you can generate code. To aid you in correctly matching the various datatypes, the connection editor shows datatypes for all inputs and outputs. There is also a complete list of the floating, integer, and bool-

ean datatype rules for inputs, outputs and states for each primitive block in [Table 4-3 on page 4-25](#), and a list of blocks for which fixed-point arithmetic is supported in [Table 15-1 on page 15-23](#), along with datatype rules for these blocks. For parameter rules for each block you may refer to the online help.

The TypeConversion block is provided to help with some of your datatype mismatches. It accepts a vector of a given type and converts it to an identically-dimensioned vector of the type you specify. So if you want a block input to be integer, but the block feeding into it allows only floating point outputs, you can insert a TypeConversion block that converts floating inputs to integer outputs. For a full explanation see the online help for the TypeConversion block.

For operations that accept fixed-point inputs and perform logic or arithmetic on them, you may not need to use TypeConversion blocks, because if you specify a fixed-point datatype on an external input, the data on that input will be presented in the specified datatype.

4.5.2 Datatype and Typecheck Example

This example shows how the `typecheck` keyword affects simulation results.

1. Go to the Xmath command window and type:

```
copyfile "$SYSBLD/examples/integer_sim/intsim.cat"
```

The catalog has been copied to your current working directory.

2. Load the catalog.
3. In the Xmath command area, type:

```
[,y]=sim("IntSim1", [0:3]', {typecheck, !simclock})?
```

The sim results show that the integer and float datatypes have been preserved.

4. Reissue the `sim` command without typechecking:

```
[,y]=sim("IntSim1", [0:3]', {!typecheck, !simclock})?
```

Integer simulation does not take place; only floats are returned.

4.5.3 Traditional Datatyping Example

This example demonstrates how to resolve an input/output datatype mismatch.

Consider the model, `DataTypeGain`, shown in [Figure 4-6 on page 4-24](#).

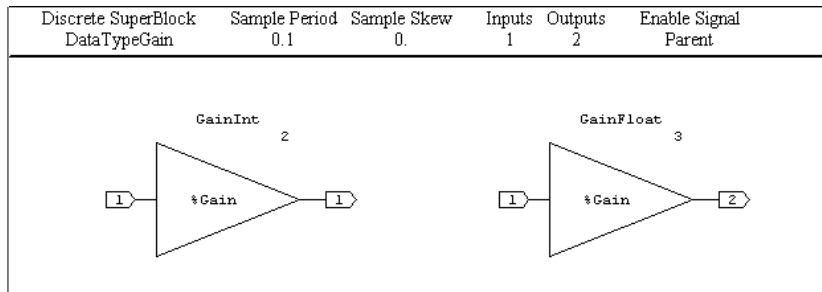


FIGURE 4-6 Datatype Mismatch Model

Check the model for errors using the `analyze` function:

```
analyze("DataTypeGain", {typecheck})
```

The keyword `{typecheck}` instructs `analyze` to include datatype consistency checking as part of the analysis. As a result of the datatype check, the following error is generated:

```
Input Type Mismatch: {DataTypeGain.GainFloat.3}
Input 1 is connected to INTEGER.
Rule: Input Type Must Match Output Type.
Expecting FLOAT.
Problems Loading model from catalog. Exiting.
```

Let's examine the model. The only difference between the gain blocks is that the output type is set to Integer for `GainInt`, and it is set to Float for `GainFloat`. Both gain blocks accept the same input signal from the parent SuperBlock `DataTypeGain`; the type is Integer.

As indicated in the error message, and also [Table 4-3 on page 4-27](#), for Gain blocks, the input datatype must be the same as the output datatype. The block `GainFloat` breaks this rule because its output datatype is float, which does not match the input datatype (inherited from the external input) which is Integer.

To correct the problem, go to the `GainFloat` Output tab and change the Output `DataType` to Integer. Run the `analyze` command again, and the analysis completes with no errors.

4.5.4 Datotyping Rules

The [Table 4-3 on page 4-25](#) lists the blocks and their datatype rules. Blocks which share common rules are grouped together. If not otherwise stated, all input and output channels must be the same datatype. Comments for each group of blocks contain specific exceptions to the general rules; when a superscript appears you can find the corresponding comment on [page 4-28](#).

TABLE 4-3 Legal Data Types for Each Block (Except Fixed-point Datatypes)

Block Type	Legal Inputs	Legal Outputs	Legal States
Summer, ElementProduct, DotProduct, CrossProduct, ElementDivision, AbsoluteValue	Same as output	Integer or Float	NA
TypeConversion	Any	Any	NA
TimeDelay	Same as output	Integer OK if discrete or procedure	Same as outputs
ShiftRegister LogicalOperator	Any input > 0 is True, else False	True = 1, False = 0.	Same as outputs
RelationalOperator	Integer or float	True = 1, False = 0.	NA
DataPathSwitch	First channel, any input > 0 is True, else False; Other channels, same as output	Any	NA
Stop	Any input > 0 is True; else False	NA	NA
STD	Any ¹	True = 1, False = 0.	NA

TABLE 4-3 Legal Data Types for Each Block (Except Fixed-point Datatypes) (Continued)

Block Type	Legal Inputs	Legal Outputs	Legal States
SpringMassDamper StateSpace NumDen Pole Zero ComplexPoleZero Integrator Hysteresis LimitedIntegrator PIDController UserCode Fuzzy Logic	Must be float	Must be float	Must be float
SquareRoot Logarithm Exponential SignedSquare Root Sin Cosin Atan2 SinAtan2 Cosin Asin CosAsin) Acos Cartesian2Polar Polar2Cartesian Cartesian2Spherical Spherical2Cartesian AxisInverse AxisRotation CubicSplineInterp BiLinearInterp BiCubicInterp MultiLinearInterp.	Must be float	Must be float	NA
Quantization	Same as output	Integer or float	NA
AlgebraicExpression	Integer or float ³	Integer or float	NA

TABLE 4-3 Legal Data Types for Each Block (Except Fixed-point Datatypes) (Continued)

Block Type	Legal Inputs	Legal Outputs	Legal States
Waveform PulseTrain SquareWave Step	NA	Integer or float ²	NA
SinWave UniformRandomGenerator NormalRandomGenerator	NA	Must be float	NA
LogicalExpression	Integer or float ¹	True = 1, False = 0.	NA
UPowerConstant ConstantPowerU	Same as output	Must be float	NA
BlockScript	Defined in the BlockScript		
GainScheduler	First channel can be Integer or Float. Other channels same as output.	Integer or float	NA
Polynomial Gain Encoder Decoder ConstantInterp LinearInterp Breakpoints Deadband Saturation LimitedIntegrator BiLinear Interp Table Preload	Same as output	Integer or float	NA
WriteVariable	Must be float ³	NA	NA
ReadVariable	NA	Must be float ³	NA
Constant	N/A	Integer, Float, or Logical	N/A

TABLE 4-3 Legal Data Types for Each Block (Except Fixed-point Datatypes) (Continued)

Block Type	Legal Inputs	Legal Outputs	Legal States
ScalarGain, MatrixTranspose, MatrixMultiply, RightMultiply, LeftMultiply	Same as output	Integer or Float	N/A
MatrixInverse, MatLeftDivide, MatRightDivide	Float only	Float only	N/A

1. Each input and output channel can be a different datatype, but channel usage must be consistent across all transitions. For example, a logical input channel cannot also be used in a numeric expression, and a numeric input channel cannot also be used in a logical expression.
2. The parameters related to the TIME variable are always floating point, while variables related to the OUTPUT variable match the output data type in the generated code.
3. Variable Rules: The variables written to may be Float or Integer. If Bit addressing is used, the variable must be Integer.

4.6 Connecting Blocks

Connections are data and control flows that direct input signals through a model. They appear as orthogonal lines on the screen. Within a block diagram, each (internal) connection routes the (output) signal from a block so that it is the input signal to a specific pin in the next block in the sequence. Signal connections go from left (inputs) to right (outputs).

By default, the connections are displayed in scalar mode, meaning each pin is drawn separately. The number of pins and any related labels are displayed in the editor.

In [Figure 4-7](#), the labels make it easy to follow the signals inherited from the SuperBlock ed2.

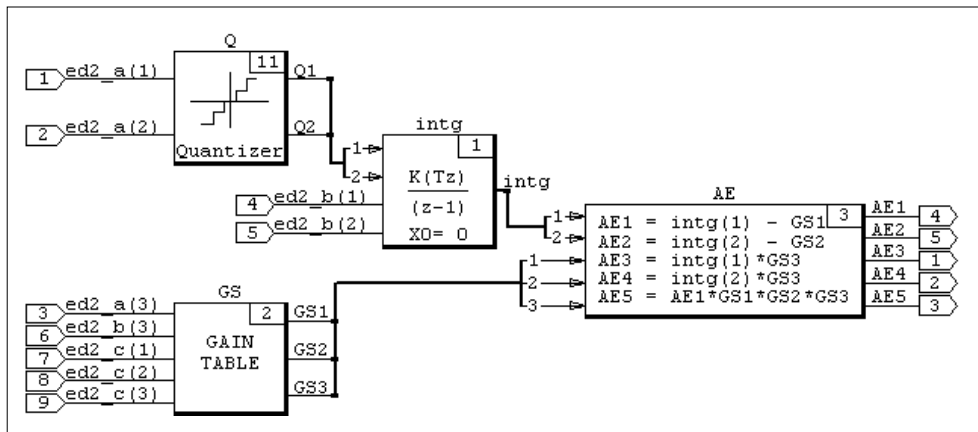


FIGURE 4-7 Scalar Connections

4.6.1 Connection Rules

With very few exceptions, the following rules govern SystemBuild block connections:

- In general, a single input accepts a single output from one other block.
- An output can be connected to the inputs of one or more other blocks.
- An output must not be directly connected as an input into the same block. If the signal first passes through other blocks, the connection is allowed. See [Figure 3-2 on page 3-3](#).
- Inputs and outputs are not required to be connected during an editing session. When the block diagram is analyzed for simulation, or when code is generated, any unconnected input pins will produce a warning; unconnected inputs are assigned to 0.
- Individual blocks may generate external outputs.
- Top-level SuperBlocks are required to have at least one output. Unconnected outputs from a top-level SuperBlock are set to zero. SuperBlock references, including Procedures, may have zero outputs.
- It is acceptable to connect an element of a vector output to a scalar input, or a scalar output to an element of a vector input.

4.6.2 Creating Connections


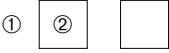
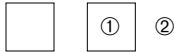
You can connect blocks using selections from the SuperBlock Editor's Connect menu or using mouse shortcuts.

Creating a Simple Connection

A simple connection connects the first available **From** block output signal to the first available **To** block input pin; it then connects the next available output signal to the next available pin until all pins are connected.

To perform a simple connection, middle-click (3-button mouse) or Control-right-click (2-button mouse) as shown in [Table 4-4 on page 4-30](#).

TABLE 4-4 Simple Connections

Connection	From ①	To ②	Click
block to block	source	destination	
external inputs to block	open space to left of destination	destination	
block to external outputs	source	open space to right of source	

Using the Connect Menu

The Connect menu is fully described in the online help for the editor window; in the editor tool bar, press the ? icon to raise the editor help. The connection process is selection-enabled, that is, at least one block must be selected to enable Connect menu items. The following actions will raise the connection editor to perform the connection.

Inputs Connect an external input to the selected block. This menu item is only enabled when one block is selected.

Mouse: Middle-click (3-button mouse) or Control-right-click (2-button mouse) in an open area, then in the target block.



Select one block, then click the external input icon.

Blocks Create a connection between two selected blocks. This menu item is enabled only when two blocks are selected.

Mouse: Middle-click (3-button mouse) or Control-right-click (2-button mouse) in From object, then in To object.



Select two blocks then click the internal connection icon.

Outputs Connect the selected block to an external output. This menu item is only available when one block is selected.

Mouse: Middle-click (3-button mouse) or Control-right-click (2-button mouse) on object, then in an open area.



Select a single block then click the external output connection icon.

Manual Routing Enable manual routing of connections. Select a connection using the middle mouse button (3-button mouse) or control-right-click (2-button mouse) and change the routing by dragging the marker to a new location.

Mouse: Middle-click (3-button mouse) or Control-right-click (2-button mouse) in open area, holding down button until transition markers appear.

Automatic Routing Automatically route connections to selected block(s). The SystemBuild Editor will lay out the connections using an algorithm that picks a route that avoids crossing blocks and other connections if possible. If manual routing was previously performed on the selected block(s), selecting this option negates that effort.

Using the Connection Editor

There are four buttons at the bottom of the connection editor: **Cancel**, **Add**, **Del** and **Done**. Just above the connection buttons there is a **From**, **To** field. If external inputs or outputs are involved, the dimension of the External Input/Output vector will be displayed at the top of the form. This field is editable.

Creating Connections

- Only one connection editor is allowed at any given times, regardless of the number of editors open.
- The connection editor interface is unique in SystemBuild in that the action (**Add** or **Del**) must be selected *before* the operand (the input signals) are chosen.

- To add the maximum number of one-to-one connections in one operation, double-click on **Add** and the connection editor will make the simplest set of connections that it can without deleting existing connections. For example, if pins 1 and 2 are available on both sides of the menu, they are connected.
- To make a single connection (with the **Add** button enabled) select a pin number from the source block, then select a pin number in the destination block. Although it is natural to go from left to right it is not necessary; you can select a pin from either block as long as the next selection is from the opposite block.
- You can also create connections using the **From, To field**. This field assumes the input is from left to right (source to destination). You specify a single connection as a pair of integers separated by a comma. For example, 2,7 will connect the 2nd output of the source block to the 7th input of the destination block. To specify multiple connections, specify two vectors separated by a semicolon, where the first vector represents a range of pins on the source block, and the second represents pins on the destination block. For example, 1:5;11:15.
- To make multiple connections, drag-select (lasso) multiple consecutive pins from the source block then lasso an equal number of consecutive pins on the destination block.

Deleting Connections

- Click once on the **Del** button to enable deletion. Click any pin on the source (left-side) or destination (right-side), and its connections will be erased.
- Double-click on **Del** to undo all the changes in the last connection editor session.

Altering the Number of External Inputs or Outputs

- To alter the number of External inputs or outputs from within the connection editor, change the number in the display at the top editor and press **Return**.
- If you add to the number of External Inputs, the additional inputs will be added to the end.
- If you are adding to the number of Outputs, you can add them to the end by typing the new number of outputs and pressing **Return**. You can insert outputs into the existing output list as follows:
 - a. Type the new number of outputs (but *do not* press **Return**).
 - b. Click between any two output destination pins; the additional pins will be inserted at that location, and any previous information will be displaced (but not lost).

Displaying Connections

The way signals are displayed in the Connection Editor is influenced by the block labels or names, and the Input Pins/Output Pins settings for each blocks. By default, vectored labels will be compressed. When you have a large number of pins shown in scalar mode, a scroll box will appear above the **Cancel** button. By default, Channel 1 is always at the top. You can use the down arrow to move the view channels further down the list, and return towards the top with the up arrow. The scroll bar allows you to do the same on a larger scale; drag the box left to go to the top of the list, or drag right to display the bottom of the list. Note, when you move connections out of alignment they will be visually truncated; the connection is not affected, however.

Exiting the Connection Editor

- Single-click on the **Cancel** button to exit the Connection Editor and discard all the changes in the last session. Double-click on **Cancel** to remove all changes but leave the Connection Editor on the screen.
- Click **Done** button to accept the changes that were made in this session, transfer them to the picture on the screen, and return to the SystemBuild editor.

4.6.3 Automatic and Manual Connection Routing

By default SystemBuild uses automatic connection routing. An algorithm is used to determine a path that takes the fewest turns, does not overwrite blocks or other connections, etcetera. The results are good for most diagrams but you may need to use manual routing to adjust the routing for complex diagrams.

When Manual Routing is enabled, square “handles” will appear on the connection lines. Use the middle-mouse button to drag a handle to a new location.

There are two ways to enable Manual Routing:

- Select Connect→Manual Routing.
- Middle-click on a hidden handle on a connection. Clicking in the middle of a horizontal connection is often successful.

While changing the routing you may have difficulty “lining up” connections. In limited circumstances, it may be helpful to go to the Options menu and disable Snap. Remember to enable snap after you have made your adjustments.

Manual routing does not fix all aesthetic problems; complex diagrams may also require a combination of relocating and resizing blocks and adjusting label display. These and other issues are discussed in [Section 4.7](#).

4.7 Modifying Block Diagram Appearance

In addition to block level settings, the Editor provides many ways to improve the default appearance of your diagrams.

- SystemBuild Editor shortcuts are the most expedient way to change your diagram. For a summary of all shortcuts, go to the Xmath command area and type **help shortcuts**.
- When you are editing a large model, you can always use the scroll bars to move the view. However, the View menu includes other helpful options. Fit compresses the model so that all blocks appear within the editor viewing region; Zoom changes the image size without changing the dimensions of block diagram elements; Normal restores the default Zoom (%100). These options are also available on the Edit ToolBar, which appears below the menu bar. For a full description of each tool, select Help→Topics.

Panning is a useful option for viewing large models that extend beyond the screen. Hold down the Right mouse button in empty space; and drag the cursor in the direction that will allow you to see the hidden portions of the model.

- The Display Tool Bar, which appears just above the work area offers easy access to block attribute settings. See the online help for a description of each icon.
- Certain Algebraic blocks have display icons that can appear with their input connections in any of several locations. These include summers, multiplications, dividers, etc. To toggle the icon display, select the block and press **s** repeatedly, or, change the type using the Icon Type combo box on the tool bar.

[Figure 4-8 on page 4-35](#) shows an example of an organized, but cluttered, diagram.

1. All elements are shown at normal size. To make the labels more readable, increase the font size to 16 points. You can do this via the combo box on the Edit tool bar, or you can place the cursor in the window and type **>** until the font is the desired size (type **<** to reduce the font size).
2. At this point, block 3 is too narrow to display the Algebraic Equations. To widen the block, place the cursor over the block and type **w** repeatedly. Alternatively, click on the corner that shows the block ID and pull to drag the block wider. Move the blocks so that the labels no longer overlap.

Turn on the manual routing markers, and use the middle mouse button to drag the connections so that they no longer overlap the labels, as indicated by the arrows in ②.

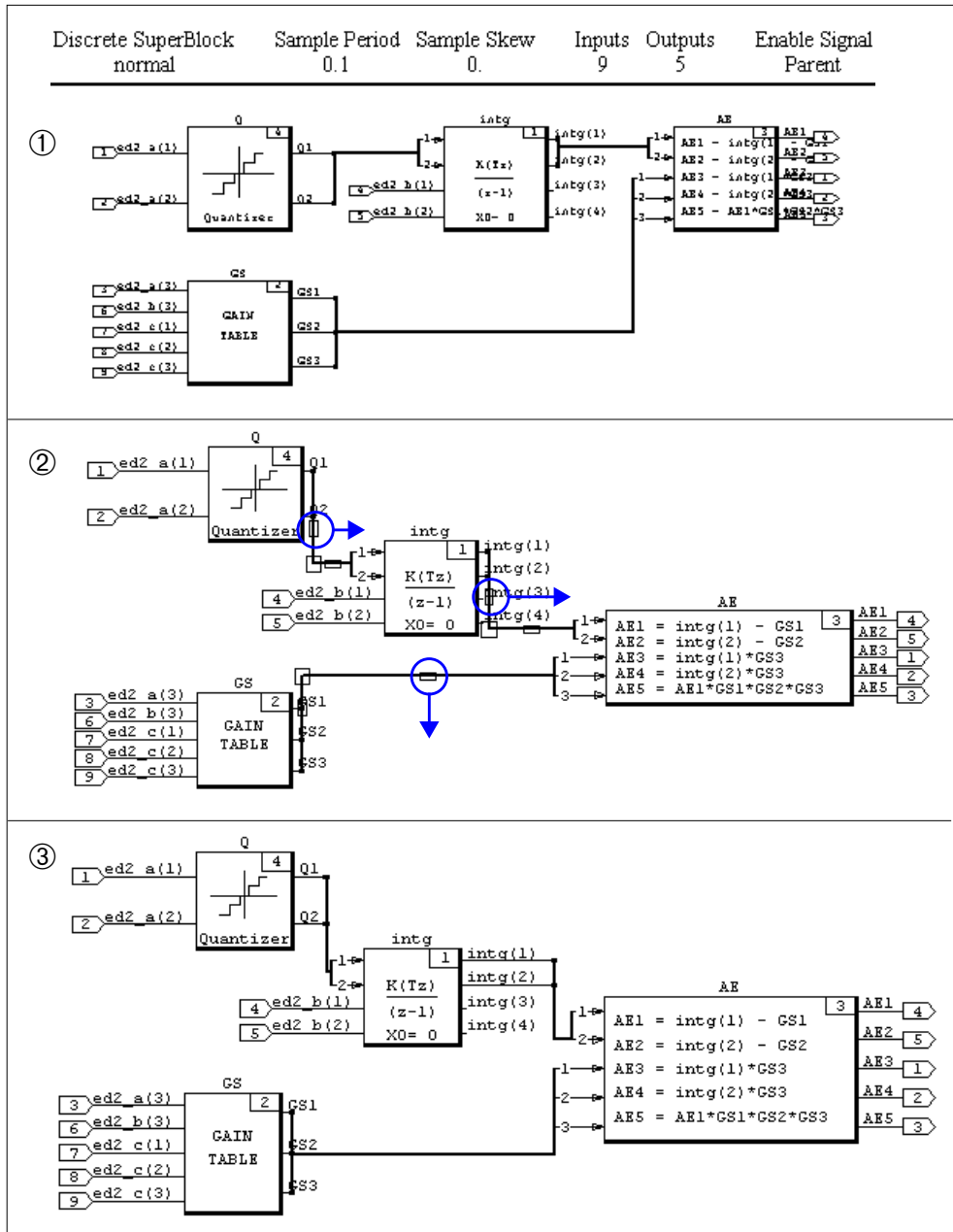


FIGURE 4-8 Modifying a Block Diagram

3. Finally, make some of the blocks taller by dragging upward on the corner by the ID number, or placing the cursor over the block and typing **T**. The diagram should now resemble ③.

4.8 Example

This example showcases the connection editor, and how block labels and settings affect the appearance of the block diagram. In the first four steps you will create the simple SuperBlock Forty, shown in [Figure 4-9](#) on page 4-36.

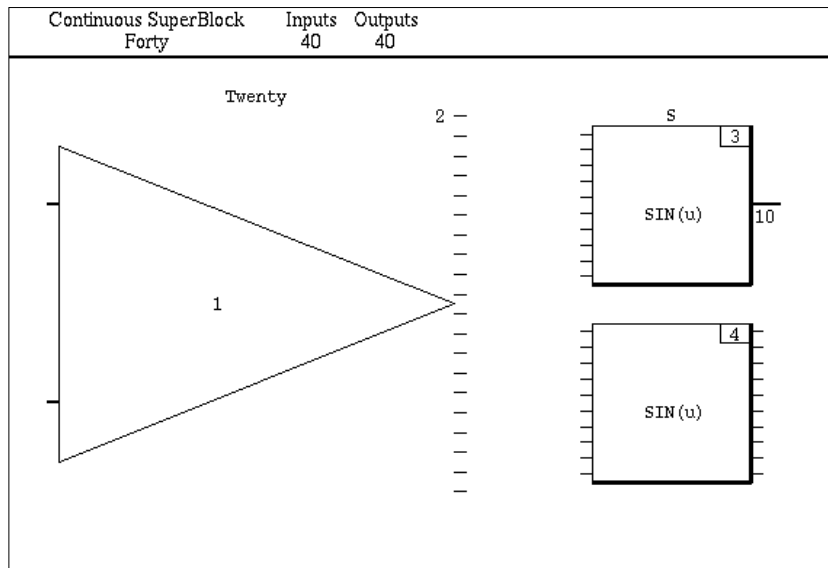


FIGURE 4-9 SuperBlock Forty Before Connections

1. Create a SuperBlock named Forty. Give it 40 inputs and 40 outputs. Set the Input Naming Field to Enter Local Label Names. Go to the Inputs tab and specify labels as follows: In cell 1, type **f1(1:10)**. Type **f2(1:10)** in cell 11, **f3(1:10)** in cell 21, and **f4(1:10)** in cell 31.
2. Create a gain block named Twenty and specify 20 inputs and 20 outputs. On the inputs tab, type **t1(1:10)** in cell 1, and **t2(1:10)** in cell 11. Go to the Outputs tab, and specify the same labels. On the display tab, enable Show Labels, and set the Input Pins field to Vector. When the finished block is displayed, note that only two inputs pins are shown. This reflects the fact that two vectors

of unique labels are used. There are 20 pins on the output, as is normal for scalar mode.

3. Create a Sin block with 10 inputs and 10 outputs. Duplicate the block (place the cursor over it and type **d**).
4. Edit the first sin block. Name it S. On the inputs tab, specify **s1(30:34)** in cell 1, and specify **s2(35:39)** in cell 6. On the display tab enable Show labels, and set the Input Pins type to Vector and the OutPut pins type to Bundle. Note that the pins are depicted as scalar because the inputs are not connected yet (see [Figure 4-9](#)). The outputs are displayed as a single thick pin; the number 10 indicates the number of pins in the bundle.
5. To connect external inputs to twenty, middle-click in white space to the left of the block, then middle-click on the block. The Connection Editor will appear.
 - a. The initial view, ①, in [Figure 4-10 on page 4-38](#), demonstrates that vectored labels are grouped together by default. To expand a vector (to scalar view), click on a filled triangle. To compress a vector, click on a hollow triangle associated with any label in the vector.
 - b. Double-click on the **Add** button to create a simple connection from the external inputs to Twenty (see ②). To expand the f1 source vector, click on the filled triangle beside f1[10]); do the same for the t1 destination. Note, a scroll bar appears above the **Cancel** button.
 - c. With the vector expanded we can create single connections or multiple connections. You can create view ③ as follows. With the **Add** button selected, lasso (drag a selection box) the first 5 pins from f1 (④), then select the last five pins of t1 (⑤). Connect f1(6:10) to t1(1:15) in the same manner.

Click **Done**.

6. Next we will connect external inputs to S. Middle-click in empty space to the left of S, then middle click on S. In the **From, To Field**, type **25:32; 1:8**. Click **Done**.
7. Select Twenty and S, then raise the connection editor. Double-click on **Add**. Note that first two available pins on each block are connected. Click **Done**.
8. Create a connection from external inputs 35:39 to the first 5 inputs of block 4. The diagram should now resemble [Figure 4-11](#).

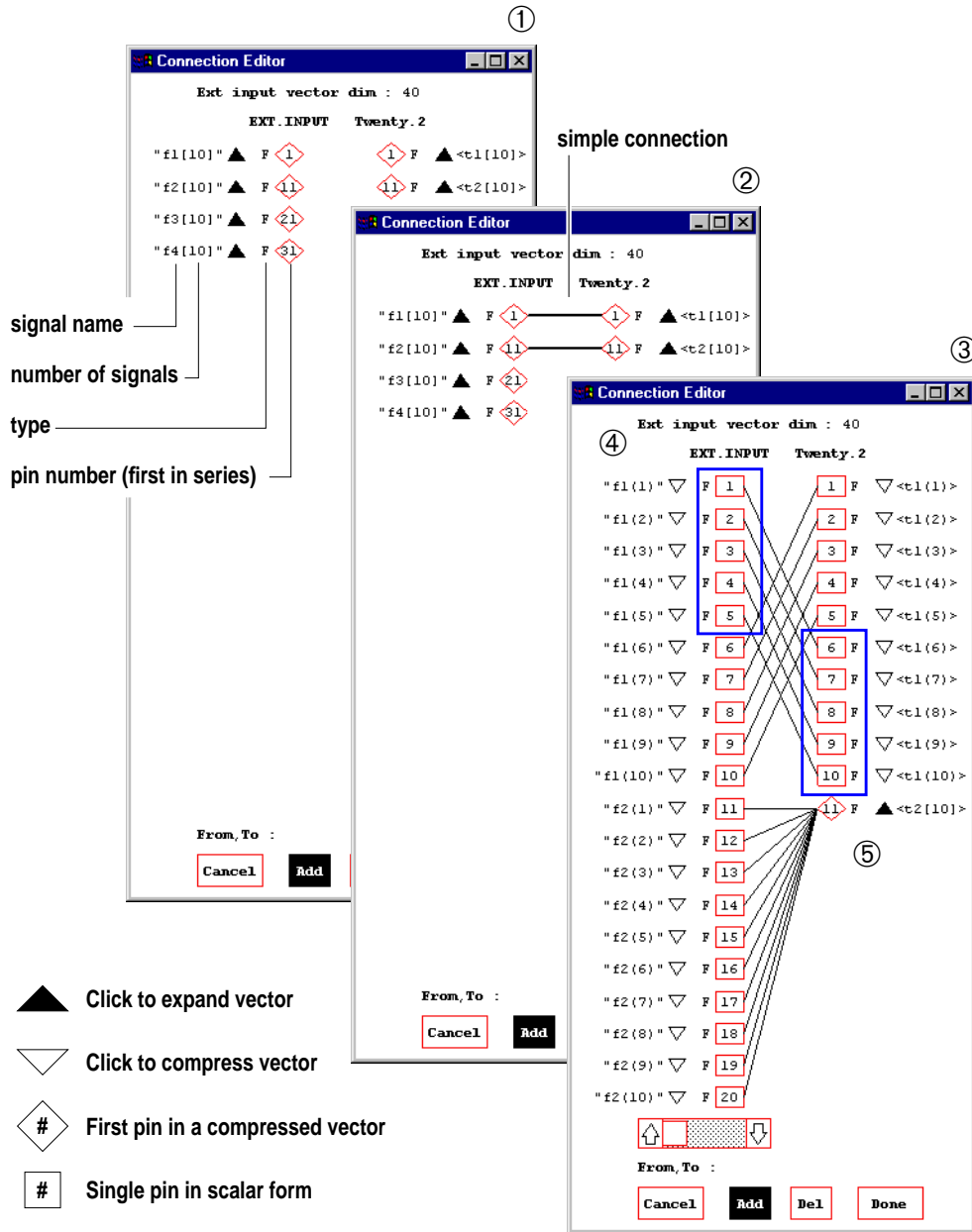


FIGURE 4-10 Connecting External Inputs to Forty

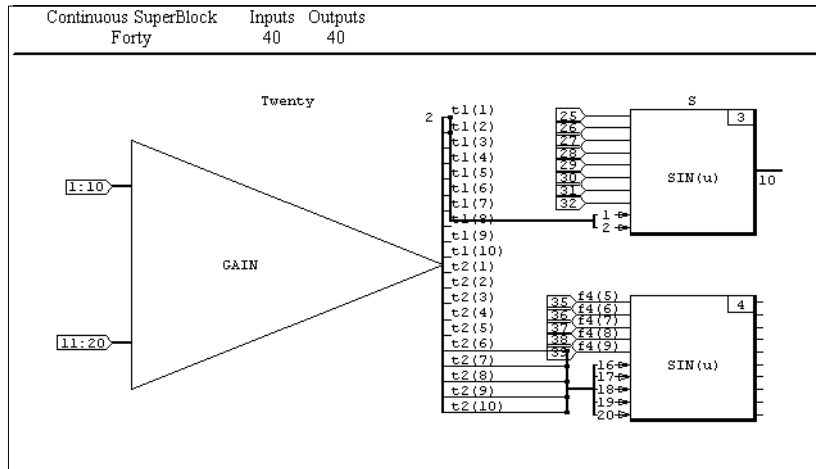


FIGURE 4-11 Diagram Before Outputs

9. Create external outputs as follows:
 - a. Create a simple connection between Twenty and the external outputs. To simply the appearance, select Twenty and select Vector from the OutPut Display Mode pulldown on the SuperBlock Editor tool bar.
 - b. Create a Connection from S to the External Outputs. Connect channels 9 and 10 to External Outputs 1 and 2. Connect 1:4 to 4:7 and 5:8 to 15:18.
 - c. Connect block 4 to external outputs. When you raise the connection editor, move the scroll bar all the way to the right. To view the final output pins. Connect from 4(1:5) to (21:25), and 4(6:10) to 36:40.
10. Your connections are complete but the diagram is disorderly. To improve the readability of the outputs, make the blocks taller (place the cursor over a block and press T). Select Connect→Manual Routing to display the connection routing marker handles. Use the middle mouse button to drag handles to new locations

that eliminate overlaps. Use Manual Routing to adjust the external outputs so that they do not overlap the inputs, as shown in [Figure 4-12](#),

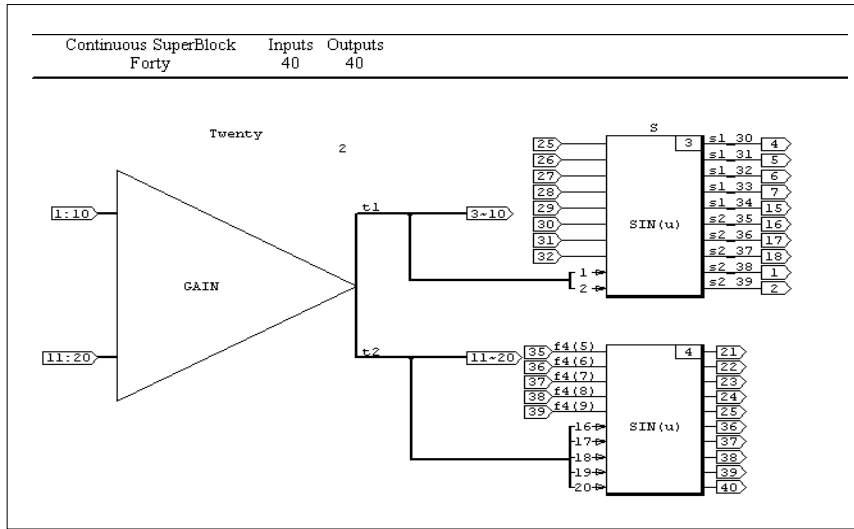


FIGURE 4-12 Diagram with Resized Blocks and Manual Routing with Vectorized Output from Twenty

Finally, change the Input Pin and Output types from Scalar to Vector to Bundle.

4.9 Special Blocks

Most of the blocks in the SystemBuild library perform a dedicated computational function. A trivial example is the Gain block, which multiplies the inputs by some fixed value and passes the result to the output channel. There are some additional blocks in SystemBuild that do not do dedicated computations, but rather control the execution behavior of other blocks and SuperBlocks in the model. These special SystemBuild blocks can conditionally execute blocks or SuperBlocks in the model, repetitively execute blocks or SuperBlocks, define the block execution order or terminate execution altogether.

4.9.1 Conditional Execution (Condition, IfThenElse Blocks)

The Condition and IfThenElse Blocks provide frameworks for conditional execution. The main difference between the two blocks is the type of blocks they control.

The Condition block controls the execution of Procedure SuperBlocks. Depending on the inputs and the mode of the Condition block, one or more Procedure Super-

Blocks listed in the its Code tab will be executed. See the online help for the Condition block for more information.

Instead of controlling the execution of entire procedure SuperBlocks, IfThenElse blocks control execution of specific blocks in a SystemBuild diagram. Each IfThenElse block has boundary area (a *container*) where blocks can be placed and connected. A logical expression, defined in the block dialog, determines the block execution order. If the logical expression evaluates true, the blocks in the IfThenElse block container are executed. See the online help for the IfThenElse block and Container Blocks.

The DataPathSwitch block might also be classified as a conditional execution block, but this would be incorrect. With this block, all inputs are executed first, and only one of the inputs is passed through to the block outputs.

4.9.2 Repetitive Execution (While, Break Blocks)

The While block has a similar structure to IfThenElse blocks. The blocks inside the While block container are executed until the inputs to a Break block are true. Each While block must contain a Break block.

4.9.3 Terminating Execution (Stop Block)

The Stop block will stop execution of a model if any of its inputs are true.

4.9.4 Execution Ordering (Sequencer Block)

The Sequencer is a simple block that has no inputs or outputs. On a SystemBuild diagram, the sequencer is shown as a double vertical line that partitions a SuperBlock Diagram into two areas. The sequencer does not do any kind of computations. Its sole purpose is to define the order that blocks are executed. This can be important if Variable blocks or Procedure SuperBlocks are used in the model.

The effect of the sequencer on block execution is simple: all basic blocks to the left of a sequencer bar are executed before all blocks to the right of the same sequencer bar. The blocks in any child SuperBlock to the left of the sequencer bar will also be executed before blocks on the right of the bar if the child SuperBlock is in the same subsystem as the SuperBlock containing the sequencer. Please see the online help for more information and examples.

5

SuperBlocks and SuperBlock Transformations

SystemBuild provides a variety of SuperBlock types to model both continuous and discrete nonlinear dynamic systems. This chapter discusses SuperBlocks types and their intended purpose and attributes.

5.1 Continuous SuperBlocks

Continuous SuperBlocks model nonlinear dynamic ordinary differential equations (ODEs) of the form:

$$\begin{aligned}x_0 &= x_{init} \\ \dot{x} &= f(x, u) \\ y &= g(x, u)\end{aligned}$$

Where u is the input vector, x is the state vector, y is the output vector, and x_{init} is the initial state provided by the user. You may choose among several ODE solvers to best approximate the solution to the continuous models. Recommendations for use of each integration algorithm are given in [Section 7.10 on page 7-18](#).

5.2 Discrete SuperBlocks

Discrete SuperBlocks model systems that sample and hold their inputs at a specified sample rate. The system can be expressed as a difference equation where k is the sample index:

$$\begin{aligned}x_0 &= x_{init} \\ x_{k+1} &= f(x_k, u_k) \\ y_k &= g(x_k, u_k)\end{aligned}$$

Attributes required to define a discrete SuperBlock are:

Sample Period — The sample rate of the discrete SuperBlock.

Sample Skew — The offset between the simulation start time and the first execution of the current SuperBlock.

Enable Signal — Determines whether a SuperBlock will be free-running or enabled. You may specify None (the default), Parent, or a pin number (an input signal).

Selecting **None** results in a free-running system. This means that the SuperBlock is always enabled and will execute periodically at its Sample Period, beginning at the specified first sample (skew) and continuing throughout the simulation.

Selecting **Parent** or specifying an input signal creates a system that will run periodically, but only when it is enabled by its parent SuperBlock or the specified input signal. A SuperBlock enabled by its parent will execute at its sample interval as long as its parent is enabled. A SuperBlock enabled by an input signal will execute at its sample interval as long as the enable signal is true.

Group ID — Allows you to assign a SuperBlock and its primitive blocks to a specific processor group. This field is enabled whenever the SuperBlock type is Discrete. By default, all systems of the same rate are grouped in the same subsystem, so this setting allows you to override the arbitrary grouping. This parameter is useful for both multiprocessing and for controlling the size of generated procedures during the AutoCode code generation process.

5.3 Triggered SuperBlocks

Triggered SuperBlocks are discrete in nature, but do not execute periodically. They are executed once each time the leading edge of the trigger signal is detected (whenever it transitions from ≤ 0 to > 0). Alternatively, a triggered SuperBlock can have Asynchronous output posting. Asynchronous Triggered SuperBlocks are executed once each time either the leading edge or the trailing edge of the trigger signal is detected.

All discrete dynamic equations for blocks nested within a Triggered SuperBlock are evaluated assuming a sample interval of 1.0 seconds. As a result, dynamic blocks that incorporate the sample rate into their equations, such as the discrete integrator, should be used with care, because Triggered SuperBlocks are not executed periodically.

Attributes required to define a triggered SuperBlock are:

Trigger Signal — You may specify either Parent or a pin number to be used as the trigger signal.

Output Posting — This topic is explained in more detail in [Section 5.7](#), Four choices are available:

- After Timing Req.
- As Soon as Finished
- At Next Trigger
- Asynchronous

Timing Requirement — When output posting is After Timing Requirement, the Timing Requirement value is the elapsed period of time between the start of execution for the triggered subsystem, and the time when the outputs of the triggered subsystem will be available to other system elements. The lower the timing requirement, the higher the priority of the subsystem.

5.4 Procedure SuperBlocks

Procedure SuperBlocks are special constructs designed to represent generated software procedures that can be called as reusable functions in the code generated by AutoCode.

Unlike types of SuperBlocks, one function will be generated for each Procedure SuperBlock, and multiple references to the same procedure will reuse the single function. Using Procedure SuperBlocks can reduce the generated code size if SuperBlocks are multiply referenced.

There are two ways to use Procedure SuperBlocks in a model: a direct reference or a reference through a Condition block. For the most part, Procedure SuperBlock behavior is consistent for both types of references, but some differences exist and these are detailed in the Condition block discussion in the online help.

There are six classes of Procedure SuperBlocks: Standard, Macro, Startup, Background, Interrupt, and Inline. Of these, only Standard and Macro Procedure SuperBlocks can be referenced from a Condition block. Standard and Macro Procedures are general purpose elements that can be used within Discrete or Triggered SuperBlocks, or any type of Procedure SuperBlock. The Startup, Background, and Interrupt Procedures are special constructs that enable you to model Real-Time asynchronous tasks in AutoCode-generated code.

All Procedure SuperBlock references are treated as individual subsystems and are executed completely as if the SuperBlock reference were an intrinsic block within the parent. This treatment facilitates the mapping of these SuperBlocks to stand-alone reusable functions. The exception to this is the Inline Procedure SuperBlock (see [Section 5.4.3](#)).

Another characteristic of Procedure SuperBlocks is that they inherit timing attributes, such as the sample interval, from the parent SuperBlock. If the same procedure is referenced in multiple SuperBlocks with different rates, each reference will inherit each different rate.

Discrete, Triggered, or Enabled SuperBlocks with identical timing attributes are typically combined into collective subsystems from which the blocks are sorted to minimize algebraic loops. Since procedures must execute completely in one pass, care must be taken to avoid algebraic loops.

Limitations

- Procedure SuperBlocks cannot be used in any continuous SuperBlock or as the top-level SuperBlock.
- Procedure SuperBlocks are the only type of SuperBlock that may be nested within another Procedure SuperBlock.
- The ReadVariable and WriteVariable blocks (see the online help) are provided for communicating information to and from procedures in a manner that is similar to accessing global data from a function. However, the read and write sequence may not occur in the order that you would assume.
- The requirement to treat procedures as stand-alone functions prohibits using DataStores in procedures.

5.4.1 Standard Procedures

The Standard Procedure SuperBlock is a generic utility that may be nested within any Discrete, Triggered, or Procedure SuperBlock. A Standard Procedure will inherit the attributes of the parent SuperBlock. These are the only Procedure SuperBlocks supported within the Condition block.

5.4.2 Macro Procedure

The Macro Procedure behaves the same as a Standard Procedure during System-Build simulations. However, AutoCode will substitute a user-supplied macro statement in place of a call to a generated procedure. As a result, you can directly call a

special I/O or utility function from the generated code and replace that equivalent behavior with SystemBuild blocks during simulation.

The procedure SuperBlock catalog item must define the call to the macro; like all other SuperBlocks, it must contain at least one block, even if only the procedure is of interest.

- The macro string is specified on the Code tab.
- The first line contains the name of the macro, terminated with a semicolon.
- All subsequent lines are arguments to the macro, one per line, each terminated with a semicolon.

EXAMPLE 5-1: Macro Procedure SuperBlock

Type the following in the Code tab:

```
Read_A0;  
Channel_1;
```

The above will result in the following generated C code:

```
Read_A0(Channel_1, in_signal, out_signal);
```

Input and output variables will be listed individually following the optional arguments.

The actual macro code must be input by the user into either an include (*.h) file or directly into the source code file.

For ADA, a simple procedure call is generated. It's up to the user to fill in the definition of the procedure.

The macro may also be redefined with pre-processor directives from within the AutoCode template file. See the *AutoCode Reference* manual for more information on importing macro code into generated code.

5.4.3 Inline Procedure

In contrast to all other classes of Procedure SuperBlocks, the Inline Procedure is not treated as an individual subsystem. The primitive blocks nested within an Inline Procedure are merged into the subsystem of the parent SuperBlock. As a result, use of Inline Procedures, as opposed to Standard procedures, will influence the block

execution order and can help eliminate algebraic loops in many cases. This treatment, however, comes at the expense of generated code size. AutoCode will not create a reusable procedure for Inline Procedures.

This treatment, however, only applies to directly referenced Procedure SuperBlocks. If an Inline Procedure is referenced through a Condition block, it will be treated like a Standard Procedure and an individual subsystem will be created for that particular reference.

5.4.4 Background Procedure

The Background Procedure represents the computations that are to be performed when the system is otherwise idle. Essentially, the Background procedure is the lowest priority task and is executed only when no other tasks require execution.

A Background SuperBlock cannot have external inputs or outputs and is required to interact with other system elements via ReadVariable and WriteVariable blocks. Hierarchies of Standard and Macro Procedures can be constructed within the Background SuperBlock. Dynamic blocks with states cannot be included in a Background SuperBlock, although state-like behavior may be achieved with Variable blocks. UserCode blocks (UCB) or BlockScript blocks in Background Procedure SuperBlocks, cannot have states.

The Background Procedure is primarily an AutoCode-specific concept and is not supported in the SystemBuild simulator.

5.4.5 Startup Procedure

The Startup SuperBlock is a means by which initialization calculations may be performed before the beginning of a simulation.

A Startup SuperBlock may not have external inputs or outputs and only interacts with other system elements through the ReadVariable and WriteVariable blocks. Variables initialized from a Startup Procedure may be used as %Variables in other SuperBlocks, or may be accessed with ReadVariable blocks. Hierarchies of Standard and Macro Procedures can be constructed within the Background SuperBlock. Dynamic blocks with states cannot be included in a Startup Procedure SuperBlock, although state-like behavior may be achieved using Variable blocks. UserCode blocks (UCB) and BlockScript blocks in Startup Procedure SuperBlocks cannot have states.

5.4.6 Interrupt Procedure

NOTE: Asynchronous Triggered SuperBlocks are expected to replace Interrupt Procedure SuperBlocks, since asynchronous triggers provide a more flexible, “simulatable” solution without most of the restrictions of Interrupt Procedure SuperBlocks. Although Interrupt Procedure SuperBlocks are still supported for backward compatibility this feature may be discontinued in the future.

The Interrupt Procedure represents the computations that are to be performed within an asynchronous Interrupt Service Routine (ISR) in a real-time environment. As a result, the Interrupt Procedure is primarily an AutoCode-specific concept and is not supported in the SystemBuild simulator.

The **Interrupt Name** field can be used to associate Interrupt SuperBlocks with specific interrupts in the AutoCode scheduler template. Note that the handling of ISRs is platform and operating system specific and requires a knowledge of these software concepts.

An Interrupt SuperBlock cannot have external inputs and outputs, and is required to interact with other system elements through ReadVariable and WriteVariable blocks. Hierarchies of Standard and Macro Procedures can be constructed within the Interrupt SuperBlock. Dynamic blocks with states cannot be included in an Interrupt SuperBlock, although state-like behavior may be achieved using Variable blocks. Hierarchies of standard and Macro Procedures may be constructed within the Interrupt SuperBlock. UserCode blocks (UCBs) or BlockScript blocks in Startup Procedure SuperBlocks cannot have states.

5.4.7 Asynchronous Procedure Execution

Figure 5-1 on page 5-8 illustrates the sequence of initializations that occur in a SystemBuild simulation. Note, only Startup Procedure SuperBlocks are supported during simulation. Figure 5-2 on page 5-9 shows the AutoCode real-time execution sequence, including Startup, Background, and Interrupt Procedure SuperBlocks.

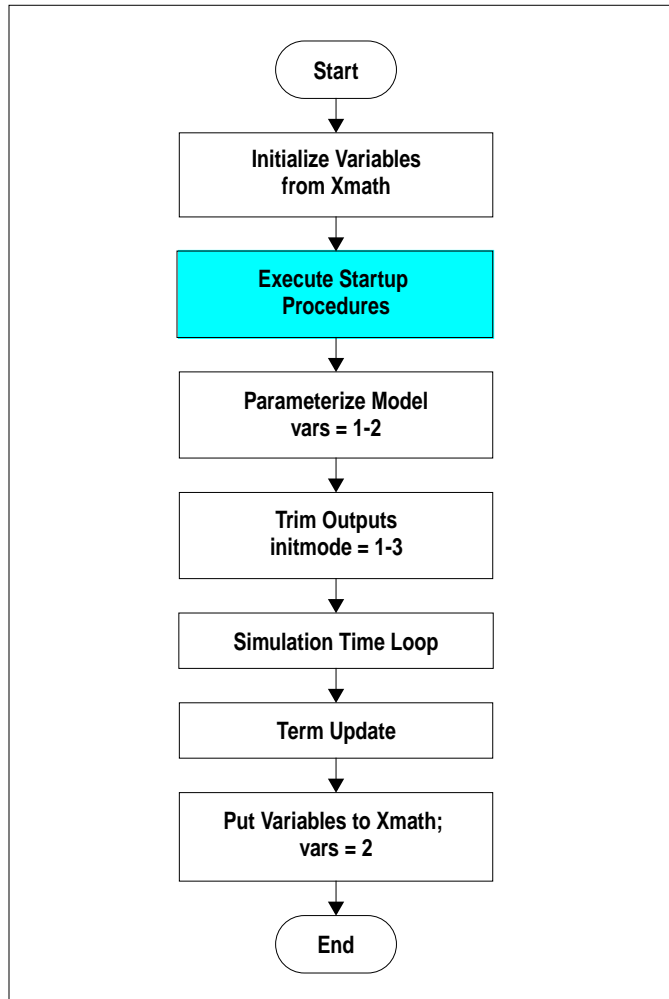


FIGURE 5-1 Pre-Simulation Initialization Steps

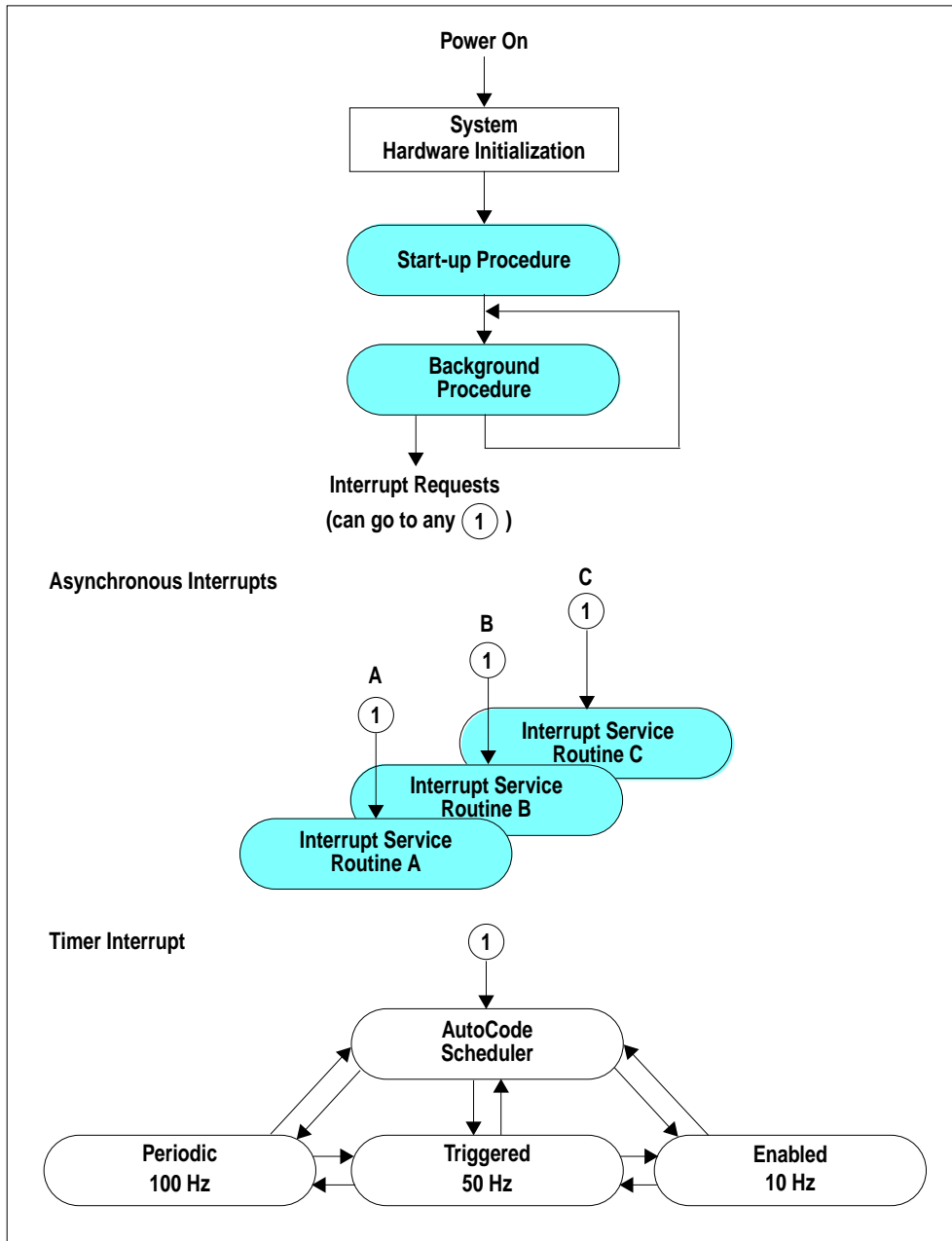


FIGURE 5-2 AutoCode Real-time Application Execution Sequence

5.4.8 Limitations of Asynchronous Procedures

The simulation support of Background and Interrupt Procedures cannot be accurately represented during a non-real-time simulation. As a result, these Procedure references are not executed during simulation.

Typically, the Background and Interrupt Procedures will be directly referenced from within the part of your model that you will use to generate code using AutoCode, but this reference will not be executed by the simulation.

5.5 File SuperBlocks

The File SuperBlock is a useful construct in environments where large system models are created by several engineers working independently. With File SuperBlocks you can create a high-level model where part of the hierarchy is specified in an external file and is represented by a File SuperBlock icon.

To use the File SuperBlock construct, follow this procedure:

1. Identify parts of your system that can be logically grouped as stand-alone datasets. The boundaries might be the physical components of your system, for example, engine, transmission, drive train, or suspension.
2. Save the SuperBlocks associated with each dataset into separate files. Both binary and ASCII files may appear in libraries.
3. Use the `SETSBDEFAULT` command to define the `SBLIBS` variable to list the data files that you have created:

```
setsbdefault,{sblibs="lib1.dat lib2.dat"}
```

4. In the SystemBuild editor, build the diagram using references to SuperBlocks:
 - a. In the catalog browser, double-click on the Libraries icon. You will see the list of filenames in the `SBLIBS` variable. Note that the libraries are static, and therefore the delete, cut, and copy options are not enabled for them.
 - b. Select a library that contains an object you want to load; the SuperBlocks are listed in the Contents view.
 - c. To create a File SuperBlock reference, drag the desired SuperBlock icon into the SuperBlock editor (drag with the middle mouse button on UNIX, and the left mouse button on Windows).

5. The simulator resolves the File SuperBlock references by sequentially searching each library in the order specified in the `sblibs` variable. During the analysis phase, messages appear indicating which library supplies each SuperBlock, where applicable.

A SuperBlock in a library may reference SuperBlocks within itself or another library, but SystemBuild will not resolve SuperBlock references by searching backward through the `sblibs` files: you are allowed to use the same SuperBlock name in more than one library. When there are multiple occurrences of a name, any SuperBlock references to that name will go to the SuperBlock in the library appearing *earliest* in the `SBLIBS` string ([step 3 on page 5-10](#)).

NOTE: When running the simulation from the operating system, specify the library file list using the environment variable `SBLIBS`:

```
UNIX:      setenv SBLIBS "lib1.dat lib2.dat"
```

```
Windows:  set SBLIBS="lib1.dat lib2.dat"
```

5.6 Timing Considerations

Effects of Nesting on Enabled/Triggered SuperBlocks

[Table 5-1 on page 5-12](#) lists the types of Enabled and Triggered SuperBlocks and shows the relationship between parent and child SuperBlocks with regard to inheritance of trigger and enabling signals. For each combination of parent and child status, the entries in the table show whether the child SuperBlock will be Free-running, Enabled, Triggered, or Idle. Enabled and Triggered SuperBlocks can inherit activation signals from their parent SuperBlocks.

Using DataStores

The DataStore block provides an array of scalar memory elements or registers in memory, which may be used in discrete SuperBlocks *only*. These registers are maintained as part of the simulation (and the AutoCode Real-Time scheduler), and do not exist as separate blocks. Especially in the AutoCode context, this property calls for special simulation timing considerations as explained in [Section 5.7](#). For a discussion of AutoCode considerations for DataStores, see [Section 5.9](#). Note that in most cases, using ReadVariable/WriteVariable blocks provides a simpler, though less deterministic, way to pass data between subsystems.

TABLE 5-1 Relationship between Parent and Child Discrete SuperBlocks

Child ↓	Parent →		DP	DN	DE	TP	TN	TT	C	Top
	SuperBlock Type	Enable Selection								
DP	Discrete	Parent	P	F	E	F	F	E	F	F
DN	Discrete	None	F	F	F	F	F	F	F	F
DE	Discrete	Pin number	E	E	E	E	E	E	F	E
TP	Trigger	Parent	N	N	N	N	N	T	N	N
TN	Trigger	None	N	N	N	N	N	N	N	N
TT	Trigger	Pin number	T	T	T	T	T	T	T	T
Key:	F = Free-running		T = Triggered							
	E = Enabled		C = Continuous							
	N = Never executed		P= E if first non-DP parent is E, else F							

5.7 Simulation Timing Properties

This section discusses simulation timing properties, including subsystems and DataStores.

Figure 5-3 on page 5-13 illustrates the timing by which discrete subsystems are updated. In simulation, DataStores are written by a subsystem at the same time the subsystem updates its outputs. In keeping with the SystemBuild requirement that the outputs of every subsystem should be asserted every cycle, the scheduler performs DataStore writing as part of refreshing zero-order holds for all outputs.

The situation differs according to whether the `cdelay` simulation keyword is specified to be true (or `actiming`, which sets `cdelay True`). If `cdelay` is specified, in order to approximate the situation of running AutoCode generated code, a nominal computational delay time is added to the execution time of a subsystem, and the subsystem posts its outputs only at the time it is next scheduled for execution. By contrast, when `cdelay` is set false, no delay is added, the computation time is assumed to be instantaneous, and the outputs are posted (and DataStores written), immediately.

The timing of trigger subsystems is shown in Figure 5-4 on page 5-14. The four types are defined in terms of their output posting requirements.

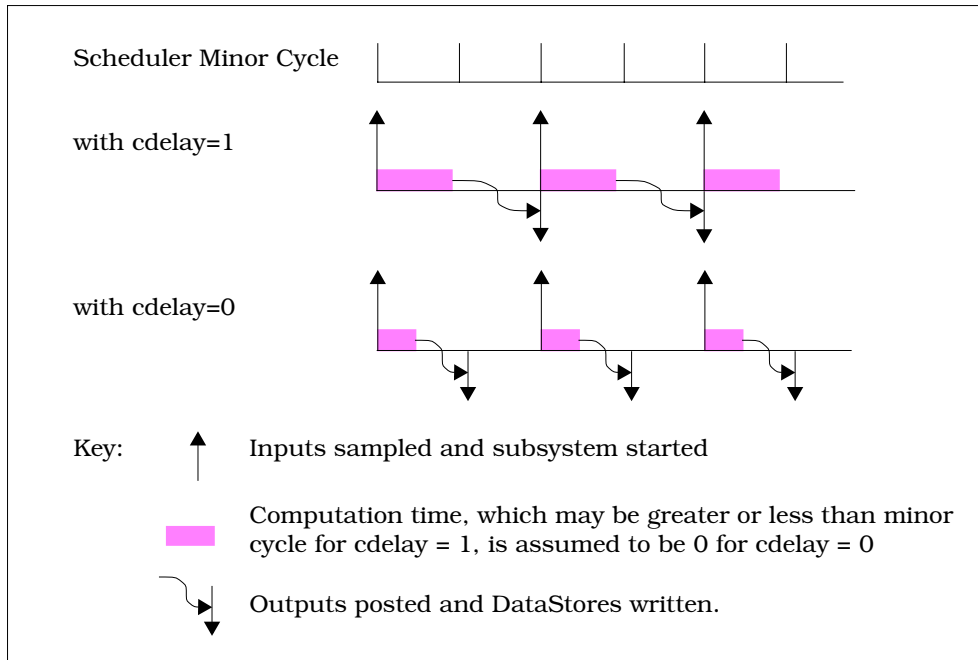


FIGURE 5-3 Simulation, Discrete Subsystem Output Timings

5.7.1 At Next Trigger (ANT)

This trigger subsystem has a variable output timing, in that the outputs of a given cycle are only posted when the next trigger is given for the subsystem. The timings for the three types of trigger subsystem are shown in [Figure 5-4](#).

5.7.2 At Timing Requirement (ATR)

In this case, a user-specified amount of time will elapse from the beginning of execution to the time that the output is posted. This type of subsystem is used when determinacy is an issue, and more important than sheer performance.

5.7.3 As Soon As Finished (SAF)

The outputs are posted at the beginning of the next minor cycle after the subsystem finishes its computations. This type of subsystem is preferred when performance is more important than determinacy.

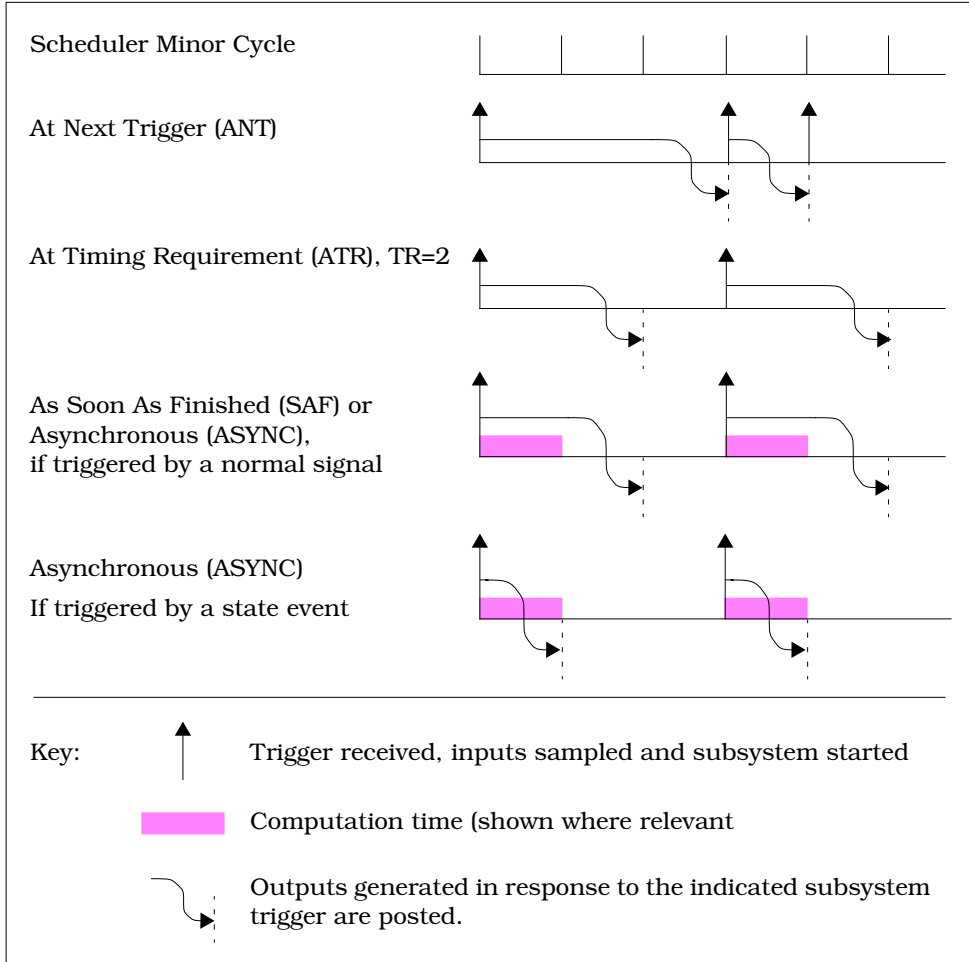


FIGURE 5-4 Triggered Subsystem Output Timings

5.7.4 Asynchronous (ASYNC)

This type of triggered subsystem differs from other triggered subsystem types, both in the way it handles simulation, and the way it is handled in AutoCode. Asynchronous SuperBlocks are designed to replace the Interrupt Procedure SuperBlocks. Not only can they maintain the asynchronous aspect of the Interrupt Procedure SuperBlock, they can support model inputs and outputs, dynamic blocks, states (including states in UCBs) etcetera.

For simulation the Asynchronous subsystem is both leading and trailing edge triggered (i.e., this subsystem will be scheduled if the triggering signal transitions from ≤ 0 to >0 , if it transitions from >0 to ≤ 0). Also, the execution and posting requirements for Asynchronous subsystems differ depending on the type of signal used for triggering.

- If the triggering signal is the output of a normal primitive block or external input, the subsystem will be treated the same as a SAF triggered subsystem, except its priority is higher and the triggering is double-edge.
- If the triggering signal is a state event ([Section 11.9](#)) then the subsystem will execute outside of the scheduler at the exact instance when the event occurs. If the triggering signal is attached to a ZeroCrossing block, and a variable step integrator is used, this allows the asynchronous trigger subsystem to function as an interrupt service routine in the simulator, where the simulated interrupts (from the ZeroCrossing block) occur asynchronously to the periodic portions of the model.

For AutoCode purposes, the Asynchronous subsystem is suited to serve as an interrupt service routine. In the AutoCode environment, the code generated to execute this type of SuperBlock will differ based upon the source of the triggering signal (as opposed to its type, since state events are not supported in AutoCode).

- If the trigger source is internal to the model (that is, it is an output from another primitive block or a SuperBlock from this or another subsystem, then the ASYNC code will be scheduled the same as a SAF triggered subsystem, except its priority will be higher and the triggering is double-edge.
- If the trigger source is an external input that has not been acted upon by a primitive block (even if that input has been brought through several layers of hierarchy) the AutoCode scheduler will not schedule this subsystem. AutoCode will generate a wrapper for the asynchronous subsystem that is intended to be used as an interrupt procedure in your system.

Using an Asynchronous Triggered SuperBlock

Assume a simple continuous plant with transfer function defined as:

$$G(s) = \frac{4}{s + 4}$$

Assume we need a discrete controller that can have a sample period no larger than $T_s = 0.4[s]$. The controller will be simple, and the overall system dynamics can be considered open loop. The controller task is to provide a step input to the plant at a precise moment (not necessarily a multiple of the sampling interval T_s).

A sensor output is available and it will detect the starting time at which we need to apply the step input. The starting time will be the first zero crossing of a sinusoidal signal $sgn = \sin(2\pi \cdot f + \text{phase})$ where $f = .5$ [Hz] and $\text{phase} = \pi/2$.

While modeling the system we should have a practical implementation in mind, such as a target real time controller executing the generated code from the graphically designed controller. Copy the catalog file to your local system as follows:

```
copyfile "$SYSBLD/examples/manual/async_trig_ex1.cat"
```

Load the file.

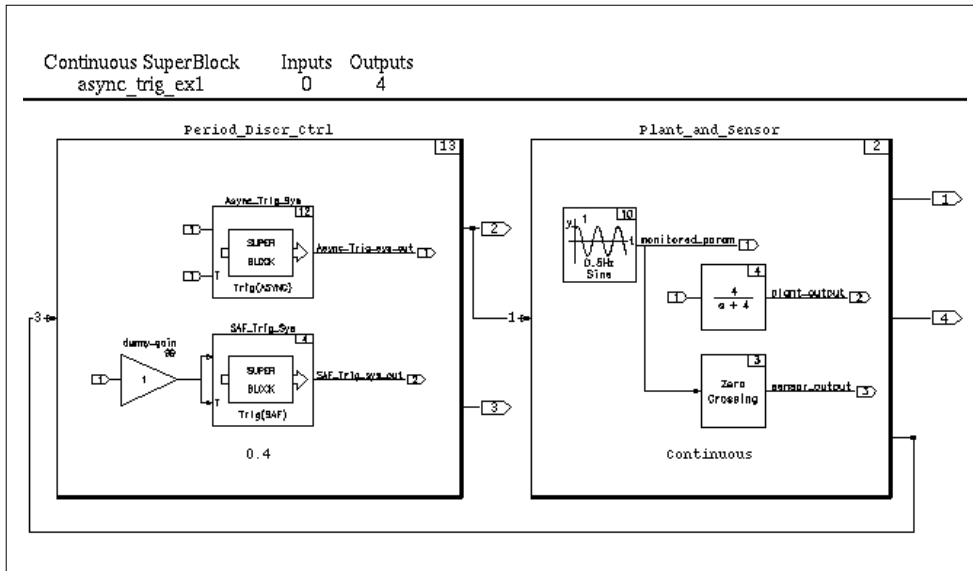


FIGURE 5-5 Asynchronous Trigger Example

As shown in [Figure 5-5](#), in order to demonstrate the difference between a SAF triggered system and an asynchronous triggered system, both triggered systems are built into the controller so that simulation outputs can be compared.

The catalog hierarchy is as follows:

0	async_trig_ex1	Top-level SB
1	Period_Discr_Ctrl	Discrete controller
2	Async_Trig_Sys	Asynchronous triggered system
3	SAF_Trig_Sys	Triggered system
0	Plant_and_Sensor	Continuous plant & sensor

Period_Discr_Ctrl contains the gain block “dummy_gain”; its purpose is to force execution of SAF_Trig_Sys every 0.4[s] (the sampling period T_s). After code generation for Period_Discr_Ctrl the scheduler frequency will be set to $1/T_s = 2.5$ Hz, which would otherwise become the rate of our real time controller. We need to add the dummy gain in our model in order to match sim, (which possibly has a faster scheduler due to the continuous top-level SuperBlock), with AutoCode ($T_s = 0.4$ [s]). Of course $T_s = 0.4$ [s] was one of the requirements for the design.

Note that the Asynchronous Triggered SuperBlock (ATSB) is contained as part of the discrete controller. The location of this SuperBlock is immaterial for simulation purposes, as long as the triggering signal is the output of the ZeroCrossing block. When code is later generated for the discrete controller, the triggering signal for the ATSB will be an external input, which will cause AutoCode to generate a routine suitable for hooking into an interrupt source as the triggering device for the ATSB Subsystem. By using this mechanism, it is possible to use the Asynchronous Triggered SuperBlock to both simulate and generate code for interrupt handlers. For the simulation we use the output of Async_Trig_Sys as the stimulus to the plant; the output of SAF_Trig_Sys is plotted for comparison.

1. Go to the Xmath command area and specify the time vector and the initial value of the %Variable phase:

```
t = [0:.05:1.5]';
phase=90;
```

2. Simulate the model:

```
[te, y] = sim("async_trig_ex1", t, {extend, vars});
```

3. Plot the results:

```
plot(te, y, {strip, marker, x_lab = "time [s]",...
  y_lab = ["Monitored param", "Async Trig system",...
  "SAF Trig system", "Plant out" ]})?
```

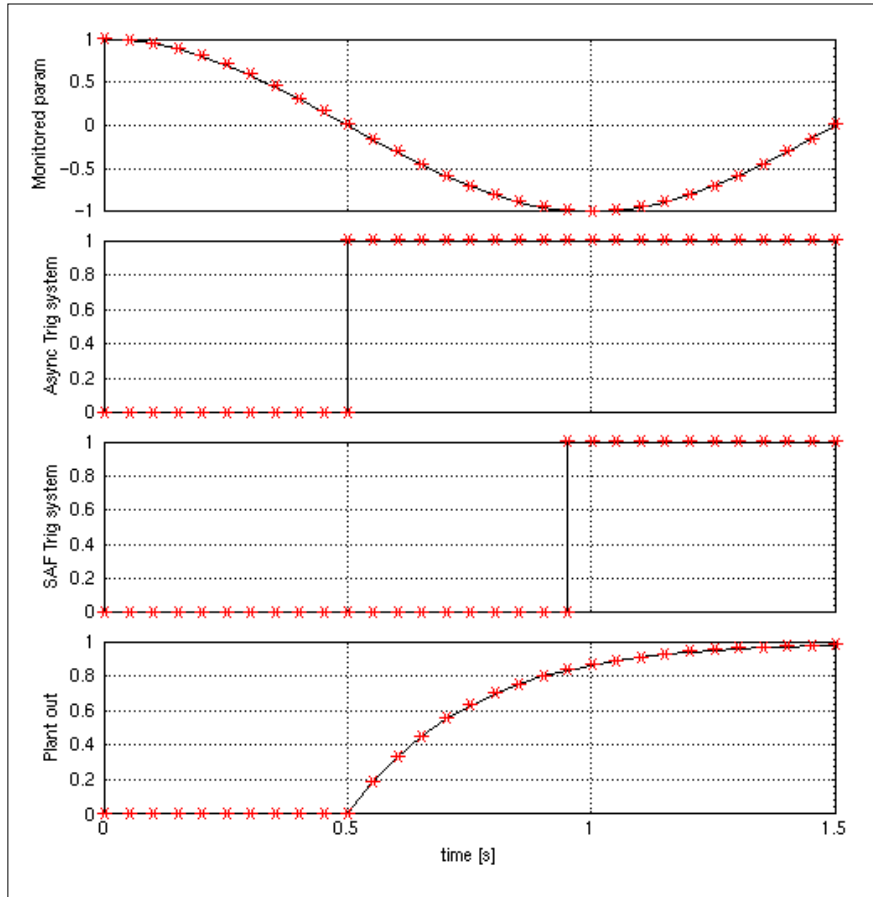


FIGURE 5-6 Asynchronous Triggered Timing vs. SAF

Figure 5-6 shows the following signals:

- Monitored param: a hypothetical monitored signal
- Async Trig system: output from the asynchronous triggered system
- SAF Trig system: output from the SAF triggered system
- Plant out: output from the plant

We can see the unwanted delay of the step signal coming from the SAF triggered system compared to the step signal coming from the asynchronous triggered sys-

tem. This is the main benefit in using an asynchronous triggered SuperBlock for applications similar to our example. Try the simulation with different values of the %Variable phase and observe the change in the “unwanted delay” coming from the SAF triggered system.

AutoCode and the Asynchronous Triggered System

What does Autocode do in order to match the simulation results we just obtained?

While generating code for the discrete controller “Period_Discr_Ctrl”, the call to the “Async_Trig_Sys” subsystem will not be included in the SCHEDULER function, but will be left as an IRQ (interrupt request) call for the real-time controller. This way, the subsystem Async_Trig_Sys will be almost immediately executed from the triggering event (in our case the zero crossing of the sine wave) instead of waiting for the next sample, as SAF_Trig_Sys does. If you are licensed for the AutoCode option, generate code and examine the results to confirm this.

This example is very generic and simple; it does not show any relation between the periodic controller and the plant, nor between the asynchronous triggered controller and the periodic controller. Because such relations will be present in many application it is important to recall that *the asynchronous triggered system will be immediately available after service of the IRQ.*

5.8 Subsystem Priorities

- Higher priorities go to faster subsystems. Faster is defined as higher sampling rate (discrete free-running) or quicker timing requirement (trigger).
- If two discrete free-running subsystems have the same rate and the same skew, the one with lowest subsystem ID has priority.
- If a discrete free-running and a trigger subsystem have the same rate, the free-running subsystem has priority.
- If trigger subsystems have priority, the order of priority is ASYNC, SAF, ATR, ANT.
- Subsystem priorities are used to determine the order in which subsystems that are executed at the same time point are executed, as well as determining which values get written to DataStores if multiple subsystems attempt to write to one DataStore at the same time.

Whenever a system with one or more discrete subsystems is analyzed for simulation or any other purpose, a scheduler frequency (“minor cycle”) is calculated, based on

the Greatest Common Divisor (GCD) of the discrete subsystem rates in the system, *including any trigger subsystem timing requirements*. This may impact the operations of an SAF trigger subsystem with a user-specified timing requirement that is created into a subsystem that is otherwise purely continuous. In this case, the scheduler minor cycle will be equal to the timing requirement, which may mean that the posting of the outputs of the SAF subsystem are unexpectedly delayed.

5.9 AutoCode Timing Properties

DataStores timing features that are peculiar to AutoCode are illustrated in Figures 5-7 through 5-9. In this example, two different subsystems are writing to the same DataStore location, and the differences between the timings of the subsystems creates situations of interest. These examples illustrate that the situation regarding DataStore timing is always determinate, and why it is prudent to avoid situations where two subsystems write into the same DataStore location unless you can ensure there will be no conflict (for example, by making it impossible to trigger or enable them at the same time).

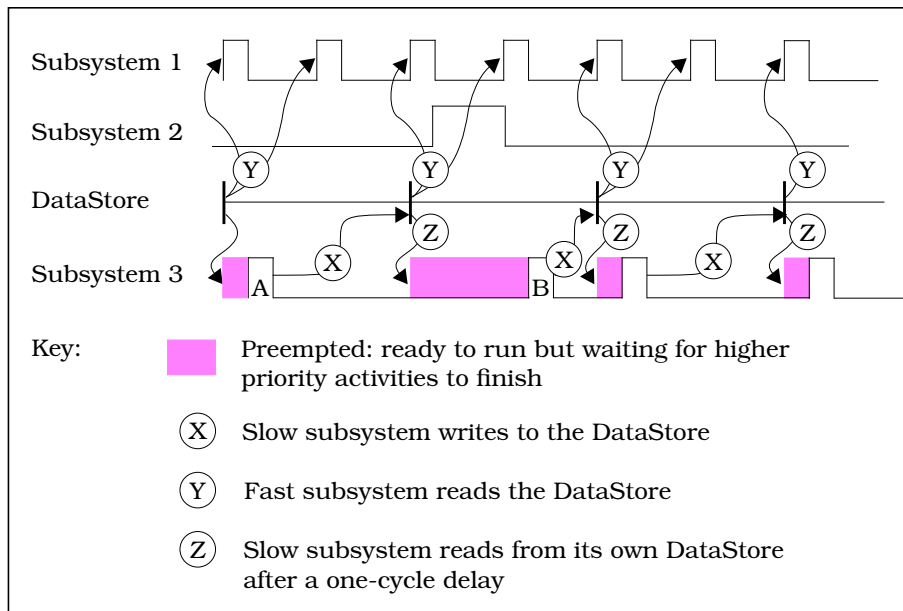


FIGURE 5-7 DataStore Timings

In [Figure 5-7 on page 5-20](#), Subsystem 1 runs faster than Subsystem 3, and thus has higher priority, allowing it to run before Subsystem 3 each time they are both primed for execution together. In other words, in an AutoCode environment, the faster subsystem regularly preempts the activities of the slower subsystem.

Whenever Subsystem 3 runs to completion, it posts its outputs in a DataStore for transmission to other subsystems and to itself. This is indicated by X in the diagram. Although the output of Subsystem 3 might become available before the next execution of Subsystem 1 (as at A), this cannot be guaranteed (as at B, when the occasionally-triggered Subsystem 2 also preempts Subsystem 3). Therefore, to guarantee determinacy, the data from the DataStore will not be made available to Subsystem 1 or other subsystems until the next time that Subsystem 3 is primed for execution. This is the same as would happen with any subsystem if the outputs of Subsystem 3 did not go through a DataStore.

The DataStore functions as a one-cycle delay for the Subsystem 3 outputs that go return to Subsystem 3. This is because the outputs become available the next time that Subsystem 3 is primed for execution, as shown at location Z in [Figure 5-7](#). This contrasts with the fact that, without the DataStore, the outputs of Subsystem 3 would be available within the subsystem on the same cycle as they were generated.

[Figure 5-8 on page 5-22](#) illustrates how two different subsystems write to the same DataStore. In this illustration, Subsystem 1 executes each third scheduler minor cycle, and Subsystem 2 executes each fifth minor cycle. There is no skew time difference between them, but they both write to the same DataStore register element. When they are initiated for execution, Subsystem 1 takes priority, executes first, and has its data posted into the DataStore on its next wakeup time (point A). Soon Subsystem 2 gets its opportunity to execute, and its data is posted at its next wakeup time (point B).

In the meantime, Subsystem 1 has executed again, and its data becomes visible at point C1. Before Subsystem 2 can have its data posted again (at point D), the data from Subsystem 1's third execution is posted in the DataStore location (point C2). At point D, the output of Subsystem 2 becomes visible, to be overwritten again by the output of Subsystem 1 at point E. Finally (point F), both subsystems post their outputs to the DataStore simultaneously. Subsystem 1 is higher priority and prevails. The output of Subsystem 2 at this time will never appear in the DataStore.

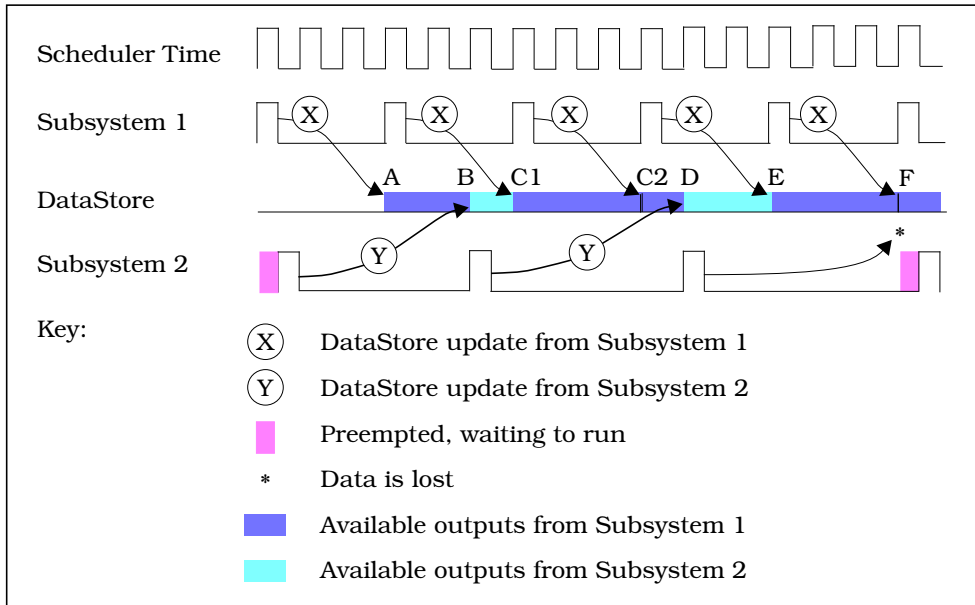


FIGURE 5-8 Writing into a DataStore Register from Two Different Subsystems

This illustrates the way that data in the DataStore may be visible for uneven time intervals, but the situation is always determinate. [Figure 5-9 on page 5-23](#) illustrates a situation where data from a slow subsystem may never show up in the DataStore, and how to work with this situation.

Enabled Subsystem 1 runs exactly twice as fast as free-running Subsystem 2, and thus has priority for execution. However, both subsystems have the same skew or start time, and thus are primed for execution at the same time; this fact is crucial to the discussion.

When Subsystem 1's first data output is ready, Subsystem 2's time has not yet arrived for having its data posted, and the Subsystem 1 output is posted without any conflict. But at point B, both Subsystem 1 and Subsystem 2 receive a wakeup signal at the same time, and Subsystem 1 executes first. At the wakeup point, data from each subsystem from a previous cycle is ready to be posted simultaneously. The resolution is made in terms of Data Priority, which has the same ordering as Execution Priority: the data from Subsystem 1 is posted, and that from Subsystem 2 is lost.

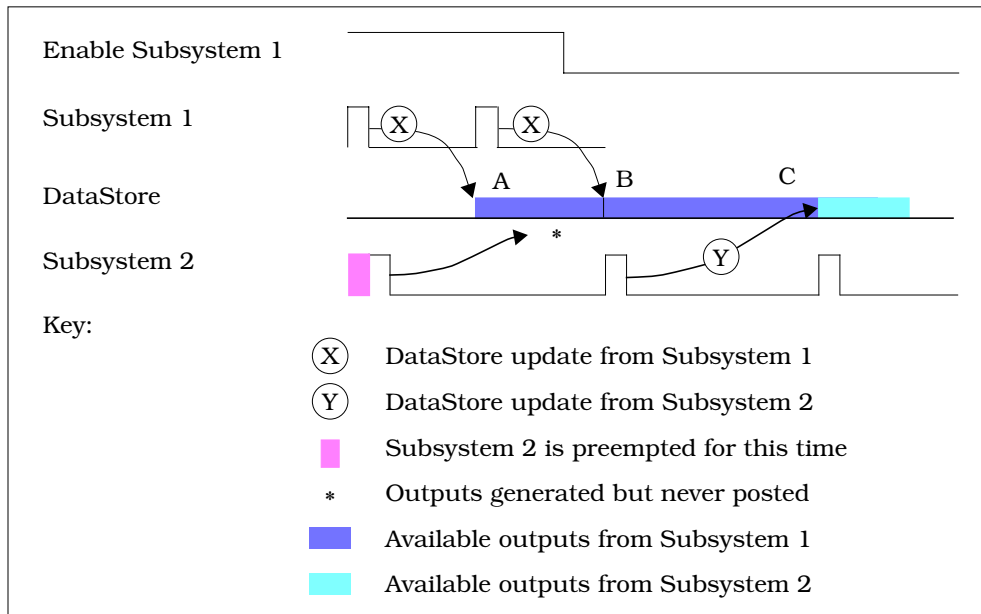


FIGURE 5-9 Enabled Subsystem Writing into a DataStore

This situation continues as long as Subsystem 1 is enabled and running, but when the enabling signal for Subsystem 1 is removed, its outputs stop being posted and the outputs of Subsystem 2 are able to be posted (point C) in the usual manner.

5.10 SuperBlock Transformation

Each SuperBlock has computational timing attributes (continuous or discrete, triggered, etc.) SystemBuild provides a powerful feature that enables you to automatically transform SuperBlocks from continuous to discrete or from one discrete rate to another. There are two ways by which transforms may be accomplished. The Transform SuperBlock dialog (invoked from the Build menu) allows multiple SuperBlocks to be transformed together and provides detailed control over how parameters are affected during transformation. A transformation capability for converting a single SuperBlock is also provided through the SuperBlock Attributes dialog.

Transforming a SuperBlock preserves interconnections and labeling information, but changes all the relevant block parameters in the SuperBlock.

NOTE: Transforming a SuperBlock overwrites the old entry in the SuperBlock catalog; the system does not keep a copy of the old SuperBlock. It is wise to make a copy of a SuperBlock before transforming it.

5.10.1 Transformation Limitations and Implications

Transformations may be between continuous and discrete SuperBlocks or between two different discrete rates. The implications vary depending on the types of blocks in the model. Algebraic, logical, and other blocks without dynamics or memory are unaffected by transformation.

Limitations

- The following blocks are supported only in continuous SuperBlocks and may not be transformed to discrete: ZeroCrossing and Implicit UCBs.
- The following blocks are supported only in discrete SuperBlocks and may not be transformed to continuous:
 - State Transition Diagram, DataStore, and FuzzyLogic blocks
 - Signal Type Conversion blocks,
 - IfThenElse, While, Break, and Continue blocks
 - Condition blocks
 - Procedure SuperBlock references.

Dynamic Blocks

Most dynamic blocks (TimeDelay and Integrators) maintain their coefficients unchanged, except for the compensation for the new sampling interval (T).

Three dynamic blocks: StateSpace, NumDen, and PoleZero blocks, require new coefficients. The method for transforming these three types of blocks between continuous and discrete is based on Tustin's rule (also known as bilinear or trapezoidal). The transformation used between continuous and discrete is:

$$s = \frac{2(z - 1)}{T(z + 1)} \quad \text{Eq. 5-1}$$

Transforming between discrete rates is done by first converting the current rate to continuous time, then converting from continuous time back to discrete with the new rate, always using Tustin's rule. See [Example 5-2](#).

NOTE: When transforming from discrete to continuous or discrete to discrete, the new rates and coefficients must be applied with caution. During the transformation, we move from the z domain to the s domain; inappropriate results can be generated from StateSpace, NumDen, and PoleZero blocks.

EXAMPLE 5-2: Transforming a Block

Assume a block sampling at 0.1 seconds in Num-Den format with transfer function in the z domain = $1/z^2$, and we want to either change the sampling rate or convert it to continuous. The resulting transfer function in the s domain is:

$$\frac{(s - 20)^2}{(s + 20)^2}$$

This continuous block represents a non-minimum phase system that approximates neither the step nor the impulse response of the discrete system with transfer function = $1/z^2$, which one would assume to be a two sample delay.

However, the continuous subsystem *does* approximate a two-sample delay when provided with sinusoidal inputs.

The better approach when transforming discrete rates is to go back to the original continuous system and apply Tustin's rule with the new sampling rate.

Gain Block

The Gain block is transformed using the pure- z transform. This transform method is impulse-invariant, and it preserves the frequency domain characteristics of the transfer function such as damping ratios and damping frequencies.

Integrators and PID Controller

Blocks that contain integrations (Integrator, LimitedIntegrator, and PIDcontroller) retain the same coefficients and are simply changed to their discrete equivalents. A discrete integrator can be selected from the block's Parameters tab. Integrators for these blocks are:

- 1 Forward Euler
- 2 Backward Euler (Default)
- 3 Tustin (Trapezoidal)

5.11 Transformation Methods

SuperBlocks can be transformed in several ways:

- Transform nonlinear dynamic systems from Xmath using the `lin` and `discretize` functions.
- Use the Catalog Browser Tools→Transform option.
- Change the SuperBlock type from the SuperBlock Properties dialog box.

NOTE: If you are uncertain about the outcome of the transformation, you should update the SystemBuild Catalog (from the editor, select File→Update), then save the SystemBuild Catalog before proceeding.

5.11.1 Transformation Using the Transform SuperBlock Dialog

Before the transformation, go to the Catalog Browser and select the SuperBlock(s) you want to transform; if you want to transform hierarchy, select Edit→Hierarchy Select Mode before making a selection. All the SuperBlocks in the hierarchy will be transformed to the new settings; the only restriction is that the hierarchy cannot contain any State Transition Diagrams or DataStores. If any of the SuperBlocks is triggered before you perform the transformation, it will become enabled.

After the selection, select Tools→Transform. The Transform SuperBlock dialog appears. You can change the block type. Other fields are as follows:

1. Discrete to Discrete Transformation: Rate Only or Rate and Block Coefficients.

Coefficients apply for NumDen, StateSpace, and PoleZero blocks in a discrete-to-discrete transform. Frequency-normalized systems require a rate-only transform.
2. Transform Initial Conditions (of StateSpace Blocks). Enable or disable. See [Section 5.11.3 on page 5-28](#).
3. Sample Period (0 = continuous).
4. Sample Skew (the first period).

This dialog also allows timing attributes of discrete SuperBlocks (Free-running, Enabled, Triggered, and Procedure) to be changed.

For additional help, press the **Help** button.

5.11.2 Transformation Using the SuperBlock Properties Dialog Box

Transforming can also be performed from the SuperBlock Properties dialog. See [Table 5-2](#) for a summary of the differences between the two methods.

TABLE 5-2 Transforming SuperBlocks

	From SuperBlock Properties Dialog	Catalog Browser Tools Menu
Action	Click on Continuous/Discrete field to change; type in sampling rate and skew.	
Result	Only the current SuperBlock being edited is transformed.	The SuperBlock and optionally its lower-level SuperBlocks are transformed.

To use the SuperBlock Properties method, simply invoke the dialog and select a new type. Note that with both methods, the transformed SuperBlocks overwrite the original SuperBlocks.

If you change the rate from the SuperBlock Properties dialog, the changes are more restricted:

1. Only the current SuperBlock is transformed: no hierarchy may be selected for transformation. However, timing information for Condition blocks and Procedure SuperBlocks that are direct children of this block are changed.
2. Initial conditions are not transformed.
3. If the rate is changed from one discrete rate to another (where both are non-zero), the rate is updated but no transform occurs; there is no change in dynamic block coefficients. (This lets you preserve sample rate normalized systems.)
4. Changing between the different forms of discrete SuperBlock (Free-running, Enabled, Triggered, and Procedure) is performed from the SuperBlock Attributes dialog. To change between Continuous and Discrete *non*-Free-running, first change to discrete Free-running and then to the other discrete form.

5.11.3 Initial Condition Transformations

When you enter a SuperBlock that has been transformed, it appears identical to a SuperBlock originally created at that rate, with the following exception. If the SuperBlock was transformed from the Catalog Browser Transform tool and Transform Initial Conditions was enabled, then all StateSpace blocks in the transformed SuperBlock(s) will have an Initial Conditions tag added to their initial conditions in the block parameter dialog, identifying the rates from which the SuperBlock was transformed.

The initial conditions themselves are not changed immediately, but the Initial Conditions tag is a reminder that they will be transformed at simulation time. If you later modify the initial conditions of a modified StateSpace block in a transformed SuperBlock, the initial conditions are henceforth assumed to correspond to the current rate of the SuperBlock, the tag disappears, and no transformation occurs at simulation time.

5.11.4 Undoing a Transformation

- A transformation initiated from the Transform SuperBlock dialog cannot be reversed.
- Transformations initiated from the SuperBlock Properties dialog can be undone with the general function Edit→Undo All. As the name implies, the SuperBlock will revert to its state at the time the display was last updated to the catalog. All changes to the SuperBlock and its displayed children will disappear.

6

SystemBuild Customization

This chapter details how to customize SystemBuild with the `/etc/user.ini` file. Users on UNIX platforms can perform additional customization with the resource file `/etc/Sysbld`. This chapter contains the following sections:

- [user.ini File Format](#)
- [Printer Settings \(UNIX\)](#)
- [Comment Editor](#)
- [Custom Menus](#)
- [SystemBuild Resource File \(UNIX\)](#)

6.1 user.ini File Format

The file `$(SYSBLD)/etc/sysbld.ini` defines SystemBuild's default printing and editor settings, its menus, and other resources. You cannot change this file, but, you can customize SystemBuild by creating a similarly formatted file named `user.ini` that overrides or adds to `sysbld.ini` defaults. If your custom `usr.ini` file is placed in `$(SYSBLD)/etc` (by someone with root or administrator privileges) all users will see the customization (be sure to make a copy of the sample file first). SystemBuild will also read this file if it is in your start-up directory.

The basic `user.ini` file format is shown in [Example 6-1 on page 6-2](#). Use Xmath's `copyfile` command to copy this file to your local start-up directory:

```
copyfile "$(SYSBLD)/etc/user.ini"
```

The `user.ini` file has two parts: a `COMMON_SECTION` for customizing environmental options and specifying text editors, and a `SUPERBLOCK_EDITOR_SECTION` in

which you can specify custom menus for the editor. Your local `user.ini` file only needs to contain settings that differ from or override `$/SYSBLD/etc/sysbld.ini`. However, your customized `user.ini` file **must** present information **in the same order** as that shown in the sample file. At a minimum, you must restart SystemBuild after any change to `user.ini`. If your changes involve new scripts or programs that Xmath must know about, you must restart MATRIXx.

EXAMPLE 6-1: Sample `user.ini` File

```

=====
#                                     user.ini
=====
#                                     SystemBuild Configuration file for User Menu
#-----
# This file contains settings used to customize the appearance and behavior
# of SystemBuild. This file provides examples for creating custom menus.
# The example template can be used to specify menu items. Any line in this
# file which starts with pound sign (#) is ignored as comment by SystemBuild
#
=====

#-----
# COMMON_SECTION allows specification of options and text editors
#-----

[COMMON_SECTION]

[OPTIONS]
TempDir      = "/tmp/"
IconDir      = "/usr/local/sysbld/icons"
PaletteDir   = "/usr/local/sysbld/palettes"
TextEditor   = "/usr/local/bin/emacs"
PrintCommand = "lp -h"
PrinterOption = "-d"
PrinterName  = "hp13"
PrinterName  = "hp9"

[TEXT_EDITORS]

#-----
# Unix example

#TextEditorItem = CommentEditor
# Name          = "xemacs"
# Path          = "/usr/local/bin/xemacs"
# Extension     = "txt"

```

```

#-----
# PC example

#TextEditorItem = CommentEditor
# Name      = "Word Pad"
# Path      = "C:\Program Files\Windows NT\Accessories\wordpad.exe"
# Extension = "rtf"

#=====
# SUPERBLOCK_EDITOR_SECTION allows specification of custom menus
#=====

[SUPERBLOCK_EDITOR_SECTION]
# SuperBlock Editor

[MENU]

#-----
# MenuItem = PulldownMenu
# Label    = &Custom
# Help     = User defined Menus

# MenuItem = PushButton
# Label    = ls
# Help     = Lists the files in the current directory through Xmath
# FuncType = Xmath
# Function = oscmd ("ls")

# MenuItem = Separator

# MenuItem = PushButton
# Label    = pwd
# Help     = Prints current working directory in Xmath
# FuncType = Xmath
# Function = oscmd("pwd")

#-----
# MenuItem = PulldownMenu
# Label    = &System
# Help     = User defined System Messages

# MenuItem = PushButton
# Label    = &Xterm...
# Help     = Brings up an Xterm
# FuncType = System
# Function = /usr/bin/X11/xterm

# MenuItem = PushButton
# Label    = &Calendar
# Help     = Brings up a Calendar
# FuncType = System
# Function = /bin/calendar

```

6.2 Printer Settings (UNIX)

User.ini printer settings are ignored on Windows platforms; they will be read for UNIX systems only.

To change the list of printer names and the printer command, edit the [OPTIONS] block of the [COMMON_SECTION], as shown in [Example 6-1](#). The PrintCommand field specifies the UNIX print command without the printer option. The PrinterOption field specifies the option to be used before the printer name. The PrinterName specifies the names of the available printers.

6.3 Default Text Editor

The default text editor is used when you are typing directly into the Icon tab or Comment tab text area.

NOTE: The comment editors ([Section 6.4](#)) require different settings; they are launched independently rather than used in the block diagram.

The default text editor programs are vi (UNIX), and Notepad (Windows NT/95). You can change the default text editor from the .ini file or from your operating system command line.

To change the default text editor in your .ini file, go to the [OPTIONS] section, and alter the TextEditor definition (see [Example 6-1](#)). For example.

```
TextEditor = "/usr/local/bin/xemacs"
```

Alternatively, type the appropriate command from your operating system command line:

```
UNIX:      setenv EDIT_COMMENT Editor_name
```

```
Windows:   set EDIT_COMMENT=Editor_name
```

Editor_name is the name of your editor program (e.g., xemacs, wordpad, etc.). The change will take effect after you close and reopen SystemBuild.

6.4 Comment Editor

Text entered into the Comments tab can be used to document your work and/or generate inputs for the DocumentIt program. SystemBuild allows you to choose a text editor for the Comment tabs of the SuperBlock, block, or State Transition Diagram (STD) Bubble or Transition dialogs. The text editor operates just as though you had invoked it from the command line. To return to SystemBuild, exit the editor, using the editor's normal exit procedure.

You can remove unwanted editors or add new ones as demonstrated in [Example 6-1](#). See the [TEXT_EDITORS] block.]

The comments are stored as part of the catalog file when the model is saved, or when a Real Time File (RTF) is generated. For a primitive block (not a SuperBlock), the Comments are attached to the block dialog box. Although the comments document is stored with the model file and cannot be accessed from outside SystemBuild, you may save the file from within the text editor so that it can be manipulated or used elsewhere.

A SuperBlock has two dialogs that have different meanings for the way that documentation is generated:

- Comments in a SuperBlock Properties Dialog.

The comments document attached to the SuperBlock Properties dialog may be referred to as a *root document*, and corresponds to a Computer Software Component (CSC) or Low-Level Computer Software Component (LLCSC) document for purposes of MIL-STD-2167A.

- Comments in a SuperBlock Block dialog.

This dialog box pertains to one instance of the SuperBlock; *i.e.*, to just one of the potentially many references to this SuperBlock that may be scattered about your block diagram. The comments document attached to this dialog box may be referred to as a leaf document, and corresponds to a Computer Software Unit (CSU) document according to the requirements of MIL-STD-2167A.

6.5 Custom Menus

You can define one or more menus in the SystemBuild editor menu bar. Your menus may invoke MathScript functions or commands, send a command to your operating system, or execute a shell script. (You cannot alter the standard SystemBuild menus or their contents. The Catalog Browser and Palette Browser menu bars cannot be changed.)

Custom menus are defined in the platform independent ASCII file named `user.ini`. In it you can add multiple custom menus; in the Editor, they will appear before the standard Help menu in the order they are defined in the file.

- To create a menu, you must first specify `menuItem=Pulldown`. Menu Items that will appear on the pulldown menu can be `PushButton` (a normal menu entry), or `Separator` (a dividing line).
- The Label is the text that will appear on the menu bar or menu, depending on the `menuItem` type.
- The Help field is ignored on Windows platforms. On UNIX systems you have access to the message area at the bottom of the SystemBuild editor. When the cursor is over the labeled menu item, the text is shown in the editor message area.
- `FuncType` can be either `Xmath` or `System`. If `Xmath` is specified, the call in `Function` is sent to `Xmath`. If `System` is specified, the call is sent to the operating system and a separate system process is started.
- `Function` is a legal call to `Xmath` or your operating system; the `FuncType` must match this call syntax.

NOTE: You must restart SystemBuild whenever `user.ini` is modified or a new `user.ini` is introduced.

6.5.1 A Sample `user.ini` that Calls MSCs

`$SYSBLD/examples/sbmenu` contains the file `user.ini` and two MathScripts that are called by the custom menus specified in the `.ini` file.

The basic steps to create your own custom menus are:

1. Create a `user.ini` file ([Section 6.1](#)). You can copy a sample file from the SystemBuild distribution; the sample calls two MSCs, so copy them as well:

```
copyfile "$SYSBLD/examples/sbmenu/forcedsave.msc"
copyfile "$SYSBLD/examples/sbmenu/renumber.msc"
copyfile "$SYSBLD/examples/sbmenu/user.ini"
```

COPYFILE copies the files to your current working directory.

2. Restart MATRIX_x.
3. Try loading a model and calling the MSCs.

6.5.2 A Typical Template for User Menus

A typical template for menu item declaration is shown in [Example 6-2](#); this file can be copied from `$$SYSBLD/examples/sbmenu/user.ini`. The lines that start with the pound (#) sign are comments. The template is structured into sections and sub-sections. A typical option is set with the appropriate tag or identifier (for example, the label of `ls`). To specify a pull-down menu, use `MenuItem=PulldownMenu`, followed by `MenuItem=PushButton` definitions. Note, this template contains UNIX-specific system calls that can be commented out or deleted as necessary.

EXAMPLE 6-2: Typical user.ini with Custom Menus

```

=====
#                               user.ini
#-----
#                               SystemBuild Configuration file for User Menus
#-----
# This file contains my personal custom menus.
# Any line that starts with pound sign (#) is ignored as # a comment.
#
#-----
[SUPERBLOCK_EDITOR_SECTION]
# SuperBlock Editor

[MENU]

#-----
MenuItem = PulldownMenu
Label    = &Global Diagram Changes
Help     = These items globally change blocks in the current SuperBlock.

MenuItem = PushButton
Label    = &ReNUMBER
Help     = Renumbers all blocks in current SuperBlock starting with 1.
FuncType = Xmath
Function = RENUMBER

#-----
MenuItem = PulldownMenu
Label    = &Miscellaneous
Help     = Miscellaneous functions and commands.

MenuItem = PushButton
Label    = &ForcedSave
Help     = Force save to an ASCII file named by time: dYYMMDDtHHMMSSsave.asc
FuncType = Xmath
Function = FORCEDSAVE

```

6.6 SystemBuild Resource File (UNIX)

On UNIX systems, the ASCII file `$(SYSBLD)/etc/Sysbld` contains the SystemBuild application defaults.

- To change your SystemBuild colors or the size and placement of SystemBuild window, make a local copy of this file and edit it as described in the following sections.
- If this file is modified in the `$(SYSBLD)/etc` directory, the new values will become the defaults for all users who do not have a local copy of `Sysbld`.

When initializing your configuration, SystemBuild looks first in your working directory, then in the home directory, then finally in `$(SYSBLD)/etc`.

CAUTION: Version 6.0 and higher only read the Sysbld file (note the case). If you have a local copy of an obsolete SysBld file, merge your custom settings into a local Sysbld file, then delete the obsolete file.

6.6.1 Controlling Colors

Colors, sizes, and positions of the SystemBuild and Interactive Simulation (ISIM) windows can be adjusted by creating a SystemBuild defaults file named `Sysbld`.

Foreground and Background

Foreground and background colors in the SystemBuild and Interactive Simulation windows are each controlled by two variables in the `Sysbld` file. The variables are listed in [Table 6-1](#).

TABLE 6-1 SystemBuild and ISIM Color Defaults

Variable	Default
SystemBuild Editor	
<code>sysbld.background</code>	white
<code>sysbld.foreground</code>	black
Interactive Simulation (ISIM) Editor	
<code>isim.background</code>	white
<code>isim.foreground</code>	black

The selection of colors that you may use in the screen specifications on UNIX platforms are listed in the file `/usr/lib/X11/rgb.txt`, along with their RGB color specifications.

SystemBuild and ISIM Color Settings

You can define a maximum of 14 colors in Sysbld. Colors are available in SystemBuild through the color settings in the block dialog boxes. To view colors for the higher hexadecimal numbers (those represented by an alphabetical character) select a block, press 9, then press the apostrophe key repeatedly until the desired color is displayed. The variables to be defined for these colors are `sysbld.color1` through `sysbld.colorE` for SystemBuild, and `isim.color1` through `isim.colorE` for Interactive Simulation. The color names are the same as for the foreground and background colors.

6.6.2 Resizing, and Repositioning the Display

Each window's position and size are defined by four numbers: the x and y location of the screen window's lower-left-hand corner, and the window's width and height. The numbers are percentages of the screen's width and height, as appropriate. [Table 6-2](#) lists the variables and their permissible ranges.

TABLE 6-2 Display Sizing and Positioning Variables

Variable	Range	Default
SystemBuild Window		
<code>sysbld.percent.x_offset</code>	$1 \leq X \leq 99$	28
<code>sysbld.percent.y_offset</code>	$1 \leq Y \leq 97$	25
<code>sysbld.percent.width</code>	$1 \leq W \leq 98$	70
<code>sysbld.percent.height</code>	$1 \leq H \leq 96$	65
Interactive Simulation (ISIM) Window		
<code>isim.percent.x_offset</code>	$1 \leq X \leq 99$	2
<code>isim.percent.y_offset</code>	$1 \leq Y \leq 97$	0
<code>isim.percent.width</code>	$1 \leq W \leq 98$	72
<code>isim.percent.height</code>	$1 \leq H \leq 96$	60

The dimensions are the percentage of the full-screen width or height inside the border of the window. The maximum values of the vertical percentage dimensions are slightly lower than the horizontal dimensions to allow for the wider label border at the top of each window, which is a system-dependent value equal to about 2% of the screen height. The minimum width or height of a window is approximately 25% of the full screen width or height.

7

Simulator Basics

7.1 Introduction

Once you have built a model in the SystemBuild Editor you're ready to simulate and analyze your system using the SystemBuild simulator. When prototyping a system design, simulation tools allow you to readily perform experiments by providing an immediate analysis of the current system. This chapter explains how to use the simulator and customize your simulations with simulation keywords. We also discuss the following related functions: `analyze`, `creatertf`, `lin`, and `simout`.

7.1.1 Simulation User Interfaces

SystemBuild simulator can be accessed three different ways:

- from the SystemBuild editor simulation dialog
- from the Xmath command area
- from the operating system command line

This variety affords flexibility in running, analyzing, and modifying your models,

7.1.2 SystemBuild Editor Simulation Interface

The simulator may be invoked from the SystemBuild editor by selecting Tools→Simulate. In the dialog box presented (see [Figure 7-1 on page 7-2](#)), you may enter the time and input variables for your simulation, specify selected simulation keyword options (including fixed-point arithmetic), and enter the output variable name. This is the preferred way to invoke the simulator. Press the **Help** button on this form for more information about how to select simulation options.

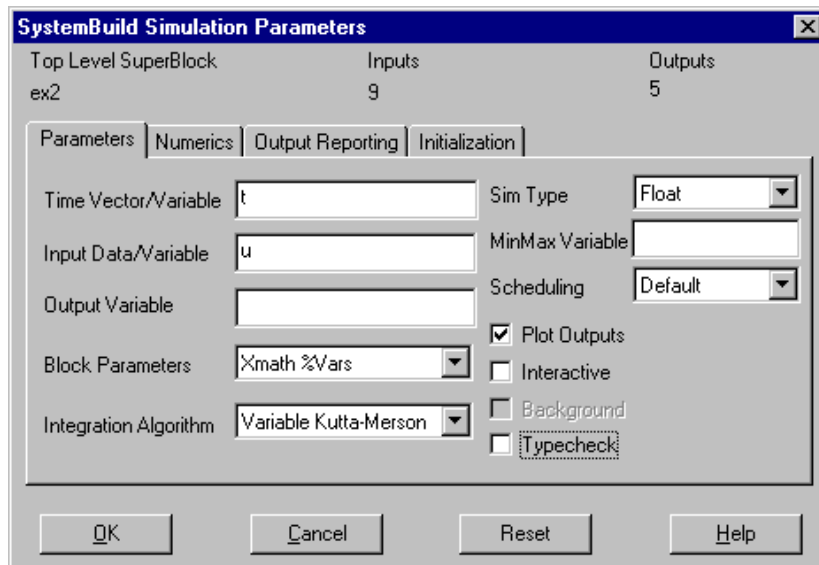


FIGURE 7-1 Simulation Selection Dialog Box

7.1.3 Xmath Simulation Interface

The `sim` function at the Xmath command line allows you to specify a model currently loaded in the SystemBuild catalog (the model does not have to appear in a SystemBuild window), the inputs to that model, and any simulation keyword options. The `sim` function can be incorporated in Xmath scripts, making it possible to set up an environment, run multiple simulations, and analyze the results. This interface is also well-suited for batch mode, where many simulations can be run unsupervised.

Once the simulator has been invoked from Xmath, it remains memory-resident, so that additional simulations run much faster than the first. (If for some reason you have an active `simexe` process that will not close properly, you can use `undefine simexe` to terminate it.)

Sim Function Syntax

The basic syntax of the `sim` function is included here. If you have questions about the syntax of the `sim` function or its keywords, type `help sim` in the Xmath command window.

Two formats are provided to invoke the simulator from the Xmath command line.

```
yPDM = sim("model", t, u, {keywords})
[t, y] = sim("model", t, u, {keywords})
```

For the `yPDM` form of the `sim` function, the domain of the output Parameter-Dependent Matrix, `yPDM`, is the time vector, `t`. For example:

```
y1 = sim("MySystem", t, u)
```

The above call simulates a model named `MySystem`. It returns the result as the PDM named `y1`. The following call returns the output column matrix `y` and the `sim` time column vector `t`.

```
[t,y] = sim("MySystem",t,u)
```

`model` is a text string, which is the name of the top-level SuperBlock in the System-Build editor.

`sim` keywords, like keywords in MathScript functions, are enclosed in braces, and separated by commas. When a keyword is assigned a string value, the string must be enclosed in quotes. See the online help for a complete description of all keywords.

`t` is the required time vector, which must be strictly increasing, and typically starts with zero. It is conventional, but not necessary, to name the time vector `t`; it can have any valid Xmath variable name (alphanumeric plus underscore; no more than 31 characters; first character not numeric). For example:

```
t = [0:0.1:10]';
```

This creates a column vector of 101 values, starting with 0 and increasing in increments of 0.1 to 10. the square brackets are required; the apostrophe (') transposes the matrix (vector in this case), and the semicolon suppresses echoing the 101 values to the screen.

`u` is an input data matrix (required if the model has external inputs), ignored if `t` is a PDM. Both `t` and `u` must be of the same row dimension. The column dimension of `u` must match the number of external inputs of the model. It is conventional, but not necessary, to name the input vector `u`; it can have any valid Xmath variable name (must be alphanumeric, alpha character first, no more than 32 characters).

The following call creates a vector of the same dimension and orientation as `t`, consisting of all ones, and assigns it the variable name `u`.

```
u = ones(t);
```

Existing `t` and `u` variables can be used in the simulation dialog.

Given a SuperBlock with three inputs, create an input compatible with the output dimensions:

```
t = [0:0.1:10]';
u = [t,t,t];
y1 = sim("model",t,u'); # Time and input row dimensions doesn't match
y2 = sim("model",t,u); # Works.
```

Background Simulation

The `bg` keyword causes a simulation to run in the background, freeing Xmath for other work. This feature allows multiple simulations to run simultaneously. You can monitor the progress of a simulation by watching the status of its output variable on the Xmath stack. Therefore, if you wish to run multiple simulations, you must be sure that the output variable names are unique (i.e., if you issue three simulation commands and the output variable is `y` for each one, the ones that finish first will be overwritten).

The familiar Xmath `WHO` command displays the variables in the current partition; if the variable is stable, its dimension is shown; if it is still being calculated, i.e., if the simulation is still running, the variable is followed by `busy` and the job number of the related simulation process. See [Example 7-1](#).

EXAMPLE 7-1: Using `who` and `stop job`

Type `WHO` to list Xmath variables. If a simulation output variable is followed by `busy`, the simulation is still running. If your machine is fast, you may need to specify a lengthy time vector in order to observe this.

`who`

```
main:
      t -- 10001x1
      u -- 10001x2
      y1 -- busy (job #1261)
      y2 -- busy (job #1262)
      y3 -- busy (job #1263)
```

To stop a particular job, type `stop job =`, followed by the job number:

```
stop job = 1262
```

```
Sim is stopped.
```

Simulator Termination

The simulator will normally terminate automatically at the end of a simulation. Simulation termination conditions include:

- End of the simulation input time-line.
- Simulation error (divide by zero, square root of negative number, etcetera).
- Error reported by UserCode Block
- Stop Block encountered with input signal greater than zero

For interactive simulation (see ISIM, [Chapter 8](#)) each of these conditions terminate the current simulation, however, the simulation may be restarted in the graphical environment. Selecting File→Exit terminates ISIM.

In addition to the above, the user may decide to abort the simulation at any time by hitting Ctrl-Break (Windows) or Ctrl-C (UNIX) in the Xmath window as long as the simulator is running in foreground mode (the `-bg` flag is not being used). These key combinations will immediately terminate the currently executing simulator, however, saved output data may be lost, and the termination section of UserCode Blocks will not be invoked.

7

7.2 Parameterization

It is often useful to study the effects of changing one or more parameters in a system and running repeated simulations. In selected fields in each block's parameter dialog, you may specify an Xmath variable name to be used in place of the block's field data. Simply enter the variable name in the `%Variable` field of the block form. The variable name can be entered with a partition specified (`partition.variable_name`) or without (`variable_name`). If you specify a partition name, SystemBuild will look there to resolve variable names, not in the default partition.

You may assign a default value for the variable. You can automatically copy the Variable with its default value into the Xmath workspace. When you have typed the default value and the Variable name, after you press **Return** on the keyboard you can type **Ctrl-p**. This copies the variable name and value into the Xmath workspace, and registers the fact that it is a `%Variable` that can be changed at simulation time. By contrast, you can set the default value in the block form to be the same as the Xmath value of the variable by entering the variable name in the value field of the block form and pressing **Return**.

7.2.1 The Simulation vars Keyword

The `vars` keyword provides control over whether the value of your model's parameters can be changed using Xmath variables.

If the `vars` keyword is asserted, the simulator will pick up %Variables specified in the model and use them for simulation. Since the default for the option is on, parameter variables specified are always used unless you specify otherwise. If you desire the default block data, specify `{vars=0}` in the `sim` options keyword list.

The block will use its default value if its referenced Xmath variable does not exist or if there is an error in the variable data. Variable errors include:

```
variable's values are out of range  
variable's dimensions are incorrect  
variable does not match the expected data type  
variable not found in this scope
```

7.2.2 Parameter Variable Scoping

Parameter variable scoping uses the hierarchical nature of SystemBuild SuperBlocks in conjunction with Xmath partitions to allow you to assign separate parameter values to each of multiple instances of a SuperBlock, based on the partitions of the calling SuperBlocks.

As mentioned in the previous section, you may enter a parameter variable in one of two ways, either with or without a partition specified. If you enter a parameter variable with a partition, the variable under the specified partition is used during simulation instead of a value from the working partition. In this case there would be no hierarchical scoping of the variable because it is explicitly specified.

Rather than specify the partition on a block-by-block basis, you may enter a partition name in the `Partition:` field of the SuperBlock Reference dialog. The simulator will then look up the SuperBlock hierarchy for the existence of a partition name. If the partition is specified, the simulator looks for each block parameter variable under that partition. This is referred to as *parameter scoping*—the variable is in the scope of the SuperBlock reference's partition.

If no partition is specified in the SuperBlock Reference dialog for this instance, the simulator continues its search for the next SuperBlock up the hierarchy. If the search through the SuperBlocks is exhausted, the simulator finally looks for the variable in your current working partition. See [Example 7-2 on page 7-7](#).

EXAMPLE 7-2: Demonstrating Parameter Scoping

This example demonstrates the flexibility of parameter scoping; see [Figure 7-2](#). In Xmath you are working in partition main and have created three partitions X, Y, M. You have also created the following variables under each partition:

```
main.A0 = 1
X.A1 = 5
Y.A1 = 2
M.A1 = 3
```

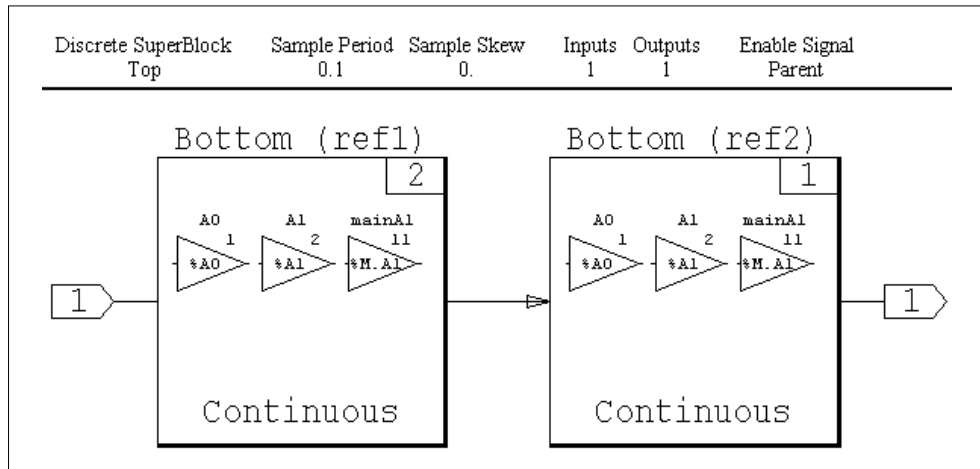


FIGURE 7-2 Example of %Variable Scoping

The model has two SuperBlocks, Top and Bottom. In the Top SuperBlock two different SB references of SuperBlock Bottom have been included. ref1 uses X as its SB reference partition and ref2 uses Y as its SB reference partition.

Inside SuperBlock Bottom there are three Gain blocks with parameter variables, A0, A1, and M.A1. Since M.A1 is explicitly specified, no hierarchical search is performed, and variable A1 under partition M is picked up during simulation. Since M.A1 has been set to 3, this block will have a gain of 3. For the gain block using variable A0, the simulator begins its search up the hierarchy since no partition is specified in the block. When it looks in ref1, it cannot find variable A0 in the X partition, likewise when it looks in ref2, it cannot find variable A0 in the Y partition. The last place the simulator looks is in the partition main, your working partition. Finally A0 is located in partition main, so the block picks up a gain of 1.

For the block with A1 specified, a gain of 5 is picked up under the X partition for SuperBlock ref1 and a gain of 2 is picked up under the Y partition for ref2.

7.3 SystemBuild Keyword Default Options

SystemBuild default values are provided for each keyword used by `sim` and related functions (`lin`, `simout`, etc.). Any option default is automatically overridden by specifying a new value in the keyword list. A simulation invoked from the SystemBuild dialog uses the same default values as the corresponding function call; selections made in the simulation dialog override any default value. To see all current default values, type:

```
SHOWSBDEFAULT
```

You can use `SETSBDEFAULT` to alter the default value of any keyword. The new value will be in effect until you change it or exit SystemBuild. For example,

```
SETSBDEFAULT, {autosavefile="autosave.cat",autosavetime=330}
```

This will set the selected keyword value as a default for your current Xmath session. Note that the comma after `SETSBDEFAULT` is required.

You can use a null string to reset a keyword that takes a string variable as an argument. See [Example 7-3](#).

EXAMPLE 7-3: Reset a Keyword that Takes a String Argument

Enable minmax logging and set the default minmax dataset name to foo:

```
setsbdefault,{minmax = "foo"}
```

Disable minmax logging:

```
setsbdefault,{minmax = ""}
```

7.4 Operating System Command Line Simulation Interface

Simulation from the Operating System (OS) command line, like the Xmath command line interface, accepts the SystemBuild model, its inputs and any simulation options as arguments. There is a direct one-to-one correspondence between the key-

words of the Xmath `sim` function and OS command line options. Additionally, you must specify the SystemBuild model file that contains your system.

Simulating from the OS command line gives you the benefit of working with OS command scripting languages such as C shell or PERL. This gives you added power to pipe files to other processes, redirect output to files, run automated simulations, and more.

sbsim Syntax

The syntax of the `sbsim` executable is discussed here. If you need assistance with the command's syntax or available options while working with the simulator, you can get help from the OS command line:

sbsim -help

Help for `sbsim` is only available from the OS. The syntax for the `sbsim` executable is as follows:

```
sbsim [-option] [argument] top modelFile
```

EXAMPLE 7-4: Using `sbsim`

To simulate a model called `MySys` located in the model file `mysys.dat` and to supply inputs `t` and `u` from the input file `mysys.in` and output the results in `mysys.out`, the command for invoking this simulation would be:

```
sbsim -i mysys.in -o mysys.out MySys mysys.dat
```

The default format for the output file is `MATRIXX` binary. To output your data in ASCII format, invoke the simulation as follows:

```
sbsim -i mysys.in -o mysys.out -fsave 1 MySys mysys.dat
```

NOTE: The input file must be in `MATRIXX` saved format (whether it be binary or ASCII) and contain a time vector `t`, and, if external inputs are required, an input data matrix, `u`.

To avoid retyping your options each time you invoke the simulator, you can use the `-opt` keyword, which takes as an argument a file that lists all the options for a particular simulation. The `sbsim` syntax for this option is:

```
sbsim -opt optionFile top modelFile
```

In the option file, each option and its argument must be listed on a separate line. Below is a sample options file which specifies the input and output files, the output file format, and the QuickSim integration algorithm.

```
-i mysystem.in
-o mysystem.out
-fsave 1
-ialg 8
```

If an option is listed twice, as in the following example, only the last option encountered is used by the simulator:

```
-i mysystem.in
-o mysystem.out
-fsave 1
-i myothersystem.in
-ialg 8
```

The input file `myothersystem.in` would be used in this case.

7.5 Analyze Function

The `analyze` function lets you query the system SuperBlock hierarchy, the system's parameter information, and any system errors. This information is useful for documenting system characteristics, but more importantly is an essential tool for debugging your models before simulation. The `analyze` function returns a list with all the names and numbers of inputs, outputs, and states of the system. It displays this list and a map of the SuperBlock hierarchy.

The `analyze` function is automatically invoked by the simulator, but can also be directly invoked from the Xmath command line or the Build pulldown.

7.5.1 Analyze from the Xmath Command Area

The syntax for the `analyze` function is as follows:

```
sbInfo = analyze("model", {keywords})
subsysInfo=analyze("model", {keywords, subsystem})
```

When the subsystem keyword is not present, `sbInfo` is an Xmath list object containing the following information:

```
SBInfo(1)      Number of inputs
SBInfo(2)      Number of outputs
```

<code>SBInfo(3)</code>	Number of implicit outputs
<code>SBInfo(4)</code>	Number of states
<code>SBInfo(5)</code>	Number of implicit states
<code>SBInfo(6)</code>	Names of inputs
<code>SBInfo(7)</code>	Names of outputs
<code>SBInfo(8)</code>	Names of implicit outputs
<code>SBInfo(9)</code>	Names of states
<code>SBInfo(10)</code>	Names of implicit states
<code>SBInfo(11)</code>	System attributes: continuous, discrete, hybrid, or multirate
<code>SBInfo(12)</code>	Rates of subsystems, ordered from slowest to fastest

When `subsystem` is used, the list object returns the following information:

<code>subsysInfo(1)</code>	<code>RateArray</code> .The rate array lists the subsystem the SuperBlock has been assigned to. All continuous SuperBlocks are given the value 0 and all DataStores are given the value -1.
<code>subsysInfo(2)</code>	Parent Index. The parent index contains the index for the name of the Parent SuperBlock in the SuperBlock name array (<code>subsysInfo(4)</code>).
<code>subsysInfo(3)</code>	Block IDs.The block ID array contains the block ID of the SuperBlock reference within the Parent SuperBlock.
<code>subsysInfo(4)</code>	Names Array.The names array contains the name(s) of every SuperBlock in the model.

In Xmath, you can index into the list object to get specific information. For example:

`sbInfo(11)`

```
ans (a string) = hybrid multirate
```

analyze outputs are stored on the Xmath stack. and the following information is shown in the Xmath Command Window message area by default:

- The SuperBlock Reference Map, including:
 - Subsystem Number (continuous = 0, 1 = fastest discrete, 2 = second fastest discrete, etc.)
 - SuperBlock Names
 - Library Number, if any
- Number of inputs

- Input names
- Number of states
- State names
- Number of outputs
- Output names
- Sampling rate status, including all rates for multirate and hybrid systems

If the analyze keyword `silent` is used, analyze generates the output list, but will not display system inputs, outputs, states, and block names.

The analyze keywords `delaybuf`, `vars`, and `typecheck` are the same as corresponding `sim` keywords. See the online help.

The `sbdefaults` keywords also apply to analyze. For instance, if you want type checking to be on at all times, then you can set it by typing:

```
SETSBDEFAULT, {typecheck = 1}
```

7.5.2 Analyze from the SystemBuild Editor

The SystemBuild Analyze SuperBlock dialog is available from the SuperBlock editor. Select `Tools`→`Analyze`. See [Figure 7-3](#) for this dialog.

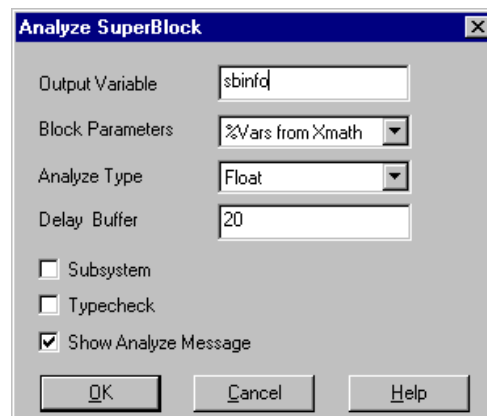


FIGURE 7-3 Analyze SuperBlock Dialog Box

For a complete explanation of the Analyze tool, open the tool and press the Help button.

7.5.3 Automatic Analyze

Since there can be no system errors in your model before you run a sim, the simulator runs `analyze` automatically on newly edited models. Automatic `analyze` provides you with the `analyze` outputs, system errors and the SuperBlock hierarchy, but not the system parameter information. If you wish to view your system's parameter information, the `analyze` function must be explicitly executed from the SystemBuild editor Tools menu, or from the Xmath command area.

7.6 Algebraic Loops

An algebraic loop occurs in a block diagram when an input to a block depends on one of the outputs of the block from the *current* cycle. This is illustrated in [Figure 7-4](#), where the input to the gain of 2 block is dependent on its output. This results in a situation where the simulator cannot readily decide which block to evaluate first.

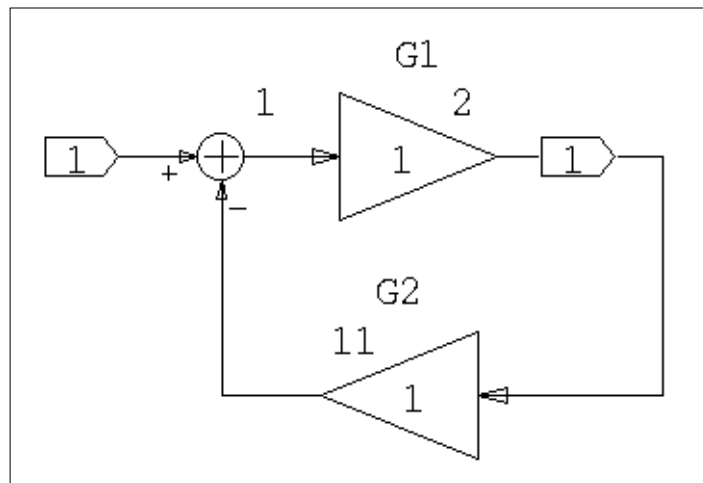


FIGURE 7-4 Algebraic Loops Computation Problem

There are several practical problems with the presence of algebraic loops; for example, the function evaluation provided by most algebraic solvers may not give a fine enough resolution for a solution to be reached; or a signal may be needed for initialization of the loop before the signal is generated. For most systems, the use of a default value for initial states (zero, for example) is usually inappropriate.

A simple continuous-time example, [Figure 7-4 on page 7-13](#), illustrates some of these problems. The following system is entirely algebraic, consisting of a summer and two gain blocks. The transfer function for this system is:

$$\frac{G1}{1 + G1 \times G2}$$

This is equal to 2/7. If one were to propagate the input several times around the loop, trying to obtain the solution, the process would prove to be unstable:

C=0 to start.

```
A = 1 + C = 1
B = 2 * A = 2
C = 3 * B = 6 (6 disagrees with 0, do another pass)
```

Pass No. 2:

```
A = 1 + C = 7
B = 2 * A = 14
C = 3 * B = 42 (42 disagrees with 6, do another pass)
```

The numbers just get bigger until the ---FIXUP OVERFLOW--- message occurs. The starting and finishing values for C will never agree, and the system runs away.

The solution to this particular example lies in selecting an integration algorithm designed to solve algebraic loops: the implicit Stiff System Solver, DASSL, or ODASSL (which works the same, for over-determined systems).

Other problems with algebraic loops occur when the SystemBuild simulator has difficulty deciding where in a loop to start its processing. As shown in [Figure 7-5 on page 7-15](#), the UserCode block (UCB) is a block that accepts a number of inputs and generates corresponding outputs. On initialization, however, the situation becomes more problematic. For example, if UCB_1 has no direct (or feedthrough) terms, then it may be necessary for the system to evaluate the outputs of the UCB before it evaluates either SuperBlock_1 or SuperBlock_2. If so, you will probably need to furnish initial states for the UCB. On the other hand, if there *are* direct terms in the UCB, SuperBlock_1 may need to be executed first, to supply the UCB inputs to process. If so, SystemBuild may have difficulty determining which block to process first. One

way to help is to set as many initial states as possible in any system with an algebraic loop, to help the system condition its calculations on start-up.

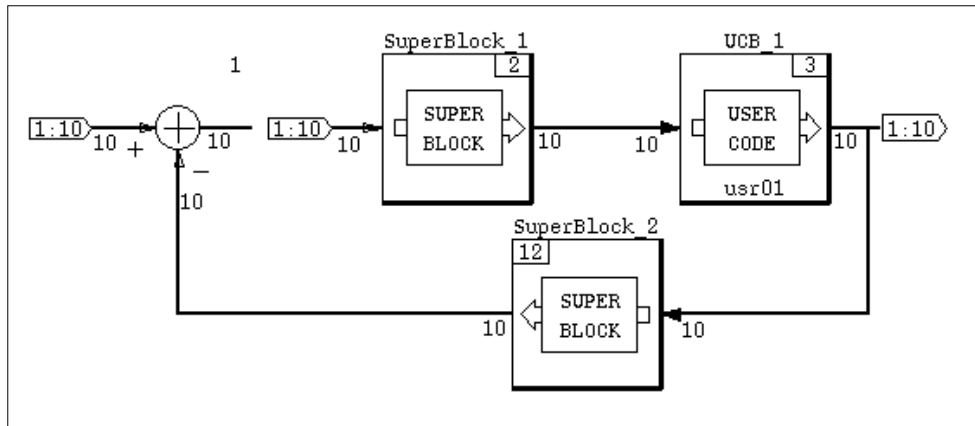


FIGURE 7-5 Algebraic Loops Initialization Problem

DASSL is the method of choice when a system has algebraic loops, since an operating point is computed for the loop instead of adding a delay. This allows the true system model to be integrated, and avoids any adverse effects a delay may cause in the system. DASSL has a built-in method for changing the error norm used for the computation of its local error test. See [Section 11.7.2 on page 11-32](#) for more on DASSL.

Connecting one of the outputs of a SIMO block to one of its inputs[†] will result in an algebraic loop being detected. See [Figure 7-6 on page 7-16](#) for an illustration of this idea. The blocks at **A** in the figure are not the same as the blocks at **B**, because the simulator tries to execute both parts of the SIMO block (serial number 1) at the same time, and cannot. Using the default integration algorithm in SystemBuild, a

[†] They will have to be connected through other blocks because the SystemBuild Editor won't let a block's output be connected to one of its inputs. The algebraic loop condition won't occur if one of the blocks presents a computational delay.

delay will be inserted in the loop, and the output of the blocks at **A** lags the outputs at **B** by one cycle.

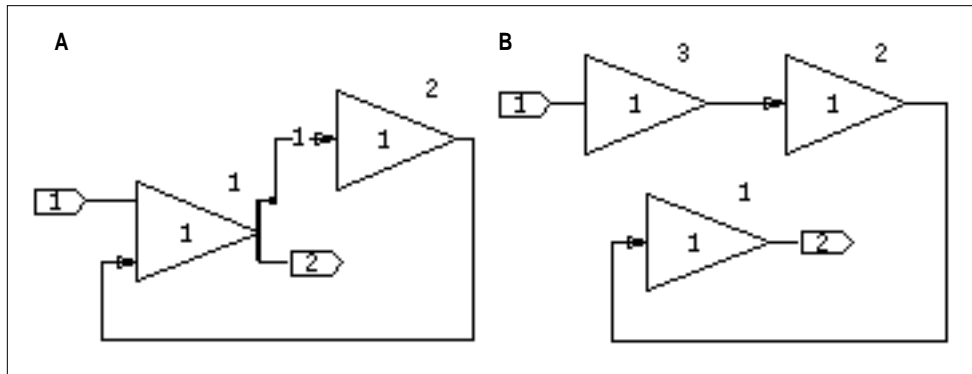


FIGURE 7-6 MIMO Blocks Example

Finally, if you wish to insert initial conditions for the algebraic loops reported by SystemBuild (the ordering is indicated in the warning message), you can use the option `yimp0` in the `sim`, `simout`, and `lin` functions. When you do this, the operating point computation that calculates the algebraic loop values uses the initial condition `yimp0` as its starting value. In order to skip this operating point computation, use the option `{initmode=4}`. When the operating point computation is skipped, it is the user's responsibility to provide consistent values for `yimp0`.

7.7 lin Function

The `lin` function performs linearization of a continuous, discrete, or hybrid system. Both explicit and implicit forms of linearization are supported.

The forms of the command are:

```
sys = lin("model", {keywords})
list = lin("model", {implicit, keywords})
```

For a complete treatment of `lin` type help `lin` in the Xmath command area, or see [Chapter 9](#). For the impact of fixed-point arithmetic on linearization, see [Section 15.6 on page 15-44](#).

7.8 simout Function

`Simout` can extract the dynamic state values (x), rates (xd), outputs (y), and continuous implicit outputs (y_{imp}) from a SystemBuild simulation, either at the initial time or at the end of a simulation. If any of the values are extracted at the initial time, this represents the starting operating point of the system, taking only the initial conditions into account. If values are extracted after a simulation, they represent a snapshot of the system's operating point at the completion of the simulation.

To run `simout` from the Xmath command prompt, enter the following:

```
[x,xd,y,yimp] = simout(model, {keywords})
```

where `model` is a string that is the name of a top level SuperBlock in the current catalog. All `simout` keywords are identical to their `sim` keyword counterparts. As with the `analyze` function, the `sbdefaults` apply. For a complete explanation of this function, go to the Xmath command area and type `help simout`.

7

Simout-specific Keywords for Initial Condition

- u0** Real vector of initial inputs. The default is zero.
- x0** Real vector of initial states. The default is the SystemBuild Catalog Value.
- xd0** Real vector of initial state derivatives in implicit User Code Blocks. Note that the meaning (and most likely the dimension) of `xd0` is different from `xd` in the output argument list.
- yimp0** Real vector of initial implicit outputs (algebraic loops).

Outputs

One, two, three, or four outputs may be requested. If multiple outputs are requested, they must appear in square brackets ({}).

- x** The state vector.
- xd** The state derivative vector. For discrete subsystems, this is a pseudo-rate obtained from the equation

$$xd = [x(k+1) - x(k)] / tsamp$$
 where `tsamp` is the sampling period of this discrete subsystem, `y` = the output values vector, and `yimp` = algebraic loop or implicit output values vector.

If your model includes Padé States, these are appended to the `x` and `xd` vectors, and thus `x` and `xd` cannot be directly used in the next simulation.

For the impact of fixed-point arithmetic on `simout`, see [Section 15.6.2 on page 15-44](#).

7.9 creatertf Command

The `creatertf` command creates a Real-Time File (RTF) for use by AutoCode, DocumentIt, or other products. The RTF is an intermediate form of the model file. The syntax of `creatertf` is as follows:

```
CREATERTF "model", {rtf, delaybuf, vars, typecheck}
```

`model` is a text string representing the name of a SuperBlock in the SystemBuild catalog. For a full description of `creatertf` syntax, go to the Xmath command area and type `help creatertf`.

7.10 Selecting an Integration Algorithm

Dynamic models created in SystemBuild can be broadly categorized as follows:

- Continuous
- Discrete, including discrete free-running, enabled, triggered, and/or procedure subsystems.
- Hybrid (i.e., a combination of continuous and discrete subsystems)

The problem of “simulating” the SystemBuild model, or obtaining a sequence of solutions to the system equations given the user-defined initial conditions and input vector, is fairly straightforward for discrete systems. Starting from the given initial conditions, the discrete state equations are iterated until the specified final time.

Finding a numerical solution for continuous and hybrid systems, on the other hand, requires a proper method of approximation. The purpose of an “integration algorithm” or differential equation solver is to calculate an accurate approximation to the exact solution of the differential equation. Then the solution is “marched” forward from a starting time and a given set of initial conditions.

Since all continuous system integration algorithms are inherently approximations, there are a number of important points to consider in selecting a proper method: computational efficiency, truncation and round-off errors, accuracy and reliability of the solution, and stability of the integration algorithm.

7.10.1 Integration Algorithms

[Table 7-1](#) lists the supported integration algorithms. The numbers correspond to the selection indices used in Xmath and SystemBuild to specify an algorithm:

1. Euler's method
2. Second-Order Runge-Kutta
3. Fourth-Order Runge-Kutta
4. Fixed-Step Kutta-Merson
5. Variable-Step Kutta-Merson
6. Differential-Algebraic Stiff System Solver (DASSL)
7. Variable-Step Adams-Bashforth-Moulton
8. QuickSim
9. ODASSL
10. Gear's method

The default integration algorithm is 5 (Variable Kutta Merson). The algorithm may be set globally using the command:

```
SETSBDEFAULT, {ialg=alnumber}
```

`alnumber` is taken from the list above. You can also determine the current default number using the command:

```
SHOWSBDEFAULT
```

To set the integration algorithm for a given simulation run, see the simulation dialog Parameters tab, or set the `ialg` keyword in the `sim` function call:

```
y = sim(model,t,u,{ialg = alnumber})
```

7.10.2 Integration Algorithm Recommendations

[Table 7-1 on page 7-20](#) lists recommendations for choosing an integration algorithm. Note that a great variety of systems will fall into more than one category listed in the table. In choosing an algorithm, therefore, it is advisable to try more than one method for those systems that belong to more than one class. These topics are covered in detail in [Chapter 11](#).

When algebraic loops are present in a model, all methods other than the Stiff System Solvers, DASSL and ODASSL, will introduce a delay into the system, even though they may integrate the equations successfully.

TABLE 7-1 Selecting an Integration Algorithm

Problem Type	Euler	RK2	RK4	FKM	VKM	DASSL, ODAS, GEAR	ABM	Quick Sim
Linear, non-stiff	+	+	++	++	+++	+	+++	
Linear, stiff					+	+++	+	+++
Nonlinear with continuous derivatives			+	+	++	+	+++ fastest	
Nonlinear with discontinuous derivatives	+	+	+	+	+++ **	+*		
Nonlinear, stiff					+	+++	+	
Systems with algebraic loops						+++		
Hybrid				+	+++	+	+	
Cont. with switch				+++				
Systems with UCBs			++	+++	++			
Differential Algebraic systems						+++		
ODAEs						+++ (ODAS, GEAR)		
Key:	<p> + Marginally suitable. ++ Very suitable. +++ Best for this problem type. * Only with state events modeling the discontinuities. ** Recommended with state events/appropriate dtmin option. </p>							

7.11 Simulation Errors

There are several categories of simulation errors, including those trapped by the hardware or operating system, the SystemBuild analyzer, or the simulator. The following errors are trapped and posted by the simulation software, and the type of block may be indicated.

7.11.1 Simulation Software Errors

`sim_ERROR: Division by 0.0 produces infinity.`

If the second input vector to a divide block contains a zero value, then this simulation error occurs.

`sim_ERROR: Raise 0.0 to a nonpositive power.`

A simulation error occurs when the input to an exponential block is zero and the constant power is less than or equal to zero.

`sim_ERROR: Both arguments to ATAN2 are zero.`

The output of the arctangent function is undefined when both inputs are zero.

`sim_ERROR: ASIN or ACOS argument out of range.`

The input to the arcsine or arccosine function block must be in the range -1 to +1. The output of this function is in the range 0 to π .

`sim_ERROR: Natural log of zero or negative number.`

A simulation error occurs if any input to the log block is less than or equal to zero.

`sim_ERROR: Square root of negative number.`

A simulation error occurs if any input to the square root block is negative.

`sim_ERROR: Raise negative number to noninteger.`

A simulation error occurs when the input to an exponential block represents a floating point power and the constant is less than zero.

`sim_ERROR: Overflow in $y = \text{EXP}(u)$ function.`

Quantity out of range of hardware.

7.11.2 Hardware Errors

Although the Simulation software catches all the errors that it can, errors that cannot be checked for in advance are trapped by the hardware and posted by the Simulation software.

```
--- Fixup Overflow ---
```

A floating point overflow occurred, and the software tried to compensate by substituting the largest possible real number. Simulation will proceed, although the results may be suspect.

```
--- FORMAT CONVERSION ERROR ---
```

```
--- FLOATING DIVIDE BY ZERO ---
```

```
--- INTEGER DIVIDE BY ZERO ---
```

```
--- SIGNIFICANCE LOST IN MATH LIB ---
```

```
--- MATH LIBRARY OVERFLOW ---
```

```
--- INVALID ARGUMENT TO MATH LIBRARY ---
```

```
--- LOGARITHM OF ZERO OR NEGATIVE VALUE ---
```

```
--- UNDEFINED EXPONENTIATION 0.**0 ---
```

```
--- FLOATING OVERFLOW ---
```

```
--- INTEGER OVERFLOW ---
```

7.11.3 Operating System Errors

The following errors are also caught by the operating system, and report an I/O error or other catastrophic system failure. If any of them occurs, contact your ISI representative.

```
--- OPEN OR DEVICE ERROR ---
```

```
--- INTERNAL IO ERROR ---
```

```
--- REWIND ERROR ---
```

```
--- RESERVED OPERAND ERROR ---
```

7.12 Scheduler

Different scheduler programs are used for simulation and for execution of generated code. The SystemBuild scheduler simulator controls the overall flow of data between subsystems, scheduling of the subsystems and posting their outputs. The generated code scheduler is optimized for real-time operations, whereas the simulation scheduler is optimized for performance in simulation. Also, the `actiming` keyword is provided for matching AutoCode's scheduling of discrete models.

7.12.1 Subsystems

By default, subsystems are determined internally by the simulator and require no intervention by the user. You can assign discrete and trigger SuperBlocks to subsystems using the processor Group ID field located on the SuperBlock Properties Attributes tab. Either way, an understanding of how they function is helpful in the interpretation of simulation results. As discussed in Sections 5.7 and 5.9 subsystems in the SystemBuild simulator are groupings of one or more SuperBlocks that fall into one of five categories:

Continuous	Integrated over each time interval in the simulation.
Free-Running Periodic	Executed repetitively at a fixed frequency.
Enabled Periodic	Executed repetitively, but only while its enabling signal remains active.
Triggered	Executed when its trigger is detected.
Procedure	Executed when its parent SuperBlock is active.

The following is an example of how a hybrid (continuous and discrete) model is divided into subsystems. Consider a model with two continuous SuperBlocks, a discrete SuperBlock at a sample rate of 0.1, and two discrete SuperBlocks with a rate of 0.05. The subsystems making up this model would then include one continuous subsystem (made up of two SuperBlocks), a Free-Running Periodic subsystem (made up of one SuperBlock) sampled at 0.1 and another Free-Running Periodic subsystem (made up of two SuperBlocks) sampled at 0.05. Each of these subsystems will then accept inputs and post outputs under the control of the scheduler at the scheduler-specified times.

Processor Group ID

Consider the model described in the previous topic, but increase the number of SuperBlocks with a rate of 0.05 to 200. By default, all 200 SuperBlocks would be as-

signed to the same subsystem. As the number of SuperBlocks assigned to a single subsystem increases, problems may occur in compiling the generated code. The code for the subsystem may become simply too large for certain compilers to deal with.

Also, in another scenario, in multiprocessor applications it may be desirable to break up the generated subsystem code and divide its execution among several processors.

The processor Group ID flag is a solution to these problems. For this flag to be used, the SuperBlocks in the subsystem that is to be divided must have identical timing attributes, and the same Processor Group ID value. SuperBlocks with the same timing attributes, but different Group ID values, will be assigned to different subsystems.

The Processor Group ID field is found in the SuperBlock Attributes dialog. It is enabled for Discrete and Trigger SuperBlock types only.

Using this flag, it is possible to divide up a set of SuperBlocks with identical timing characteristics into different subsystems. Using our example, the model may be anywhere from one subsystem (all SuperBlocks assigned the same ID) to 200 subsystems (each SuperBlock assigned a unique ID) sampled at 0.05 seconds, depending on settings for the processor Group ID fields.

Subsystem Scheduling

The simulation scheduler uses the computational attributes of the subsystems to establish the execution priority. The computational attributes include the type of subsystem, and for discrete subsystems, the sample rate.

Besides determining the priority of each subsystem, a scheduler cycle or minor cycle must be calculated so that each subsystem can be scheduled at the proper time. The minor cycle is defined as the smallest sample rate of all the subsystems in a model. If the shortest interval does not divide evenly into all the sample intervals, or if there is a timing skew, a faster "pseudo-rate" is derived from the Floating Point Greatest Common Divisor (FGCD) of the sample intervals. This is the largest floating point value that can divide each member of the set of time intervals that must be serviced into an integral number of times.

See [Figure 7-7](#) for an illustration of these ideas. Subsystem 1 runs at a time interval of 0.2 units, but Subsystem 2 runs at an interval of 0.3, so that no direct divisor of the intervals is available, and the pseudo-rate of .1 is generated.

Continuous Subsystem Scheduling

For subsystem scheduling the simulation time-line is segmented according to user time points and discrete events. The continuous subsystem however, can be thought of as running at all times throughout the simulation and is not scheduled at discrete times. This subsystem rather is integrated continuously over each time interval in a piece-wise fashion.

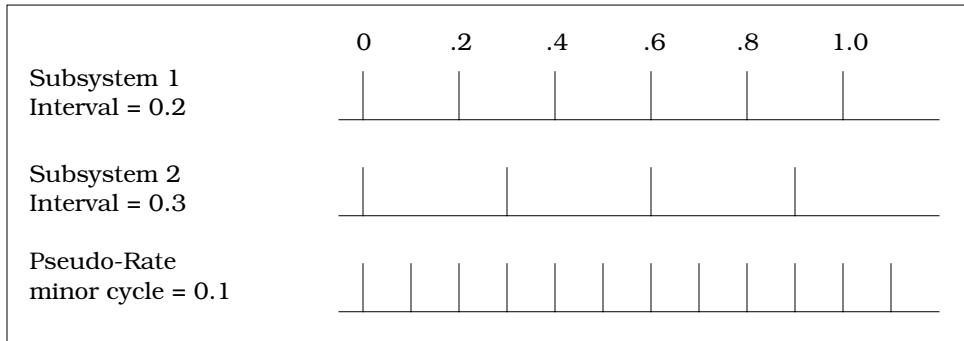


FIGURE 7-7 Derivation of a Pseudo-Rate

Discrete Subsystem Scheduling

For discrete subsystems, the scheduler is based on the principle of rate-monotonic scheduling, deriving priorities for execution from the rate of periodic subsystems and the timing requirement for triggered subsystems; the algorithm assigns higher priority to the faster subsystems and lower priority to slower ones. The priorities among the discrete subsystems are shown in the following table:

Priority	Subsystem
1	Free-running Periodic
2	Enabled Periodic
3	Triggered Asynchronous (ASYNC)
4	Triggered As-Soon-As-Finished (SAF)
5	Triggered At-Timing-Requirement (ATR)
6	Triggered At-Next-Trigger (ANT)

Properties of Discrete Scheduled Subsystems

The scheduling of execution is permanently set for each type of subsystem, but the posting of outputs can be modified with certain keywords. Posting outputs for Free-Running and Enabled Periodic subsystems can be modified with the global CDELAY (Computational Delay) keyword. When CDELAY is set, the output of discrete subsystems is delayed one minor cycle. Otherwise, the posting of the output occurs immediately following the subsystem's execution. For triggered subsystems, the posting of outputs is controlled on a SuperBlock-by-SuperBlock basis. A description of each of these posting options is given under Triggered Subsystems below.

Free-running Periodic Subsystems

A free-running subsystem is always enabled, and gets executed when its sample time has arrived.

Enabled Periodic Subsystems

An Enabled Periodic subsystem runs when it is either enabled by its parent SuperBlock or by an input signal. A SuperBlock enabled by its parent executes at its sample interval as long as its parent is enabled. A SuperBlock enabled by an input signal, on the other hand, is scheduled to execute at its sample interval as long as the enable signal is true.

Triggered Subsystems

There are four types of triggered subsystems that differ in the way they post outputs:

- | | |
|------------------------------------|--|
| At Next Trigger (ANT) | The subsystem only posts its outputs when it is next triggered for execution. ANT is used for modeling certain kinds of variable-rate but repetitive activities, such as a shaft that rotates at a variable speed. |
| At Timing Requirement (ATR) | The timing requirement is specified in the SuperBlock dialog. Outputs are posted that number of cycles after the subsystem is triggered for execution. This type of posting is a way of placing a priority on the subsystem's output availability. |

- As Soon As Finished (SAF)** The outputs are posted at the beginning of the minor cycle after the subsystem finishes running. This type of posting sets the subsystem output availability at the highest priority.
- Asynchronous (ASYNC)** The outputs are posted immediately (asynchronous to the scheduler) if the triggering signal is a state event (see [Section 11.9 on page 11-42](#)). If the triggering signal is not a state event, the outputs are posted at the beginning of the minor cycle after the subsystem finishes running (identical to SAF).

Simulation actiming Option Scheduling

The SystemBuild simulator provides the `actiming` keyword in order to match AutoCode results for discrete systems. The simulator accomplishes this by matching AutoCode's scheduler cycle, system initialization, and execution and posting times for each subsystem.

Two simulation keyword values are forced so that the initialization and posting of outputs matches AutoCode:

Keyword and Value	Description
<code>cdelay = 1</code>	The output posting is always delayed one minor cycle.
<code>initmode = 0</code>	This keyword setting disables the initialization that is normally performed at simulation time.

7.13 Simulation Time Lines, Inputs, and Outputs

7.13.1 The Input Time Line

The input time line is formed from the time vector entered via the Simulation Parameters dialog **Time Vector/Variable** field (on the **Parameters** tab) or as an input to the `sim` function. Consequently the time vector value must be a monotonically increasing column vector. The simulator uses the time vector as follows:

- The largest (and last) value in this vector is used as the simulation stop time. Changing the largest value in the input time line changes the duration of the simulation.
- Time vector time points will be synchronized with input data points (see, [Computing External Input Values](#)).

- In continuous and hybrid systems that use variable step integration algorithms, the user is guaranteed that the integration algorithm will converge on each of these time points (note that there might be other time points for which the algorithm will converge).
- Outputs from the simulation are saved for each point of the input time line (i.e. each point of the input time line is also a point on the output time line (see, [Output Time Line](#)).

7.13.2 The Internal Time Line

The simulator calculates an internal time line based on the model and the integration algorithm selected. The values in the internal time line are not known prior to the simulation.

7.13.3 Computing External Input Values

For every external input, a data point must be supplied for each time point of the input time line. The input data must have the dimension:

(Number of input time points) x (Number of external inputs).

In the Simulation dialog, this data is entered in the **Input Data/Variable** field. For the `sim` function, the input variable (traditionally named `u`) is paired with the time vector (referred to as `t`). The `sim` function allows you to specify a PDM (you cannot do this interactively). The PDM's domain is extracted for the time vector, and the range will be used for input values.

Whenever the simulation requires external inputs, it first compares the current simulation time (from the internal time line) vs. the input time line. If the current simulation time matches one of the input time points, the simulation reads the value of the external inputs directly from the input matrix.

If the current simulation time falls between two time points on the input time line, the simulator performs a linear interpolation using the known data points and assigns the resulting value to the time point (on an input by input basis).

7.13.4 The Output Time Line

The simulator saves the values of external outputs at various times during the course of the simulation. The collection of time points for which the external output values are saved is referred to as the output time line. By default the output time line includes the input time line.

The output time line is computed as follows:

- Every time point on the [input time line](#) is also on the output time line.
- If a “Reporting Period” (see, the `sim` keyword `dtout`) is specified, every integral multiple of this value is a time point on the output time line.
- If “Use Extended Time Vector” (see the `sim` keyword `extend`) is enabled, every discrete subsystem transition time point, and every discrete event time point will be added to the output time line.

8

Interactive Simulation

This chapter describes Interactive Simulation (ISIM) in SystemBuild. You can animate a simulation session by placing interactive input and display icons into your SystemBuild model; interactive simulation is helpful in debugging models. This feature contrasts with Interactive Animation (IA), an optional SystemBuild package that lets you build separate stand-alone “control panel” displays for monitoring and controlling SystemBuild simulations and the RealSim hardware interface. See [Table 8-1](#) for an illustration of the differences between ISIM and IA.

TABLE 8-1 ISIM and IA Compared

Interactive Simulation (ISIM)	Interactive Animation (IA)
Operates inside SystemBuild only. Invoked by calling <code>sim</code> with the <code>interact</code> keyword, or checking Interactive on the Simulation dialog.	Operates stand-alone only. A separate license is required.
Block icons are placed inside SystemBuild block diagrams.	A special IA Builder window is used to construct block icon diagrams.
Icons are part of the block diagram, stored with the SystemBuild model file.	Icons in separate <code>.pic</code> files, linked to SystemBuild diagrams. IA Translator converts <code>.pic</code> files to SuperBlocks for ISIM use.
Limited or optional full set of animation icons.	Full icon set only.
Provided as part of SystemBuild.	Optional.

For a complete treatment of IA, see the *Interactive Animation User's Guide*.

An example of an ISIM screen is shown in [Figure 8-1](#). In this example, you can manually change the throttle setting, brake position, and road incline while the simulation is running.

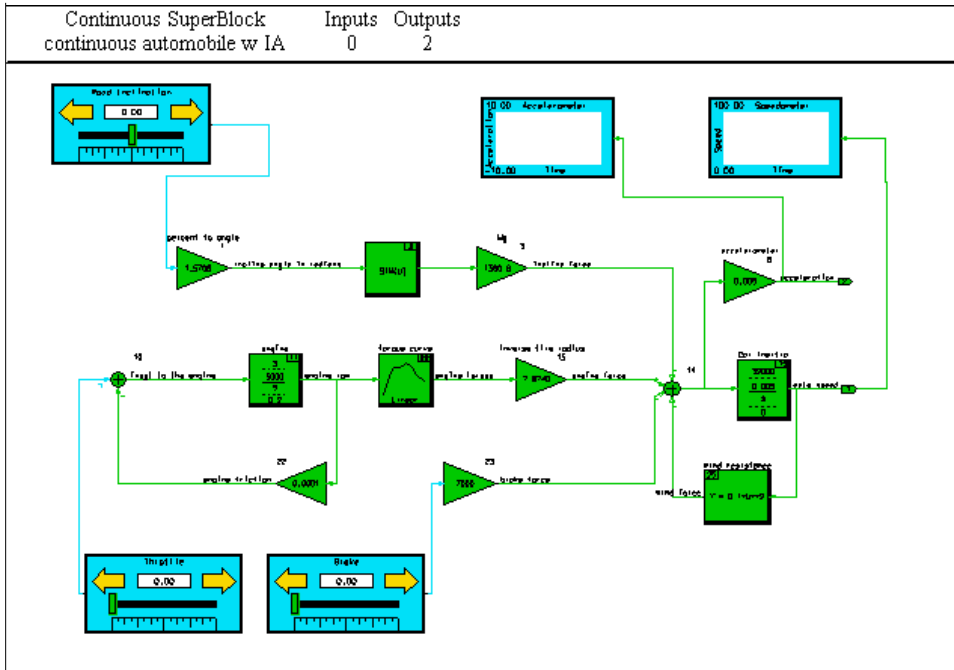


FIGURE 8-1 Diagram with Input Icons and Stripchart/Text Displays

Using ISIM, IA diagrams (which resemble control panel displays) can be built up quickly as part of a SystemBuild model. You can include IA block icons representing analog and digital control and display blocks, adjust the attributes and parameters of the block icons to fit the model's needs, and connect the icons to inputs and outputs of interest in the SystemBuild model.

Building a model that has simulation displays and controls is no different from building the rest of your model. You can select a block icon from a menu, drag it into place, define attributes via an on-screen dialog box, and connect the icon to other block icons.

When you finish your model and simulate it interactively, an ISIM display window appears. You can start, stop, restart, and resume simulation, step through blocks, view selected outputs, modify selected parameters during simulation, and other functions.

ISIM may be executed as a background task under Xmath, which allows you to enter commands in the Xmath window as an interactive simulation is running.

8.1 The Interactive Simulation Process

The IA icons are displayed when you select the *IA* button on the SuperBlock Editor toolbar. You can select an icon and drag it anywhere in your model. The icon can be connected to any SystemBuild block or blocks. The contents of the palette depend on your license; the default icon palette available to all SystemBuild users is shown in [Figure 8-2](#). The separately licensed IA palettes are discussed in the *Interactive Animation User's Guide*. Because SystemBuild is in a modal state when the IA palette is open, only one IA palette can be displayed at a time.

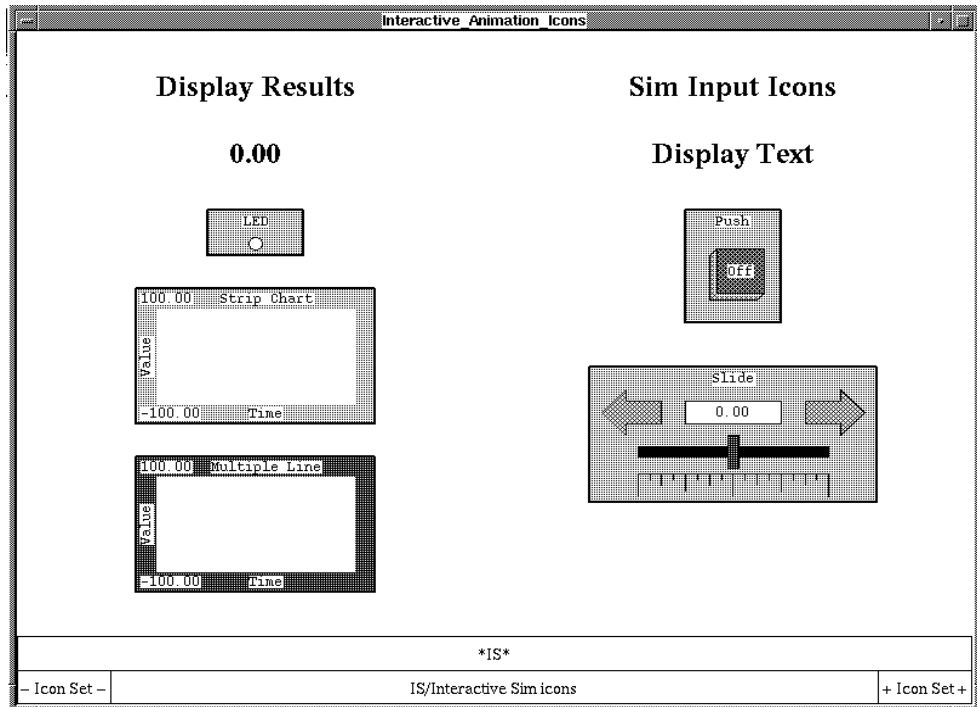


FIGURE 8-2 Default Icon Palette

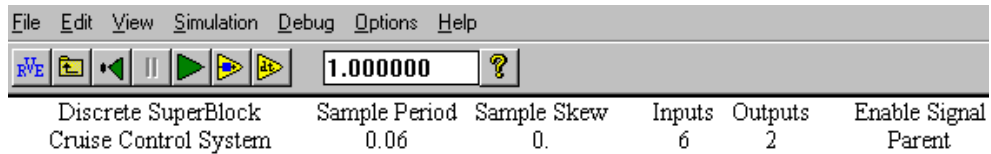
The default palette supports:

- Icons that provide inputs to SystemBuild blocks
- Icons that display SystemBuild simulation outputs

NOTE: It is not necessary to include IA icons in your model to take advantage of the other ISIM (data display, block and time stepping, RVE, etc.) However, IA icons are the most convenient way to observe or set internal simulation values while the simulation is executing.

8.2 The ISIM Window

When the interactive simulation starts the interactive simulator will be opened with the simulated SuperBlock shown. At first glance it is very like the SystemBuild editor window, but you will note that the icon tool bar immediately below the menu bar is different, as is the information displayed in the “hints” area.



ISIM options can be chosen from the tool bar or from a pulldown menu:

- Variables** Invokes Run-time Variable Editing (RVE), to allow you to change the value of a %Variable at a point in an interactive simulation. See [Section 8.7 on page 8-15](#) for a complete treatment of RVE.
- View Parent** Displays the SuperBlock containing the current instance of the currently displayed SuperBlock.
- Reset** Initialize the model to its initial conditions (from the SystemBuild catalog), with the input icon settings preserved from the previous run.
- Pause** While the simulation is running, clicking this button temporarily halts (“holds”) the simulation. You can also click in the **Hold Time** field on the tool bar and type a time (in units commensurate with the simulation t-vector) at which the simulation will pause next.
- Resume** To start a simulation, or restart a paused simulation, click on the **Resume** button. Once the simulation reaches the end of the time line, all buttons will be disabled, with the exception of the **Reset** button.

Block Step	Show order of block computations. With ISIM paused or not yet started, click repeatedly on the Block Step Button to view the sequence of block executions; the next block to fire will be highlighted. Note that if you step through an interactive simulation all the way to the end all buttons will be disabled until the return button is pressed (or you exit the simulation).
Local Block Stepping	If enabled, limits block stepping to the currently displayed Super-Blocks; the simulation will not stop on blocks outside this Super-Block. This option is only available from the Debug menu.
Time Step	Execute a single time step of the simulation. The time step is determined by the simulation time vector.
Hold Time	This allows you to specify a time for the simulation to pause. Once you start the simulation (by clicking Resume on the icon bar), it will run as long as Current Time is less than Hold Time . When the Current Time reaches the specified Hold Time, the simulation pauses. After making any desired changes to your IA icons, click Resume to start the simulation going again. For a way of using Hold Time, see Example 8-1 on page 8-18 .

8.3 Special Notes on ISIM

- A principal use of ISIM is for debugging SystemBuild designs. Attach a numeric output icon to any signals of interest (e.g., to every output of every block), and run the model in **Blockstep** mode. The outputs are updated as you proceed.
- By default, IA icons update at each input time point, but you can specify less frequent updates in the Icon dialog box, by entering a value in the **Sampling Interval** field.
- You can click on IA input icons any time the simulation is running or paused, and change any input value.
- To monitor the outputs of any block in your model, place the mouse cursor on the block and press **v**. The output labels are replaced by the current values on each of the output pins of the block. Every time you click either **Timestep** or **Blockstep** (individually) or **Resume** followed by **Pause**, the values are updated; only the labels may be displayed during **Resume**. You can monitor the outputs of any number of blocks. To turn off the feature, put the mouse cursor on the block and press **v**.
- The current simulation time is always displayed in the lower right hand corner of the ISIM window.

- You may choose global or local block step mode. If Global, the **Block Step** button will cause the interactive simulator to step to the next block that is to be executed. However, the next block could be in another SuperBlock. If Local, the **Block Step** button will resume the simulation and will pause only when the next block is the currently viewed SuperBlock. Intervening blocks that are outside this SuperBlock are executed, but do not cause the simulation to halt. The Local Block Stepping Block Debug pulldown menu.
- You can run ISIM on any SuperBlock from which simulation is available.
- You can redefine the parameters of an IA icon while the simulation is running or is paused; simply point the cursor at the icon and press Return. Values thus changed pertain to this simulation run only and are not kept.
- You can resize IA icons by placing the mouse cursor in the id area in the upper right corner of the icon and clicking and stretching with the left mouse button.
- To change the color of a icon on the screen, move the mouse cursor to the icon, then repeatedly press the ' key to cycle through the colors.
- Only one palette of IA icons is available to with the default ISIM capabilities, but users who have purchased the optional IA feature have access to seven palettes of icons. Also, full IA users can add user-written icons and add palettes of icons to the IA palettes. For more on this subject, see the *Interactive Animation User's Guide*.

8.4 sim Keywords for ISIM

The following syntaxes and keywords are provided:

To Invoke ISIM:

The general ISIM syntax is as follows:

```
sim(model_name, t_vector, ..., {interact, ...});
```

The model name and t-vector, are required. In the above syntax ... represents any optional input (such as the u_vector), and additional simulation keywords.

To Invoke ISIM for a Specific SuperBlock:

Use the sbview keyword to specify the SuperBlock name. For example, if top is the SuperBlock name, specify:

```
sim(model_name, t_vector, ..., {interact, sbview="top"});
```

The `sbview` keyword is only available from the `sim` command in Xmath; it may be discontinued in future versions.

Non-interactive Simulation with IA Blocks

In non-interactive simulation, IA display icons have no effect, while IA input icons are held at their initial values. If the `interact` keyword is not present in the `sim` call, the simulator will default to non-interactive simulation mode.

```
sim(...,{!interact}) #or
sim(...,{interact = 0})
```

To make ISIM the default, execute the command:

```
setsbdefault {interact=1}
```

Pausing ISIM at a non-zero time

```
sim(...,{interact, iahold = pausetm})
```

where `pausetm` is in the same units as the t -vector of the simulation and must be less than the simulation duration established by the t -vector.

8.5 Standard Animation Icons

As shown in [Figure 8-2 on page 8-3](#), the IA selection on the SystemBuild Toolbar provides access to seven[†] predefined icons on a single palette.

Sim Input Icons:

- An analog slider input
- A push-button switch to input yes/no binary signals

Display Icons:

- A strip chart history (output), suitable for use as an output gauge
- A bar graph output display
- Text
- A numerical output display

[†] Users who have purchased IA have access to multiple palettes of icons. Refer to the *Interactive Animation User's Guide* for details on the other palettes.

Each icon in the palette is associated with a dialog box (similar in effect to the SystemBuild block dialog box) that allows you to define the input and output parameters, display attributes, and display text definitions for each icon type. You may gain access to the icon dialog by placing the mouse cursor inside the icon and pressing **Return**. The dialog is displayed. Icon operations are similar to SystemBuild block operations with the following differences:

- The icon selection area is in the upper right-hand corner of an IA icon, just as it is with SystemBuild blocks, but is not indicated with a separate outline.
- The icon displays are designed to work in color, and up to seven colors can be used on the different fields of a given icon. On a monochrome device, the colors are mapped into shaded patterns to emulate colors. The colors are defined through the icon dialog box; to look at the color options for a field on an icon, place the pointer anywhere in the field and click the left mouse button. Click a color to select it. Most of the color fields are self-explanatory, but threshold colors are displayed on an icon only when a user-specified value is reached. This feature is convenient for generating alarms.
- **Icon Titles and Labels** allow you to replace the default text in the fields with appropriate labeling for your icons. Click on the field to select it, backspace repeatedly to delete the default contents, and type your own text.
- Values displayed by certain monitoring and control icons can appear in either a decimal or an exponential format, under control of the **Format** field. For example, if exponential format is selected, 1.0E+02 may be displayed to represent a value of 100. Click on the item in the dialog box to select and change it.
- The **Number Length** field selects the number of digits used to represent the input value. Placing the arrow in the field and clicking the left mouse button selects the field, and placing the arrow to the left of the number and clicking the left button of the mouse decreases the number of digits. Placing the arrow to the right of the number and clicking the left mouse button increases the number of digits. If the number length is too short to represent the number, a row of stars (**...**), the same in number as the number of digits, is displayed at run time.
- The **Number Decimal Places** field selects the number of decimal places to be displayed when representing the input value. 0 = no decimal point.

- **Minimum** and **Maximum Input/Output Values** display indicators (bars, lines, etc.) appear in the display area of the icon when the input value lies in the range established by the minimum and maximum values. Output icons will not generate signals outside the user-specified range.
- The **Sampling Interval** field accepts a positive integer only, to specify the update rate for the a display icon. If the user-supplied value is n , the display is updated for every n th input value that is presented. The default value is 1, which means that the icon display is updated every input cycle.

8.5.1 Strip Chart Display Icon

The Strip Chart displays one row of values input for the last 100 (or other number) time steps; it accepts only one input display vector. The majority of the parameter fields in the dialog box are self-explanatory.

Number of Bars	The number of most recent inputs is displayed. There is no arbitrary limit on the number of bars that may displayed, except for the visibility limit due to the resolution of the monitor. If you need a larger display, stretch the icon larger by clicking in the ID corner and pulling to the right, to gain space for more bars to be displayed.								
Graph Method	This field provides four different modes for plotting data: <table border="0" style="margin-left: 20px;"> <tr> <td style="vertical-align: top;">Line</td> <td>A horizontal line elevated to the current value. This is the default.</td> </tr> <tr> <td style="vertical-align: top;">Line fill</td> <td>A horizontal line elevated to the current value and color-filled below.</td> </tr> <tr> <td style="vertical-align: top;">Bar</td> <td>A solid bar of a single color for the basic graphing color or threshold color that indicates the current value.</td> </tr> <tr> <td style="vertical-align: top;">Bar Fill</td> <td>A graduated bar of a single color for the basic graphing color or threshold color that indicates the current value. This is the default.</td> </tr> </table>	Line	A horizontal line elevated to the current value. This is the default.	Line fill	A horizontal line elevated to the current value and color-filled below.	Bar	A solid bar of a single color for the basic graphing color or threshold color that indicates the current value.	Bar Fill	A graduated bar of a single color for the basic graphing color or threshold color that indicates the current value. This is the default.
Line	A horizontal line elevated to the current value. This is the default.								
Line fill	A horizontal line elevated to the current value and color-filled below.								
Bar	A solid bar of a single color for the basic graphing color or threshold color that indicates the current value.								
Bar Fill	A graduated bar of a single color for the basic graphing color or threshold color that indicates the current value. This is the default.								
Histogram	This field lets you show the values as a bar graph (Separated) or a stair-step (Not Separated). The default is Not Separated.								

8.5.2 Multiple Line (Bar Graph) Icon

The multiple line bar graph icon allows up to five rows of line or bar graphs to be plotted simultaneously.

Movement field Allows you to have the graphs grow from left to right or right to left.

Number of Bars Can be any amount; if the number is large you can stretch the icon by clicking in the ID corner and stretching it to the right.

Line or bar Graphing method may be selected.

Color selections May be made for the background, border, strips, and each plot.

8.5.3 LED Digital Monitoring Display Icon

The **Digital Monitoring LED** icon visually indicates ranges of values of a signal from the SystemBuild model. You can program an LED for two or three colors, depending on whether two or three ranges are specified for the input.

The **First LED Color**, **Second LED Color**, and **Third LED Color** fields specify the color for the three ranges of inputs to be monitored, and the **First Cutoff** and **Second Cutoff** fields specify the value(s) that will determine which LED is illuminated. If the two cutoff values are the same, the second LED color never appears; this is the method for showing only two colors.

8.5.4 Numeric Display Icon

This icon displays a numeric value, derived from an output of the SystemBuild model. The text may appear in any of the supported fonts, as illustrated in [Figure 8-3](#); use the **Text Font** field in the Dialog to select a font.

Number Length field Select the number of digits in the display to be controlled.

Format field Select between `Decimal` (default) and `Exponential`.

8.5.5 In/Out Pushbutton Switch

The **Pushbutton Switch** icon is a single-pole, single-throw switch used to input discrete signals to the SystemBuild model. The icon dialog box permits you to:

- Assign output values and colors to each switch position.
- Specify the initial position of the switch.
- Assign attributes that determine the general appearance of the icon (*i.e.*, background color, border color, text color, icon title).

8.5.6 Text Icon

The **Text** Icon lets you place text in the block diagram, up to 80 characters. The text fonts are illustrated in [Figure 8-3](#), which is an illustration taken from one of the large IA palettes. The same fonts are supported through the single text icon in the default ISIM palette, but only one font, Font 10, appears on the palette.

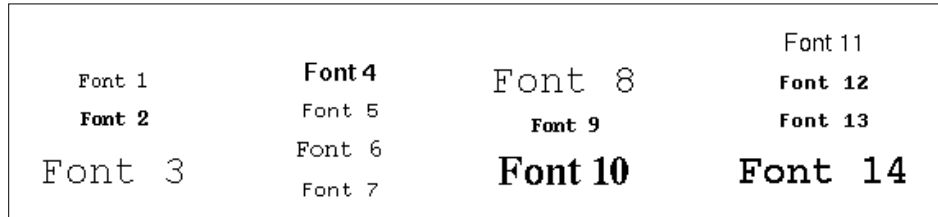


FIGURE 8-3 Text Font Choices

In the dialog box, the **Alignment** field allows the text to be centered (the default) or justified left or right about the text cursor. Click on the field to receive a menu of selections. The **Direction** field allows the text to be aligned horizontally (the default) or vertically. Click on the field to receive a menu of selections.

An important use for the Text icon is for debugging SystemBuild models. Place a test output icon at every signal of interest, and run the simulation in **Blockstep** mode. The display outputs of each block are updated each step.

8.5.7 Slide Output Controller Icon

The **Slide Output Controller** controls the inputs to your model from a display with a sliding bar that moves along a graduated scale, and has an exponential or a decimal numerical readout. The dialog box for this icon lets you specify an initial output value, minimum and maximum output values, increment and decrement values[†], and a range of values that controls the colors of the display area.

- During the execution of a simulation, clicking on either arrow of the Slide Controller icon increments or decrements the output value by the value displayed in the icon dialog box.
- Hold down the left mouse button on an arrow to get a continuous increment or decrement.

[†] Be sure to use a positive value for both the decrement and the increment.

- Middle-click on an arrow to pop up a dialog box that lets you specify a new increment or decrement value to be used in this Simulation run.
- Click any mouse button in the numeric display area to pop up a dialog box that lets you enter the exact value to the icon will output.
- The output values may also be changed by dragging the sliding bar to the desired output level, or by clicking on one of the graduations. Clicking a graduation moves the indicator to the position (and output level) marked by the graduation.

8.6 Using ISIM

Constructing an ISIM model involves little more than adding IA block icons to a SystemBuild model. Outputs to be displayed may be taken from any output pin of any block, and inputs to the model can replace any internal or external inputs of the block diagram.

The IA icons are invoked from a special palette (or set of palettes if you have purchased the extended IA icon set) that is available from the SystemBuild toolbar. Push the IA button to raise the palette, and drag an icon into the editor. To modify its parameters. Select the block and press return to raise the IA icon dialog block.

8.6.1 Building the ISIM Car Model

Load the file named `$$SYSBLD/examples/auto/cruise_d.cat`, and edit the SuperBlock continuous_automobile model shown in [Figure 8-4 on page 8-13](#).

In this model, there are several test points and control points we might want to study. At a minimum, we can add a speedometer and an accelerometer, and replace each of the external inputs with an IA signal source. To add a speedometer, proceed as follows:

1. Press the **IA** icon in the SystemBuild toolbar. The palette of IA icons appears.
2. Select the Strip Chart icon, and pull it toward the upper right of the screen. Do not worry about exact placement of the icon: you can move it later.
3. In the icon dialog box, change the **Icon Title** field to read **Speedometer**, change the **Y-axis Label** to read **Speed**, and change the **Minimum Value** to 0. Leave the **Maximum Value** at 100. Change color, then, click **DONE**.

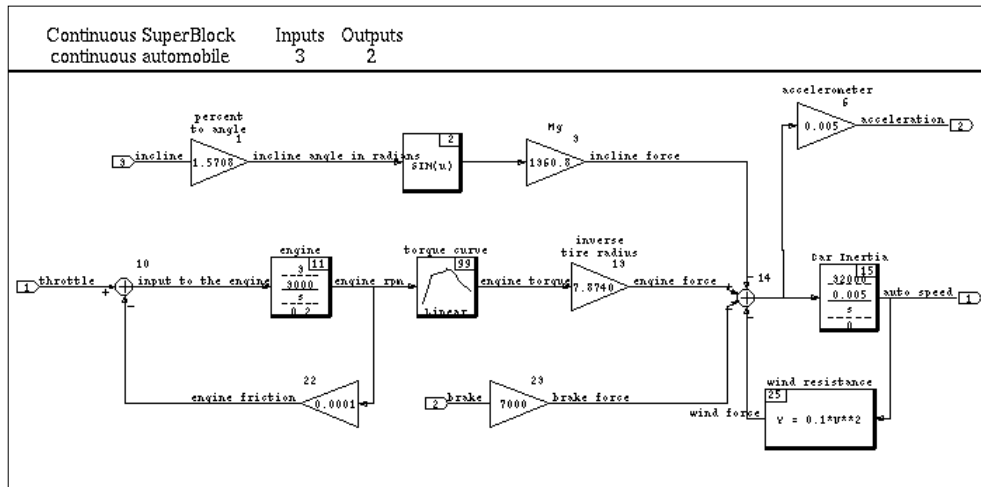


FIGURE 8-4 continuous_automobile

4. When the Speedometer icon appears on the screen, move it to the upper right side of the display. You may want to fit the diagram to the window (with the mouse cursor in open space, press **f** to make all the icons fit on the display).
5. To hook up the Speedometer, click the middle mouse button in the Car Inertia block, then in the Speedometer icon, and observe that the connection is made.
6. Add an Accelerometer. Follow the steps for the Speedometer icon, but give it the name Accelerometer, change the Y-Label to Acceleration, and change the limits to -10 and +10. Change the First Threshold to Change Color setting to 0 and the Second Threshold to Change Color to 5. Click **DONE**.
7. Move the Accelerometer to a convenient location near the top of the display. Refer to [Figure 8-1 on page 8-2](#) for a suggested arrangement of the icons. Connect the Accelerometer Gain block to the Accelerometer display icon.
8. Put in a Slide Switch to furnish a road inclination. Select the icon from the palette and give it a name, road inclination. Change the limits to -10 and +10, and change the **Negative Decrement** and the **Positive Increment** to 0.1. Click **DONE** and place the icon in the upper left part of the screen.
9. Connect the slide switch to the input side of the Percent to Angle Gain block. Click the middle mouse button in the slide switch, then in the gain block. When the Connections Editor appears, click to connect these blocks, just like any other. Click **DONE** to release the editor, and observe that the connection to the

slide switch replaces the external input to the Gain block. External inputs thus displaced still remain in the count of external inputs in the SuperBlock ID bar at the top of the screen. Remove them (from the SuperBlock Attributes dialog box) before analyzing the SuperBlock for simulation.

10. Add a brake control. Place another slide switch somewhere near the bottom middle of the screen. Give it a name, `brake`. Make the limits 0 and 1. Change the increment to 0.01 (this gives a delicate range of brake controls). Click **DONE**.
11. Connect the brake switch to the brake input Gain block and observe that the brake external input vanishes.
12. Add a throttle control near the lower left-hand side of the display. Name it `Throttle`. Change the limits to 0 and 1 and change the increment to 0.01. Click **DONE** to release the throttle control icon and connect the icon to the summing junction at the left side of the engine part of the model. Compare the completed model to [Figure 8-1 on page 8-2](#).

8.6.2 Simulating the Car Model

Now we are ready to simulate the model using ISIM. Proceed as follows:

1. First create a `t`-vector large enough to give you a little time at the wheel. Type:

```
t = [0:0.1:200]';
```

Depending on the speed of your system, the 500+ time points may not be enough to exercise the model adequately; test this parameter as necessary.

2. To run the simulation on the modified model, use the `sim` command. Type:

```
y = sim("continuous automobile", t, {interact});
```

3. You will observe a display of your ISIM model. Click the **Time Step** button to advance the simulation one step. Click the **Resume** button once to start the simulation and the **Pause** button to pause the simulation.
4. Test the controls before putting the car in motion. Click the **Time Step** button once to see the **Current Time** setting move. Move the slide switches back and forth, and observe that the changed values appear in the icons.
5. Put on your wraparound shades and driving gloves, and click the **Pause** button. Move the throttle slide forward gingerly and learn to drive all over again.
6. After you pause, select **Exit** from the **File** menu to exit simulation and return to Xmath.

8.7 Run-time Variable Editor

The Run-time Variable Editor (RVE) allows you to change %Variables and Variable blocks during the execution of an Interactive Simulation, AutoCode generated code session, or RealSim test-bed session. For ISIM operation, it operates through the **Variable** button on the ISIM toolbar. This feature differs from ISIM input icons that allow you to interact with the input channels of the primitive blocks, where RVE allows you to manipulate selected intrinsic block parameters whose values can be adjusted at simulation time. For a complete treatment of %Variables, see “Parameterization”, [Section 7.2 on page 7-5](#).

[Figure 8-5 on page 8-16](#) shows the connections of RVE in the ISIM and RealSim systems. In the ISIM context within SystemBuild, as shown in the left-hand side of [Figure 8-5](#), the RVE software is part of the ISIM program, and the user interacts with both ISIM and RVE through the ISIM toolbar. As shown in the right hand side of [Figure 8-5](#), the situation with RealSim is more complex. By default the RealSim Client Control Panel is used, which contains an RVE GUI. Also, if desired, a mechanism to access RealSim RVE from Xmath is supplied. Separate copies of the RVE software are maintained for the RealSim Client Control Panel and for Xmath, and they communicate with each other to service user Xmath requests. The RealSim Control Panel copy of RVE also communicates with the RealSim, to perform the user interface for RVE on the hardware testbed and the Xmath copy of RVE communicates with the RealSim to perform RVE script processing.

8.7.1 RVE and ISIM

This procedure explains using RVE from ISIM. The term “RVE Variable” appears throughout this discussion: it refers interchangeably to both Variable Block variables and the subset of %Variables that are supported by RVE (see [Table 8-2 on page 8-21](#).) which are treated in the same way by RVE. The procedures for AutoCode and RealSim are similar; see the *AutoCode User's Guide* and the *RealSim User's Guide* for details.

Proceed as follows:

1. Prepare your model for simulation. You must have one or more RVE Variables in the model, although whether you have ISIM icons in the model is optional.
2. Be sure your RVE Variable is copied into the Xmath data area and given an initial value. If you are working with %Variables, there are two ways to accomplish this. When you enter the %Variable, press **Ctrl-p** on the keyboard and the variable will be entered automatically. Or enter it explicitly by typing the variable name in the Xmath Command Area and setting it equal to the initial value for

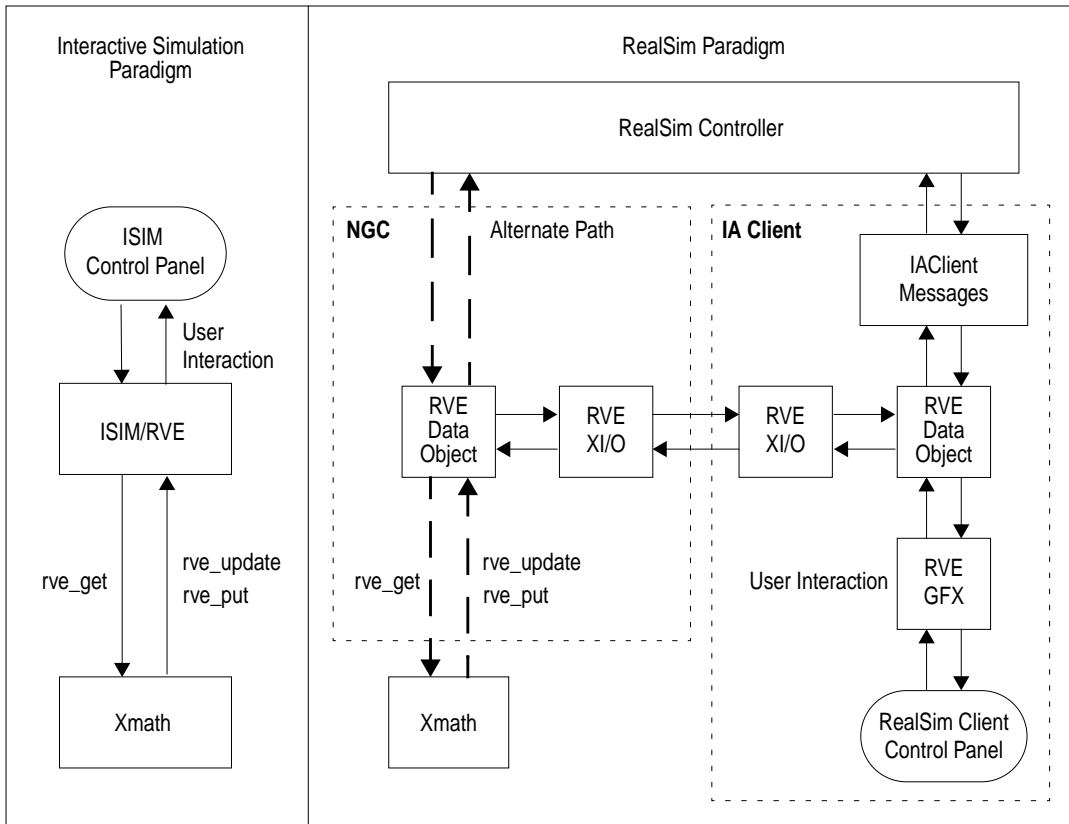


FIGURE 8-5 RVE in ISIM and RealSim Contexts

your simulation. Either way, the %Variable will acquire an initial value, as required by RVE. If you are using Variable Block variables, they are initialized by SystemBuild; if you have given them no value, the default is 0.

3. When you run the simulation, make sure that the `interact` keyword is set True. You can do this on the simulation dialog Parameters tab, or by including the `interact` keyword a `sim` command issued from Xmath.
4. Because you are simulating interactively, the Interactive Simulator window appears when you execute the `sim` command.

RVE may be invoked by selecting Edit→Variables or pressing the RVE icon on the ISIM toolbar. The simulation may be executing during the editing process, or may be paused.

5. [Figure 8-6](#) shows a typical Variable Browser dialog. To change a column width, click on a divider and drag it left or right. To edit a variable, double-click on its name, or select the name, then select Edit→Open.

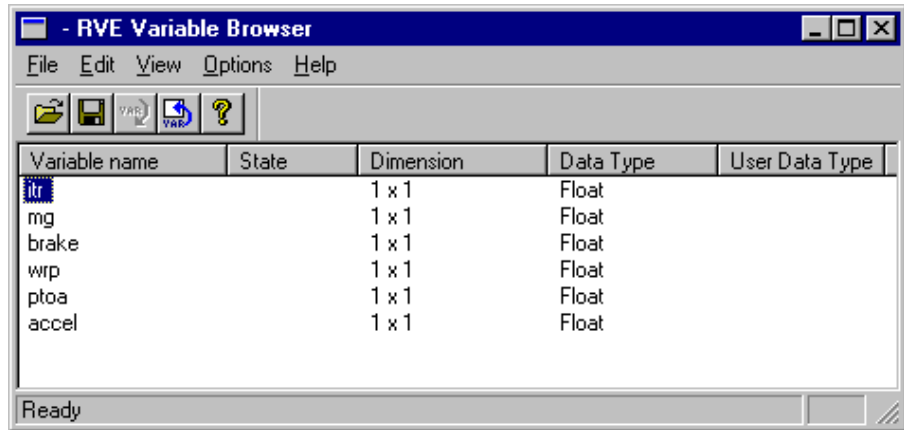


FIGURE 8-6 Run-time Variable List Dialog

6. You may add new values into the variable spreadsheet or define them from the Xmath command area. After you have edited the variable, click **OK**. This will cause the RVE browser to be displayed again. You may now select and change another RVE variable; there is no limit to the number of RVE variables you can change. When you are finished adjusting the variable values, use Edit→Select Modified to select the changed variables, then select Edit→Download or press the **Download** button on the Browser toolbar to complete the edit.
7. The new RVE variable values will be available immediately. If the simulation is running while the edit is taking place, the edit will apply to the next sim time step.

[Example 8-1 on page 8-18](#) gives a step by step example of using RVE.

EXAMPLE 8-1: Edit the Predator-Prey Model During a Simulation

In the following example, we run part of a simulation with a critical parameter at one value, then stop and change the value of that parameter. To copy the data to your current working directory, type:

```
copyfile "$SYSBLD/examples/pred_prej/pred_prej.cat"
```

1. Load the file. From the Catalog Browser, open Predator_Prey.
2. The Efficiency_factor block is parameterized via the %k parameter. To initialize this value, define it in the Xmath Command Area:

```
k = .5;
```

3. Enter values for t (time) and u (input):

```
t = [0:.1:100]';  
u = [ones(t)];
```

4. Select Tools→Simulate to raise the SystemBuild Simulation Parameters dialog. Type t in the Time Vector field, and u in the Input Data field. In the lower right portion of the dialog, enable Plot Outputs and Interactive simulation. Press **OK**.
5. When the ISIM window appears, enter 50 in the **Hold Time** field (on the icon bar) then press the **Resume** button (the green triangle). Observe that the **Time** field in the lower right hand corner of the ISIM window advances until it reaches 50, then stops.
6. Click on the **RVE** icon. In the Run-time Variable Browser, double-click on k. Change the value of k from .5 to .9. Press **OK**.
7. Back in the Run-time Variable Browser, select **k**, then press the **Download** button.
8. Back in the ISIM Main window, click **Resume**, and observe that the **Time** indicator advances to 100, then stops. Select File→Exit to return to the SystemBuild editor. The plot produced should look like [Figure 8-7 on page 8-19](#).

An examination of [Figure 8-7](#) shows how changing a parameter of a model can make significant changes in the simulation. In the left half of the plot, before time point 50 is reached, predator (below) and prey (above) populations are interacting in a certain balance. Then, at 50, where the Efficiency_factor, k, was increased, this corresponds to an increase in the efficiency of the predator, and the amplitude of the population of predators increases: at certain times in the cycle, there are more predators than ever before. This increase is at the expense of a considerable diminution in the prey population, which at best is scarcely more than half its previous

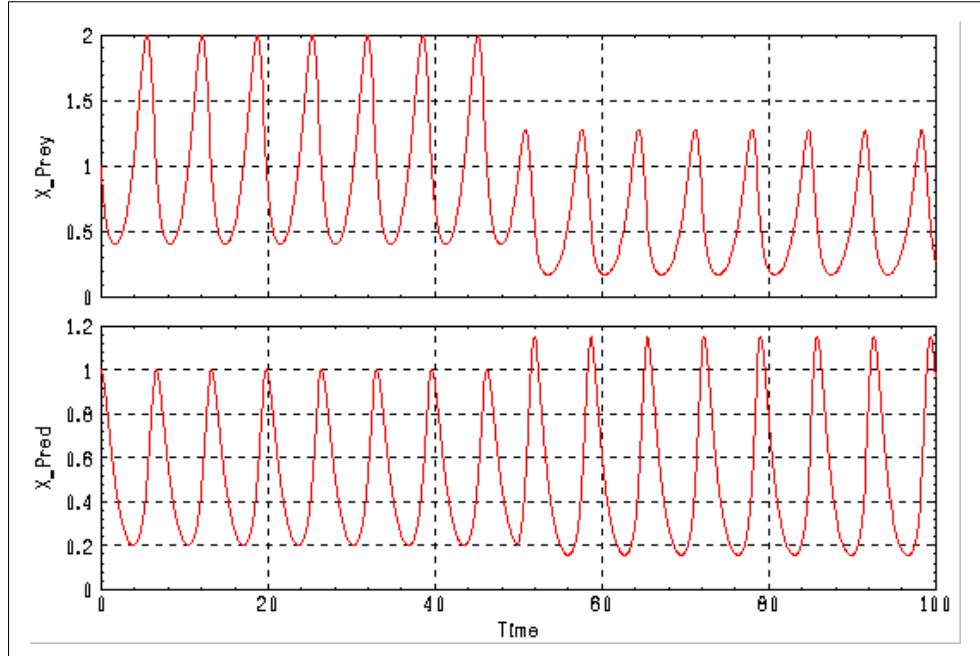


FIGURE 8-7 Plot with Run-time Variable Editing Illustrated

maximum size. And the predator population suffers, too, as can be seen at the predators' minimum population level, which is lower than it ever was in the past.

8.7.2 RVE Commands and Functions

You can operate and control RVE from Xmath. A full set of Xmath commands and functions allows you to write MathScripts that perform all the RVE functionality documented in this section. You must run ISIM in background to use these commands.

To do this, specify one of the following:

```
sim(...{bg})
setsbdefault, {bg = 1}
```

These commands are treated in more detail in the online help. To see a complete listing of the RVE commands in the online help, type `help rve` in the Xmath com-

mand area, or go to the MATRIX_x help area and select the RVE subtopic. From this point you can navigate to a complete description of the following functions.

`rve_start` Enable RVE and its commands and functions. No RVE commands or function calls can be used until an `rve_start` has been executed. `RVE_start` only works after ISIM has been started, or the `attach_realsim` function has attached Xmath to an RealSim session.

`rve_stop` Disables RVE and its commands and functions. No RVE command can be executed after an `rve_stop`, except `rve_quit` to terminate the session or `rve_start` to reenable the RVE commands.

`rve_get("var_name")`

Retrieves the current copy of an RVE Variable from the RVE workspace.

`rve_put("var_name", var_val);`

Assign the value `var_value` to RVE Variable `var_name`. This value is in an intermediate state now and is not available until an `rve_update` command is issued.

`rve_reset("var_name")`

Resets the working copy of an RVE Variable. If a variable has been modified via the `rve_put` command and has not been updated yet, the `rve_reset` command will reset the working copy of the RVE Variable to match the simulation.

`rve_update("var_list_str")`

Updates the %Variables in the simulation with the contents of the RVE workspace. The RVE workspace is modified with the command `rve_put`. All the modified fields in the RVE workspace are updated in the simulation by default. If any RVE Variables are specified, then only those variables are updated.

`rve_info` Retrieves and displays the names of all the RVE Variables that are in the model and its current status and update status. The return value, output, is binary: 1 if the operation is successful, 0 if not.

`rve_quit` Detaches Xmath from RealSim. This command is not intended for ISIM.

`attach_realsim`

Attaches this Xmath session to a RealSim session (`rtmpg`) that is currently running. The RVE Variable output will be locked while you are attached to the RealSim.

8.7.3 RVE-Compatible Blocks

Table 8-2 shows the blocks that can be used with RVE. Most %Variables on most blocks are compatible with this feature.

TABLE 8-2 Blocks Supported in RVE

Block Type	Supported Parameters	Unsupported Parameters
AlgebraicExpression	“P” parameters	Equation String, Initial Values
BiLinearInterp	Input1Points, Input2Points, OutputValues	
BlockScript	block dependent – right hand arguments only	
ConstantInterp	InputPoints, OutputValues	
ConstantPowerU	Constant(s)	
DeadBand	Deadband(s)	
Decoder	InMin, InMax	
Encoder	OutMin, OutMax	
Gain	Gain(s)	
Hysteresis	Width, Slope	InitStates
LimitedIntegrator	Upper, Lower, OutGain	InitStates
LinearInterp	InputPoints, OutputValues	
Limiter	Lower Bound(s), Upper Bound(s)	
MultiLinearInterp	Input1Points, Input2Points, Input3Points, Input4Points, Input5Points, Input6Points, Input7Points, Input8Points, OutputValues	InputLinDelta
PIDController	PGain, IGain, DGain, DTimeConst	InitStates
Polynomial	Coefficients	
Preload	Preload(s), Slope(s)	

TABLE 8-2 Blocks Supported in RVE (Continued)

Block Type	Supported Parameters	Unsupported Parameters
PulseTrain	StartTime, Magnitude, Width, Frequency	
Quantization	Resolution	
Ramp	StartTime, Slope, Limit	
ReadVariable	Variables	
Saturation	Saturation Limit(s)	
SquareWave	StartTime, Magnitude, Width, Frequency	
Step	StartTime, Magnitude	
SinWave	StartTime, Magnitude, Phase, Frequency	
UPowerConstant	Constant(s)	
Waveform	StartTime, TimeCoord, SignalCoord	
WriteVariable	Variables	

The following blocks are not supported in RVE; in most cases this is because there are no numeric values associated with the block. In some cases the simulator transforms the block to optimize its execution (as in the UniformRandom block), so that numeric entries are not available internally during simulation time:

AbsoluteValue, Acos, Asin, Atan2, AxisInverse, AxisRotation, BiCubicInterp, Break, BreakPoints, Cartesian2Polar, Cartesian2Spherical, ComplexPoleZero, Condition, Continue, Cos, CosAsin, CosAtan2, CrossProduct, CubicSplineInterp, DataPathSwitch, DataStore, DotProduct, ElementDivision, ElementProduct, Exponential, FuzzyLogic, GainScheduler, IfThenElse, implicitUserCode, Integrator, LinearInterpTable, Logarithm, LogicalExpression, LogicalOperator, MathScriptBlock, NormalRandom, NumDen, Polar2Cartesian, PoleZero, RelationalOperator, Sequencer, ShiftRegister, SignedSquareRoot, Sin, SinAtan2, Spherical2Cartesian, SpringMassDamper, SquareRoot, StateSpace, STD, Stop, Summer, SuperBlock, Text, TimeDelay, TypeConversion, UniformRandom, UserCode, While, and ZeroCrossing.

9

Linearization

9.1 Linearization

The `lin` function is used to linearize a continuous, discrete (single rate or multi-rate), or hybrid system, about an operating point. SystemBuild supports both explicit and implicit forms of linearization. For explicit linearization, a linear Xmath system object is returned; for implicit linearization, an Xmath list object is returned. Unless otherwise specified, the initial inputs are set to zero and the operating point for linearization is given by the catalog definition of the initial states.

Simple systems (purely continuous or purely discrete), where all SuperBlocks have the same computational timing attributes[†], are linearized either by evaluating exactly linearized models for the nonlinear functions, or by using finite-difference approximation. Multirate or hybrid systems are linearized using the Kalman-Bertram method. The forms of the `lin` command are:

```
sys = lin(model, {keywords})  
list = lin(model, {implicit, other keywords})
```

The required input `model` is a string specifying the name of the SystemBuild model to be linearized. Keywords are optional, except in the implicit case, where keyword `implicit` must be present. To see a full description of each keyword, go to the Xmath command area and type `help lin`.

`sys` is the Xmath System object in state-space form. `sys` incorporates the (A,B, C, D) matrices of a system into an Xmath system object.

[†] Computational attributes are defined as the timing attributes and requirements of the SuperBlock.

The implicit form of linearization is:

$$\begin{aligned} E\dot{x} &= Ax + Bu \\ y &= Fx + Cx + Du \end{aligned}$$

In the implicit form, the list output has eleven items:

```
list(1) = A
      (2) = B
      (3) = C
      (4) = D
      (5) = E
      (6) = F
      (7) = tsamp
      (8) = State Names
      (9) = Input Names
     (10) = Output Names
     (11) = Implicit Output Names
```

9.2 Linearizing Single-Rate Systems About an Initial Operating Point

9.2.1 Continuous Systems

Continuous systems are represented by the following non-linear differential and output equations:

$$\begin{aligned} \dot{x} &= f(x, u) \\ y &= g(x, u) \end{aligned} \tag{Eq. 9-1}$$

y is the system output vector, \dot{x} is the time derivative of the state vector, x is the state vector, and u is the external input vector.

Explicit Form

The linearized system matrix in explicit form is:

$$\begin{bmatrix} \dot{x} \\ y \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix}$$

where:

$$A = \left. \frac{\partial f}{\partial x} \right|_{x_0, u_0}$$

$$B = \left. \frac{\partial f}{\partial u} \right|_{x_0, u_0}$$

$$C = \left. \frac{\partial g}{\partial x} \right|_{x_0, u_0}$$

$$D = \left. \frac{\partial g}{\partial u} \right|_{x_0, u_0}$$

Implicit Form

The implicit linearization form assumes the system is in Differential-Algebraic form:

$$(0 = f(\dot{x}, x, u))$$

$$(y = g(\dot{x}, x, u))$$

and the linearization becomes:

$$\begin{aligned} [E\dot{x} &= Ax + Bu] \\ (y &= F\dot{x} + Cx + Du) \end{aligned}$$

where:

$$E = - \left. \frac{\partial}{\partial \dot{x}}(f) \right|_{\dot{x}_0, x_0, u_0} ; A = \left. \frac{\partial f}{\partial x} \right|_{\dot{x}_0, x_0, u_0} ; B = \left. \frac{\partial f}{\partial u} \right|_{\dot{x}_0, x_0, u_0} \quad \text{Eq. 9-2}$$

$$F = \left. \frac{\partial g}{\partial \dot{x}} \right|_{\dot{x}_0, x_0, u_0} ; C = \left. \frac{\partial g}{\partial x} \right|_{\dot{x}_0, x_0, u_0} ; D = \left. \frac{\partial g}{\partial u} \right|_{\dot{x}_0, x_0, u_0} \quad \text{Eq. 9-3}$$

- Time is assumed to be zero.
- For continuous systems, algebraic loops are “resolved” (i.e., `lin` always computes consistent values for algebraic loops, if any, except when the `implicit` keyword is used). For discrete subsystems algebraic loops are resolved by default, but if you do not want algebraic loops to be resolved, specify `alglloop = 0`. Note that algebraic loop delays will appear as additional states when this is done.

9.2.2 Discrete Systems

Each discrete subsystem is represented by the following difference equations:

$$\begin{aligned}
 x_{k+1} &= f(x_k, u_k, z_{k+1}) & X_k &= f(X_k, U_k, Z_k) \\
 z_{k+1} &= h(x_k, u_k, z_{k+1}) & Z_k &= h(X_k, U_k, Z_k) \\
 y_k &= g(x_k, u_k, z_{k+1}) & Y_k &= g(X_k, U_k, Z_k)
 \end{aligned}
 \tag{EQ. 9-4}$$

z represents the algebraic loop variables (if any) in the model. When `alglloop = 1` (the default case), the variable z_k is eliminated by a Newton-Raphson root-solving method, and the equations are reduced to:

$$\begin{aligned}
 X_{k+1} &= f(x_k, u_k) \\
 y_k &= g(x_k, u_k)
 \end{aligned}$$

After this step, the explicit equations are linearized. When `alglloop = 0`, the variable z_k is not eliminated; instead a *delay* is added to the z_k term such that it becomes:

$$z_k = h(X_k, U_k, Z_{k-1})$$

Thus, a new state vector is obtained by appending x_k and z_{k-1} . If this new state is represented as X , then, the equations become, as before, of the form:

$$\begin{aligned}
 X_{k+1} &= f(X_k, u_k) \\
 y_k &= g(X_k, u_k)
 \end{aligned}$$

As shown, f is actually going to change when we do the math, and the linearization is performed on these equations. As a result, *every* algebraic loop reported by SystemBuild becomes a new state in this linearization.

9.3 Exact vs. Finite Difference Linearization

Many SystemBuild blocks have built-in exact linearizations, but the following do not:

- AlgebraicExpression
- LogicalExpression
- FuzzyLogic
- MultiLinearInterp
- UserCode

In the above blocks, when any (or all) of the perturbation vectors du , dx , $dxdot$ are specified, the linearization defaults to a central finite-difference linearization.

9.4 Special Linear Models

9.4.1 Continuous Time Delay

For `lin` the continuous time delay is modeled by a state-space representation using a Padé approximation:

$$H(s) = \frac{N(s)}{D(s)} \quad \text{with order of } N = (\text{order of } D - 1)$$

This representation approximates $e^{-s\tau}$. The order can be from 0 to 10, which is selected in the block dialog box of the time delay block.

The initial conditions for the states in the continuous time delay model are assumed to be zero; however they may be changed by the user from the dialog box. Use the `analyze` function to determine the names of the states; `analyze` returns a list object in which the state names are the 9th element:

```
SBinfo = analyze("model_name", {keywords});
snames=SBinfo(9)?
```

9.4.2 State Transition Diagrams

State Transition Diagrams (STDs) are used in a linearization only to compute the system operating point. This means that initial state values are used to determine state transition conditions and then, based on these calculations, new output values (either 0 or 1) are computed. These values form the operating point for the STD.

Thereafter, the STD is not perturbed during linearization, the linearized model is zero, and *the STD states are not included in the state-space representation*.

9.4.3 FuzzyLogic Block

The FuzzyLogic block is treated as an algebraic block in linearization. The linearized model uses finite differences, and the perturbation value can be defined in the FuzzyLogic block dialog box.

9.4.4 Integrator Block (Resettable)

The linear model for the Resettable Integrator is an n_{th} -order integrator. The resettable nature of the block is ignored during linearization.

9.4.5 UserCode Blocks

If you are writing UserCode blocks, code template files `usr01.c` and `usr01.f` are provided in the directory `$SYSBLD/src`. The function template `usr01` aids in specifying exact linearization models, See [Chapter 14](#). If the template linearization is not in `usr01`, the default linearization is by finite differences.

9.4.6 Procedure SuperBlocks Referenced from Condition Blocks

In linearization, Procedure SuperBlocks referenced from Condition blocks are treated as if they were algebraic; thus, the linearization of these blocks becomes 0.

9.5 Linearizing About a Final Operating Point

To linearize a single-rate system at a certain operating point, you must first perform:

```
y = sim(model,t,u)
sys = lin(model,{resume});
```

The `resume` keyword indicates that the linearization operating point will be the final operating point of the previous simulation. To save the operating point at the end of the simulation specify the `lin` options `resumeto=filename` and `resume-from=filename`.

Alternatively, you can simulate the model until you reach the desired operating point:

```
y = sim(model,t,u);
```

Find the state at that operating point:

```
[x] = simout(model);
```

Linearize around the operating point:

```
sys = lin(model, {u0=u(length(t),:), x0=x})
```

9.6 Multirate Linearization

MATRIX_x multirate linearization is based on the Kalman-Bertram method, which returns a single-rate discrete linear system. The states of the new single-rate system correspond to the continuous states and all the discrete states appended together. The rate of the new state space representation is called the basic time period. For multirate systems with rates that are integer multiples, the basic time period is the slowest rate in the system.

In the Kalman-Bertram method, the linearized system is computed from the state transition matrices of each subsystem. The state transition matrix is computed by perturbing the solution over the basic time period with respect to the initial conditions. The default perturbation value dx for the finite difference calculation is $dx = 0.001 * (1 + \text{abs}(x_0))$, where x_0 is the initial state vector. If x_0 is zero, the perturbation value is 0.001.

The perturbation values for multirate linearization can be defined, using the keywords `du`, `dx`, `dxdot`. The syntax is the same as in the single-rate case.

The Kalman-Bertram method requires that all samplers of discrete systems sample at the beginning and end of the basic time period, since signals in discrete systems are not defined between samples. `btptol` is a `lin` keyword with a default value 0.001. In this implementation, the basic time period starts at zero and all skews must be less than $1e-6$. The sample rates may be asynchronous, however. `btptol` must be greater than zero. Smaller values of `btptol` more closely approximate systems with rates that are integer multiples. Larger values of `btptol` allow modeling of systems with asynchronous rates.

The basic time period, then, is the first time when the samplers sample within a tolerance of each other, where the tolerance is proportional to `btptol`. Hence, for asynchronous systems, a smaller `btptol` results in a longer basic time period.

At the end of the basic time period all discrete subsystems are executed and their states computed. Suppose you specify:

```
y=sim(model,t,u); sys=lin(model,{resume})?
```

The sim must end at t , which should be a multiple of the basic time period, in order to obtain a correct multirate linearization result.

9.6.1 Interpretation of Multirate lin Results

It is useful to review the theory of the Kalman-Bertram method to understand the meaning of the equivalent single-rate linear system that is obtained. For example, questions can be asked such as, in what sense are the systems equivalent? How well does the single-rate system match the response of the multirate system?

Consider a simple hybrid linear system of the form shown in [Figure 9-1](#).

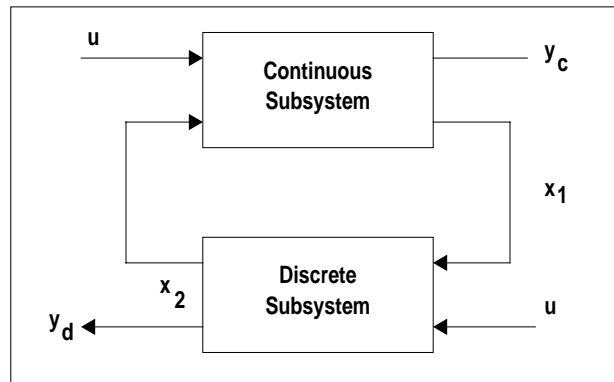


FIGURE 9-1 Hybrid Linear System

Suppose the system uses these equations:

$$\begin{aligned}\dot{x}_1 &= A_c x_1 + B_c u_1 \\ y_c &= C_c x_1 + D_c u_1 \\ x_2(k+1) &= A_d x_2(k) + B_d u_2(k) \\ y_d(k) &= C_d x_2(k) + D_d u_2(k) \\ u_1 &= B_{11} x_2 + B_{12} u \\ u_2 &= B_{21} x_1 + B_{22} u\end{aligned}$$

In this system, x_1 = vector of continuous states, and x_2 = vector of discrete states.

To apply the Kalman-Bertram method we need to compute an overall state transition matrix over the slowest rate in the system. This is done in two steps: first we compute the state transition matrix from $t = 0$ to $t = T^-$ for the continuous system, then we compute the state transition matrix for the discrete system from $t=T^-$ to $t=T^+$. The continuous state equation is computed as follows:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{u} \end{bmatrix} = \overbrace{\begin{bmatrix} A_c & B_c B_{11} & B_c B_{12} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}}^{\text{F Matrix}} \begin{bmatrix} x_1 \\ x_2 \\ u \end{bmatrix}$$

The second row of the matrix F is zero because the discrete states are assumed to be constant over the sample interval. The third row is zero because the external input is held to a constant value over the same interval. This assumption is an important simplification for the Kalman-Bertram method, implying that a zero-order hold and sampler are applied at the input to the system, and the sample interval of the sampler is the basic time period. The discrete state equation is computed as follows:

$$\begin{bmatrix} x_1(T^+) \\ x_2(T^+) \\ u(T^+) \end{bmatrix} = \overbrace{\begin{bmatrix} I & 0 & 0 \\ (B_d B_{21}) & A_d & B_d B_{22} \\ 0 & 0 & I \end{bmatrix}}^{\text{G Matrix}} \begin{bmatrix} x_1(T^-) \\ x_2(T^-) \\ u(T^-) \end{bmatrix}$$

For this part of the computation, the continuous states are held constant, as can be seen from the first row of G below. Also, as seen by the last row of G, inputs are assumed to be constant over the update of the discrete system. This means that the sampler on the external inputs is not updated until after discrete subsystems are updated.

The overall state transition matrix is:

$$\Phi = G e^{FT}$$

The single-rate system state equation is:

$$x_{K+1} = \Phi_1 x_K + \Phi_2 u_K$$

Where $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ and Φ_1, Φ_2 are submatrices of Φ .

We are now ready to discuss some aspects of this method. Because the input is held constant over the calculation of the state transition matrix, one can observe that for linear systems, the Kalman-Bertram method will give a good match to the steady state step response of the multirate system. As the frequency of the input increases, however, the match between the single-rate equivalent system and the multirate system gets worse. This makes sense intuitively because the overall response of the hybrid system must be matched by a lower-rate system. A general rule is that the input signal frequencies should be no higher than the cutoff frequency of the slowest rate system.

Another issue is that high-frequency inputs can be completely missed by the lower-rate system. For example, if a pulse is applied to a hybrid system with a continuous subsystem followed by a discrete subsystem, the continuous subsystem will respond to the pulse. However, a single-rate system can miss the pulse if it occurs between samples. Following this reasoning, the Kalman-Bertram method will not be a good impulse response matching method.

The Kalman-Bertram method can also be thought of as a discretization method using the z-transform with zero-order hold. If you are interested in matching the frequency response of the multirate and single-rate systems, The Kalman-Bertram method is not recommended because the response at higher frequencies will not be well matched.

For linear systems only, multirate `lin` results may be verified by comparing the step response obtained from the multirate `lin` results, using the `step` function:

```
sys = lin(model, {keywords})  
[t,y] = step(sys)
```

9.6.2 Linearizing Fixed-point Blocks

Fixed-point block linearization is performed the same, whether the model is single-rate or multirate. The parameters are quantized, as Fixed-point requires, then the linearization is performed in the usual manner. See [Chapter 15](#).

9.6.3 References

1. Amit, N., "Optimal Control of Multirate Digital Control Systems," Ph.D. thesis, Stanford University, SUDAAR #523, July 1980.
2. Glasson, D.P., "Development and Application of Multirate Digital Control," *Control Systems Magazine*, November 1983.

9.7 Trim

`trim` finds the trimmed input, state and output values for equilibrium points of a system. It is especially useful for finding a steady-state operating point without actually simulating the system. `trim` is also used to calculate the magnitude of the constant inputs required to keep a system at its steady-state operating point. The system can be continuous or discrete, and multirate and hybrid systems can be trimmed with this function.

The system to be trimmed must have at least one state. A system with no external inputs can be trimmed. `trim` accepts unconstrained and constrained inputs, states, and outputs as initial conditions and computes a steady-state operating point for a system. In order to obtain meaningful results, the number of constraints should not exceed the number of unknowns.

`trim` returns the states (x), the inputs (u), and outputs (y), at a steady-state operating point. If you need to verify that the solution found is indeed a steady-state operating point, call the `simout` function with x , u values obtained from the `trim` solution. The derivative vector \dot{x} calculated by `simout` should match the derivative constraint vector specified for the `trim` problem. Alternatively, you can perform a simulation using x for the initial conditions and u as constant inputs. The results of this simulation should match y .

For linear systems, `trim` usually converges in one or two iterations. For nonlinear systems, convergence to the equilibrium may take longer. You can control the total number of iterations to be performed, as well as the tolerance criterion for convergence.

Consider the following nonlinear dynamic system:

$$\begin{aligned}\dot{x} &= f(x, u) \\ y &= g(x, u)\end{aligned}$$

The trimmed operating point is defined as the state and input values for which $\dot{x} = \dot{x}_t$, where \dot{x}_t is the required derivative value at the steady-state operating point.

For the case where $\dot{x}_t = 0$, the operating point is an equilibrium point, characterized in simulation by no transients due to initial conditions.

When `trim` is applied to a discrete system, \dot{x}_t is defined as the pseudo-rate:

$$\dot{x}_t = \frac{x(k+1) - x(k)}{T}$$

9.7.1 trim Syntax

The syntax of the `trim` function is as follows:

```
[xt,ut,yt,yimpt] = trim (model,{xdt, xdt_float, u0, u_frz,
    x0, x_frz, y0, y_frz, yimp0, dx, du, iter, trimt01})
```

`model` is a required string specifying the model to be processed.

The following are optional inputs.

<code>xdt</code>	State derivative vector at which the system is to be trimmed, of dimension n_x by 1. Default = all 0.
<code>xdt_float</code>	List of integers specifying the floating (unconstrained) derivatives. The system will be trimmed at zero equilibrium by default. The constraints on derivatives corresponding to these will be removed from the <code>trim</code> equations to be solved. <code>trim</code> will usually detect “free integrators” and inform the user. In some cases, the <code>trim</code> solution may improve if indices of these free integrators are included in <code>xdt_float</code> . Default = null.
<code>u0</code>	Nominal input vector, of dimension n_u by 1 or n_u by 2. Default = all 0.
<code>u_frz</code>	List of integers specifying inputs to be frozen. Default = null.
<code>x0</code>	Nominal initial state vector; defaults are taken from the System-Build catalog.
<code>x_frz</code>	List of integers specifying states to be frozen. Default = null.
<code>y0</code>	Nominal output vector.
<code>y_frz</code>	List of integers specifying outputs to be frozen. Default = null.
<code>yimp0</code>	Nominal implicit output vector for systems with algebraic loops.
<code>dx</code>	Real (default 0.001). State perturbation vector for linearization.
<code>du</code>	Real (default 0.001). Input perturbation vector for linearization.
<code>iter</code>	Integer. Number of <code>trim</code> iterations. <code>trim</code> will continue improving the solution until either the tolerance criterion is satisfied (see <code>trimt01</code> below), or the maximum number of iterations specified with <code>iter</code> is exceeded.

9.7.2 trim Algorithm

The `trim` function is designed to find the trimmed input and state values x_t , u_t , such that the following constraints are satisfied:

$$\begin{aligned}\dot{x} &= \dot{x}_t \\ y &= y_t\end{aligned}$$

y_t represents the constrained (“frozen”) components of the output vector. To describe the algorithm we will use a continuous nonlinear system:

$$\dot{x} = f(x, u)$$

and we wish to satisfy, at the steady-state operating point,

$$\begin{aligned}\dot{x}_t - f(x, u) &= 0 \\ y_t - g_f(x, u) &= 0\end{aligned}$$

subject to x_t , u_t as specified by the user.

Linearization of the system and output equations yields:

$$\begin{aligned}\frac{\partial f}{\partial x} \Delta x + \frac{\partial f}{\partial u} \Delta u &= 0 \\ \frac{\partial g}{\partial x} \Delta x + \frac{\partial g}{\partial u} \Delta u &= 0\end{aligned}\tag{EQ. 9-5}$$

If we set, at the operating point:

$$A = \frac{\partial f}{\partial x}, \quad B = \frac{\partial f}{\partial u}, \quad C = \frac{\partial g}{\partial x}, \quad D = \frac{\partial g}{\partial u}$$

The result is:

$$\begin{bmatrix} \tilde{A} & \tilde{B} \\ \tilde{C} & \tilde{D} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix} = 0\tag{EQ. 9-6}$$

The matrices A, B, C, D are found by linearizing the system with the `lin` command. \tilde{A} , \tilde{B} , \tilde{C} , and \tilde{D} are obtained by deleting rows and columns from A, B, C, and D according to floating derivatives and frozen x, u, and y values.

In order to construct an iterative Newton-Raphson root solving problem, we rewrite Equations 9-5 and 9-6 as:

$$\begin{bmatrix} \tilde{A} & \tilde{B} \\ \tilde{C} & \tilde{D} \end{bmatrix} \begin{bmatrix} x_{i+1} - x_i \\ u_{i+1} - u_i \end{bmatrix} = \begin{bmatrix} \dot{x}_t - f(x_i, u_i) \\ y_{fi} - g_f(x_i, u_i) \end{bmatrix} \quad \text{Eq. 9-7}$$

The iterations continue until convergence is achieved:

$$\begin{bmatrix} x_{i+1} \\ u_{i+1} \end{bmatrix} \cong \begin{bmatrix} x_i \\ u_i \end{bmatrix}$$

9.7.3 trim Behavior

Stability

When a system is unstable, it may still be possible to trim the system at an unstable equilibrium point. However, this may not correspond to a steady-state operating point. When simulated, a system will diverge from such trimmed points. To check for stability, inspect the eigenvalues of the system, obtained by linearizing the system at the trimmed operating point.

Free Integrators

The `trim` algorithm may have difficulty when trimming systems with free integrators. Free integrators are dynamic states that do not contribute to the calculation of state derivatives. In the presence of free integrators, it may or may not be possible to trim a system exactly.

When `trim` detects a free integrator, it reports the index and name of the state associated with the derivative, but does not try to remove the free integrator.

If the `trim` iterations fail to converge, either:

- Some constraints on `Y` can be removed.
- The indices of free integrators reported by the `TRIM` algorithm can be included in the list `xdt_float`.

Algebraic Loops and trim

Trimming a system with algebraic loops may require that you obtain trimmed values for the algebraic loop outputs. These outputs can be obtained by adding a new term in the output list:

```
[xt, ut, yt, yimp0] = trim(model {xdt, xdt_float, u0, u_frz, x0,
    x_frz, y0, y_frz, yimp0, dx, du, iter, trimt01, message})
```

model is a string specifying the name of a SuperBlock in the SystemBuild editor.

yimp0 Nominal implicit output vector for the algebraic loop outputs.

yimp0 Trimmed algebraic loop output vector.

The trimmed state and algebraic loop outputs can be inserted as simulation or linearization initial conditions, for example:

```
y = sim(..., { x0=xt, yimp0=yimp0})
```

9.7.4 trim Examples

EXAMPLE 9-1: Simple Linear Model

To trim a simple linear model, load and analyze the F14 model Flat Model. This system has eleven states, three inputs, and four outputs, and can be copied to your working directory as follows:

```
copyfile "$SYSBLD/examples/f14/f14_2.cat"
```

Load the file. Using the following trim command, the number of unknowns is eleven (only the states are solved for; inputs are frozen). The number of constraints is 11, as seen from a simple row count in [Equation 9-7 on page 9-15](#), and so the trim problem is well-posed:

```
[xt,ut,yt]=trim("Flat Model",{u0=[2; 2; 2], u_frz=[1, 2, 3]})
```

The results of the trim operation can be verified using the following commands:

```
t=[0:10]'; u=2*[ones(t), ones(t), ones(t)];
y=sim("Flat Model",t, u, {x0=xt})
```

Note that there are no transients in the output, and the above y matches the output yt obtained from trim.

EXAMPLE 9-2: Over-Constrained Simple Linear Model

The following `trim` command gives eleven unknowns (solving only for states; inputs are frozen), and $11 + 4 = 15$ constraints (four outputs are constrained); thus the problem is over-constrained.

```
[xt,ut,yt]=trim("Flat Model", {u0=[2;2;2], u_frz=[1,2,3],
    y0=[10;20;30;10], y_frz=[1,2,3,4]})
```

For this example, `trim` returns a message indicating the size of the error, which is large because the problem as it was presented is over-constrained. The solution for x_t , u_t will give a least-square solution but cannot attain zero error.

As an example of a nonlinear `trim` problem, load and analyze the Predator-Prey model. Copy the model file to your local directory:

```
copyfile "$SYSBLD/examples/pred_prey/pred_prey.cat"
```

Load the file, then call the `trim` function.

```
[xt,ut,yt]=trim("Predator_Prey",{u0=1, x0=[2;2], u_frz=1, iter=5})
```

The steady-state operating point can be verified by simulating the system with the above `xt` as initial conditions.

Experimenting with different initial conditions reveals that this system has two equilibrium points at: $x_t = [0;0]$, $[1;0.5]$.

10

Classical Analysis

10.1 Classical Analysis Tools

This chapter describes the user interface to the SystemBuild SuperBlock Editor analysis tools. These tools provide an easy-to-use graphical interface for analysis of the current model using Xmath functions. A summary of the tools is as follows:

- The Analyze tool provides a graphical interface to the `analyze` function. Press the Help button on the dialog for information on using this tool.
- The Simulate tool is a graphical interface to the `sim` function. Press the help button on the dialog for information on using this tool.

This chapter focuses on the classical analysis tools, which are:

- Time Response
- Open-Loop Frequency Response
- Point-to-Point Frequency Response
- Root Locus
- Parameter Root Locus

Most of these tools linearize the current SuperBlock. The SystemBuild `lin` function performs classical linearization of continuous, discrete (either single rate or multi-rate), or hybrid (mixed continuous and discrete). Procedure SuperBlocks referenced from Condition blocks are linearized as algebraic. Dynamic blocks in such systems are not taken into account; their linearization is assumed to be 0.

Hybrid and multirate systems will automatically be linearized using the Multirate Linearization method. The syntax of the `lin` function is explained in detail in the online help.

10.2 Classical Analysis Tools Process

If the system contains a hierarchy of SuperBlocks, the analysis must be performed with the top-level SuperBlock open in the SystemBuild Editor window.

- To use the Time Response, Open-Loop Frequency Response, Point-to-Point Frequency Response or Root Locus tools you must first select an input block and an output block. To perform a multiple selection, select the input block, then hold down the Control key and select the output block. When you open the Tools menu you will see that the tools are enabled; select the desired tool.
- To use the Parameter Root Locus Tool, do not select anything in the model; simply select Tools→Parameter Root Locus.

A dialog box is presented, asking for the appropriate parameters, as explained in the subsections for each of the functions, below.

NOTE: Whenever you enter a new value in a dialog field you must press **Return** in that field to ensure that the value is read.

When all the parameters are entered and accepted, the software proceeds as follows:

1. The system is copied to a reserved SuperBlock named `_Analysis_System`.[†] This system is modified depending on the input and output blocks selected, and whether the mode of analysis is open-loop or point-to-point. The modified SuperBlock `_Analysis_System` may be edited and used like any other SuperBlock and is displayed in the catalog listing of SuperBlocks. However, you cannot use Analysis tools on `_Analysis_System`.

NOTE: Although the Tools menu functions alter the model by adding extra inputs and outputs where required around the area of interest, the analyzed portions remain in the diagram and the catalog. Thus, although the area of interest may be single rate, if a subsystem with a different rate is part of the original model, multirate linearization will be invoked, and may give unexpected results. If this presents

[†] The `_Analysis_System` SuperBlock is intended for your use. For example, you may choose to linearize the SuperBlock using the `lin` function, and then use the A, B, C, and D matrices obtained from linearization for analysis in Xmath.

difficulties, copy the area of interest to a temporary SuperBlock and edit it to remove the unwanted parts of the model before attempting another analysis. Likewise, if the model has other dynamic blocks that are not on the signal path of interest, these dynamics may appear as unobservable and uncontrollable modes during the analysis.

2. `_Analysis_System` is analyzed.
3. `_Analysis_System` is linearized. The operating point is fixed by the initial conditions entered in dynamic block dialog boxes and by the external input entered in the analysis dialog box.

NOTE: The external input has the reserved name `sb_uext`. This name should not be used by any user-defined Xmath variables, or they will be overwritten.

4. Finally, the appropriate Xmath function is invoked. The function results will be plotted. You can follow the progress of the function execution by checking for diagnostic or error messages in the Xmath Window.

10.3 Open-Loop Frequency Response

As soon as one or two blocks have been selected as the input and output blocks for the open-loop system, the menu item for open-loop frequency response is enabled. If you select that menu item, Tools→Open-Loop Frequency Response, the Frequency Response dialog box appears, as shown in [Figure 10-1 on page 10-4](#).

NOTE: The input block is the first block selected and the second block is the output-block. If you select only one block, it block will be used for both the input and output block.

The inputs area of at the top of the dialog box lists the input channels of the selected input block. Correspondingly, the output area at the top of the dialog box shows output channels of the selected output block. One channel each can be selected for the open-loop system input and output. This will define a Single-Input/Single-Output (SISO) system on which the selected frequency analysis is performed.

- The **Frequency MIN** and **Frequency MAX** fields allow you to define the frequency range for the resulting plot. Remember to press **Return** every time you edit a value.
- The **Number of Domain Points** field defines the number of frequency values at which the plot is calculated. Remember to press **Return** after editing this field.

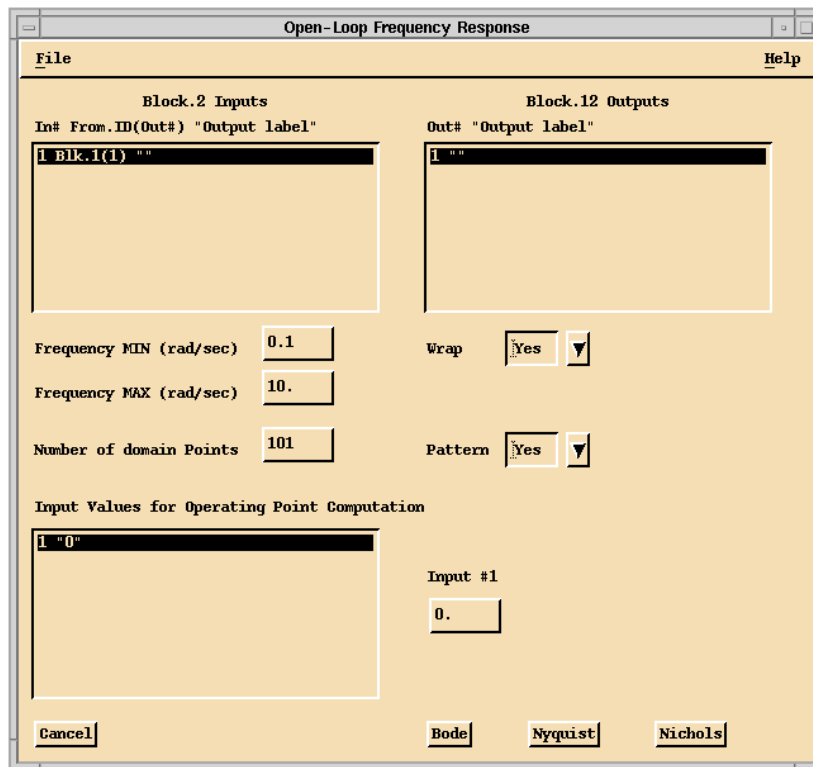


FIGURE 10-1 Open-Loop Frequency Response Dialog Box

- The **Operating Points of SuperBlock Inputs** region is a table listing the values of the external inputs with respect to which the linearization will be performed (default values are zero). Use the edit fields to the right of the list to change the value. Remember to press **Return** after editing this field.

The **Pattern** option draws constant M and N contours on the plot and **Wrap** limits the phase to $\pm 180^\circ$. Refer to the *Control Design User's Guide* for further details.

One of three frequency analyses can be performed: **Bode**, **Nyquist**, or **Nichols**. As soon as one of the analysis types is selected in the dialog box, the reserved SuperBlock `_Analysis_System` is created, containing a copy of the system. Then this system is linearized and either the appropriate continuous or discrete Xmath function for Bode, Nyquist or Nichols is invoked. A plot will appear with the desired analysis result.

Note that the Open-Loop Frequency Response analysis may be applied to components of closed-loop systems as well. In the case of a closed-loop system the loop is broken at the input channel that you specify in the Frequency Response dialog box.

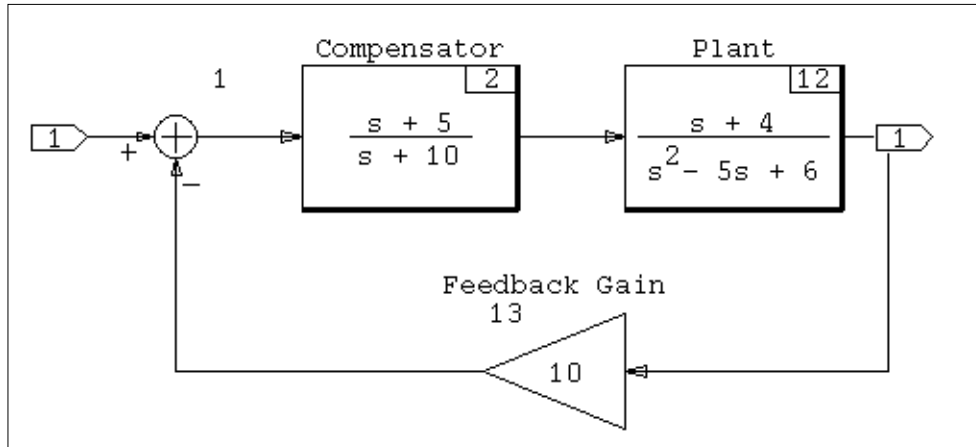


FIGURE 10-2 Open-Loop System “classical 1” Before Editing

For example, consider the system shown in [Figure 10-2](#). Type the following to copy this file to your local directory:

```
copyfile "$SYSBLD/examples/classical_example/classical.cat"
```

Load the file. To obtain the Bode plot of the open-loop Plant transfer function, first select the input then the output blocks. In this case, we want to use the block labeled “Plant” as both the input and output, so select only that block. Next, select Tools→Open-Loop Frequency Response.

The open-loop system that will be processed is available as SuperBlock `_Analysis_System`; an edited version of the [Figure 10-2](#) system is shown in [Figure 10-3](#) on page 10-6.

Note that in [Figure 10-3](#) the original loop is broken at the input to the plant and replaced by an external input; therefore the `_Analysis_System` is effectively the plant as we wanted to see it.

See [Figure 10-1](#) on page 10-4 for the Open Loop Frequency Analysis dialog box that appears next. In the dialog box Click **Bode**. See [Figure 10-4](#) on page 10-6 for the resulting plot.

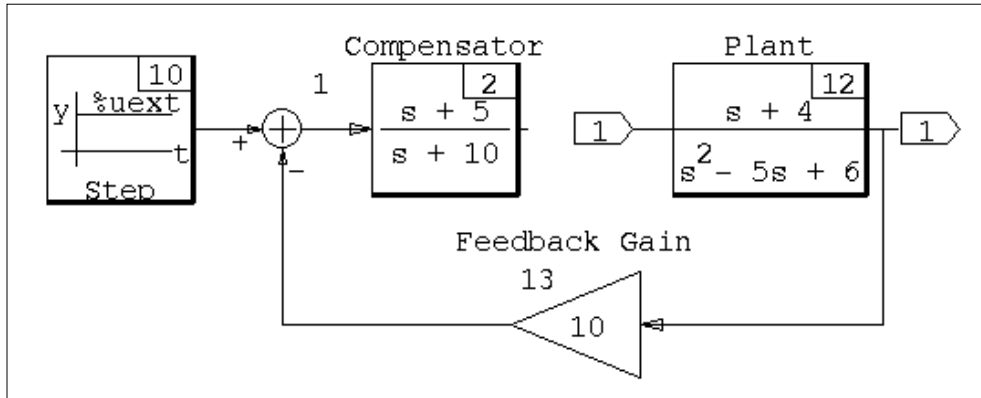


FIGURE 10-3 Open-loop System for Tools Menu Processing

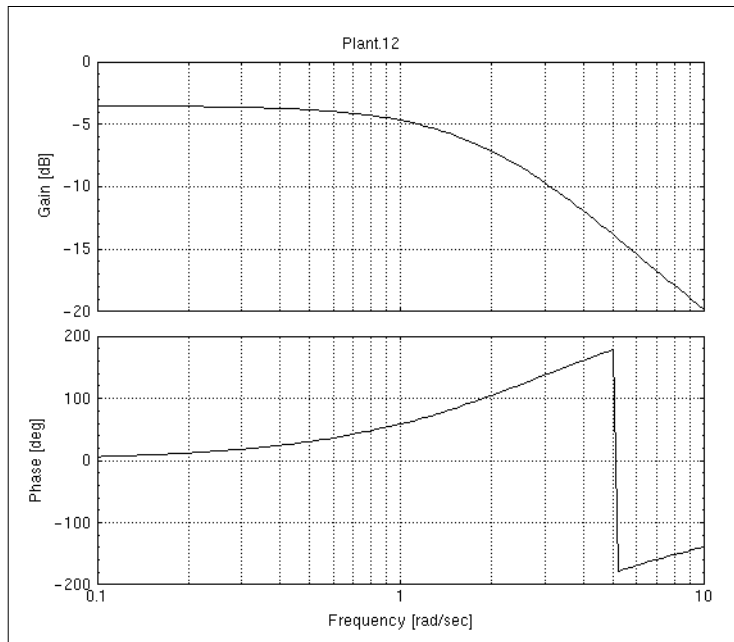


FIGURE 10-4 Open Loop Bode Plot (typical)

10.4 Time Response

As soon as one or two blocks have been selected as the input and output blocks for the open-loop system, the menu item for time response is enabled. If you select Tools→Time Response, the Step or Impulse Time Response dialog box appears, as shown in [Figure 10-5](#).

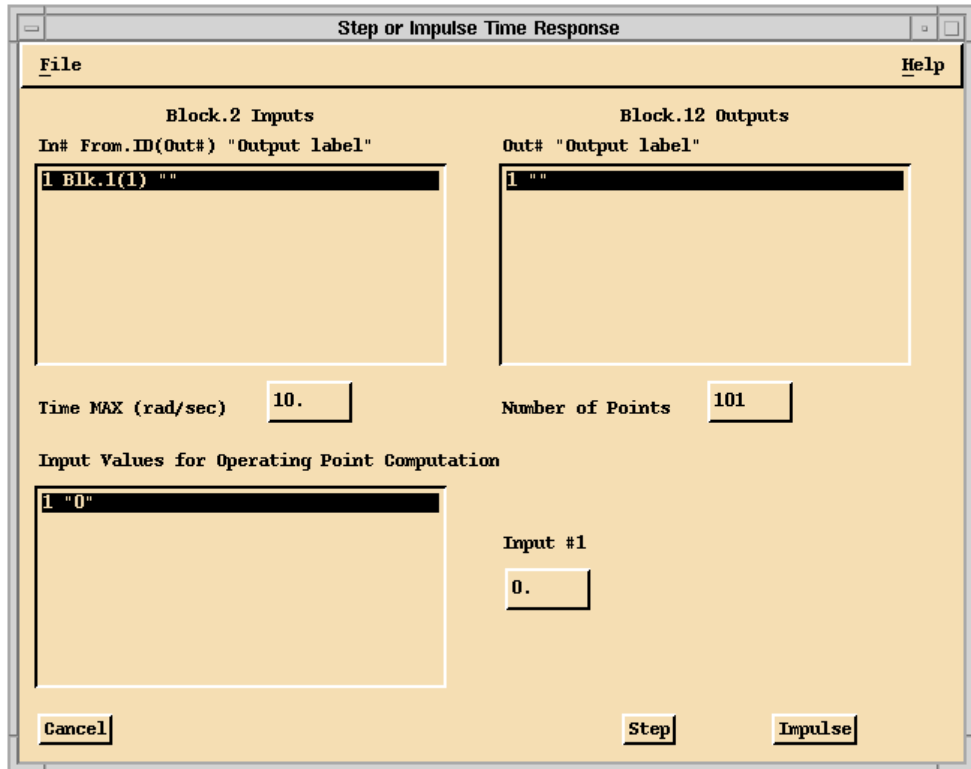


FIGURE 10-5 Time Response Dialog Box

NOTE: The block considered as the input-block is the first block selected and the second block as the output-block. If you select only one block, that block will be used for both the input and output block.

The inputs and outputs areas of the dialog box operate the same as for Open Loop Frequency Response, as explained in [Section 10.3](#). Remember to press **Return** each time you change a value.

The **Time MAX** field defines the final time for the analysis. The **Number of Points** field displays the number of points in time that the response is calculated. The **Operating Points of SuperBlock Inputs** field allows you to input a value for each input to the model and override the default value of 0. These are the values of the external inputs with respect to which the linearization will be performed.

Either step or impulse response can be selected, resulting in the modified system to be linearized and the Xmath function for step or impulse to be invoked. A plot should appear with the desired analysis result.

For example, consider the system shown in [Figure 10-2 on page 10-5](#). Again, selecting the Plant block as input and output then selecting Tools→Time Response, then press the Step, the modified system would be the same as in [Figure 10-8](#) and the resulting step response would be as shown in [Figure 10-6](#).

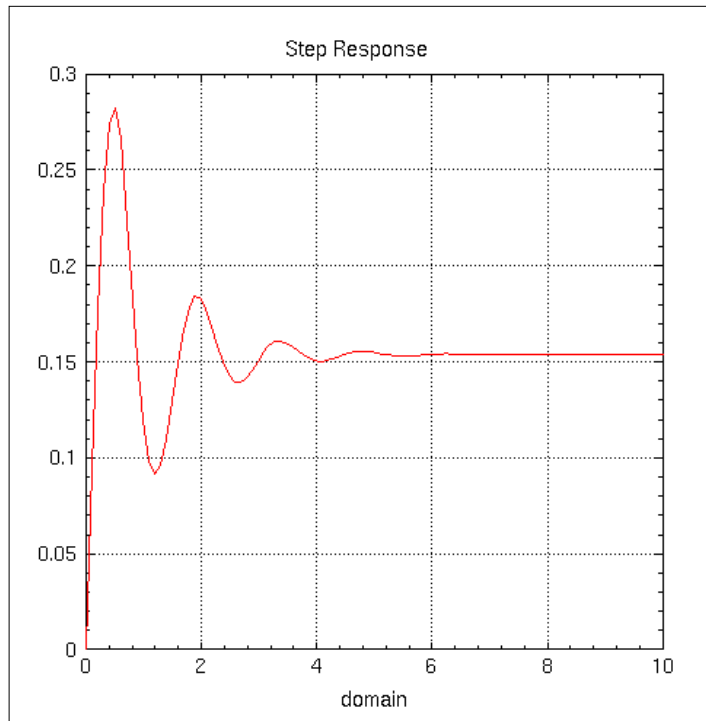


FIGURE 10-6 Step Response Plot

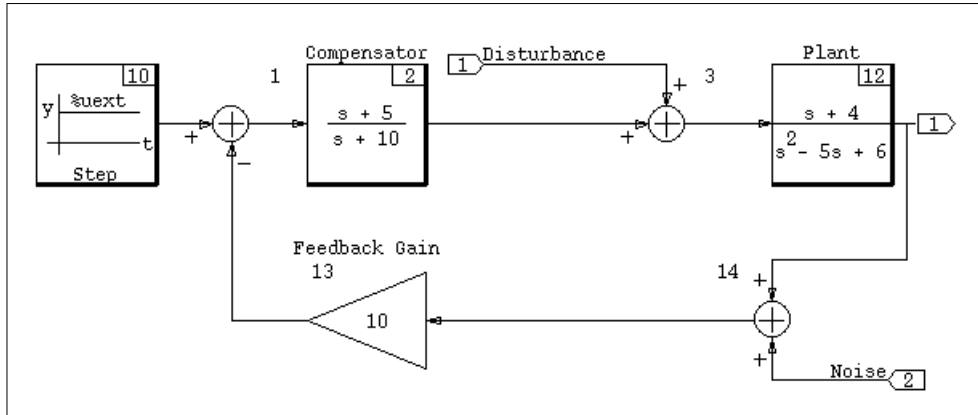


FIGURE 10-7 Analysis of the Open-Loop Example

10.5 Point-to-Point Frequency Response

A basic SISO feedback loop typically includes three exogenous signals[†]. [Figure 10-2 on page 10-5](#) incorporates only one of these three signals, namely the command (or reference) input. The remaining two exogenous signals, external disturbance and sensor noise signals, enter the loop as shown in [Figure 10-7](#) (this diagram is for illustrative purposes only). You can copy this model to your local directory with the following Xmath command:

```
copyfile "$SYSBLD/examples/classical_example/classical.cat"
```

Load the file. The SuperBlock is named `classical 2`.

In detailed or complicated block diagrams, it may not be convenient to include these extra exogenous inputs from the outset. Nevertheless, you may need to examine the transfer function(s) between various points of the feedback loop, without the loop being broken.

For example, you may need to examine the transfer function associated with the Disturbance signal (see Input #1 in [Figure 10-7](#)). To illustrate how this can be done, consider the system in [Figure 10-2](#) again. Suppose that the disturbance signal to the plant is not already built into the model, as is the case in [Figure 10-2](#), and you

[†] See Chapter 3 of *Feedback Control Theory*, by John C. Doyle, Bruce A. Francis, and Allen R. Tannenbaum, MacMillan Publishing Company, New York, 1992.

are interested in the transfer function from the disturbance signal at the plant input to the output of the feedback loop.

To accomplish this task, proceed as follows:

1. Select the block where the external input is to be injected. In our example the input is to be injected at the input channel of the Plant block.
2. Select the block from which the output signal is to be taken; in this example, this is the same block so there is no need to select another block.
3. Select Tools→Point to Point Frequency Response.
4. A dialog box appears, with input and output areas that list the input and output channels of the selected block or blocks, allowing you to select one channel each for the closed-loop system input and output. (In the example, the block is SISO and therefore the channel selection is automatic).

This selection defines a system wherein all original connections are left intact except that a signal is injected through an additional summation block, at the specified input port. Zero step inputs are attached to all the other external inputs, and the output is measured at the analysis output; see [Figure 10-8](#). This SuperBlock is named classical 4, and is available in the file you loaded earlier, `classical.cat`.

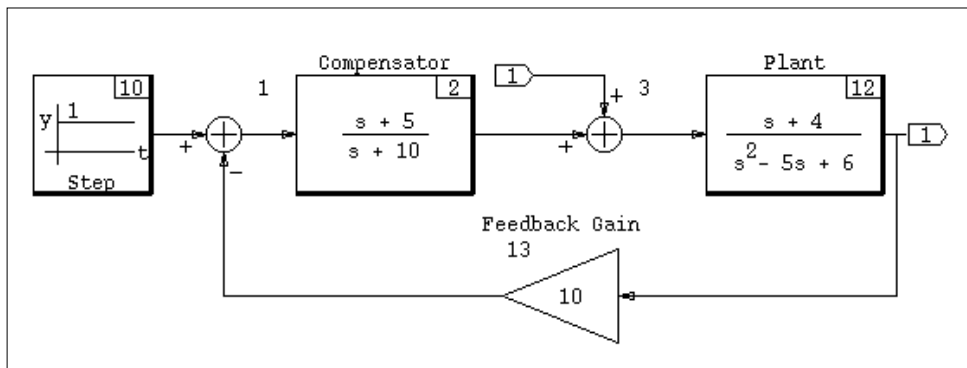


FIGURE 10-8 Closed-loop Example for Analysis

The fields on the dialog box and the three types of frequency analyses operate the same as was explained in the previous section for Open-Loop Frequency Response. If a Bode plot were requested, the plot shown in [Figure 10-9](#) would result.

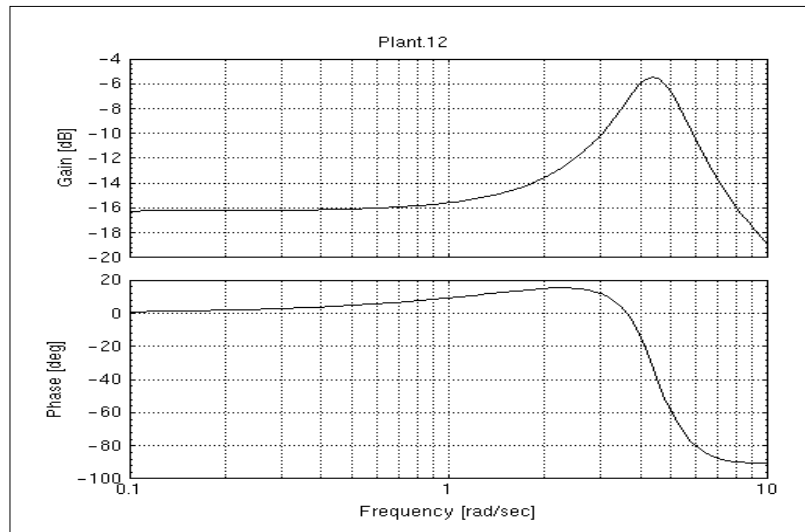


FIGURE 10-9 Bode Plot of the Point-to-Point Example

10.6 Root Locus

Given an open-loop, SISO system, $G(s)$, the Root Locus option of the Tools menu calculates and plots the roots of the closed-loop system that is composed of $G(s)$ and a variable, feedback gain K . This amounts to the calculation of roots of the equation $1+KG(s) = 0$ for a range of values of K . This range of values is typically specified by the user. The open-loop system is defined by specifying its input and output, as discussed in the previous sections.

As soon as one or two blocks have been selected as the input and output blocks for the open-loop system, the menu item for root locus is enabled. If you select Tools→Root Locus, the Root Locus dialog box appears, as shown in [Figure 10-10](#).

NOTE: The block considered as the input-block is the first block selected and the second block as the output-block. If you select only one block, that block will be used for both the input and output block.

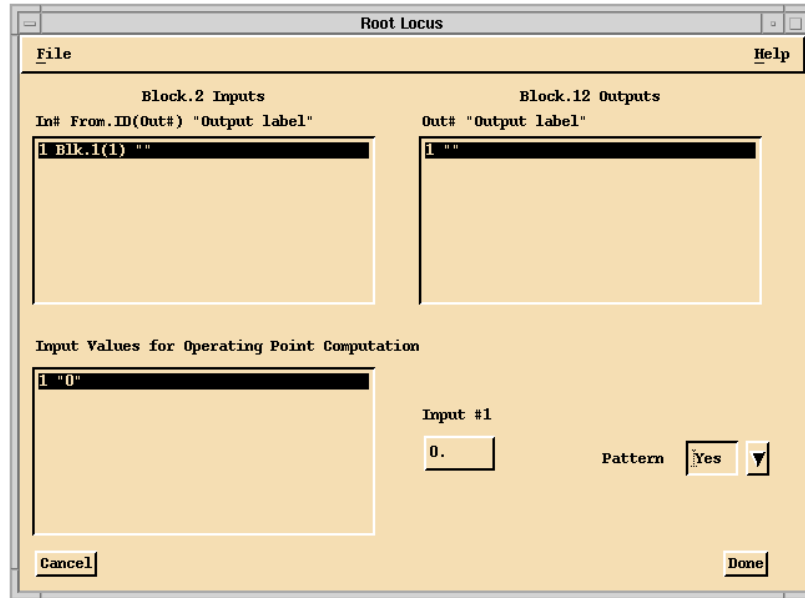


FIGURE 10-10 Root Locus Dialog Box

The Inputs area of the dialog box lists the input channels of the selected input block and the output channels show the corresponding selected output block. One channel each can be selected for the open-loop system input and output. This will define a SISO system on which the root locus analysis is performed.

When **PATTERN** is set to **YES**, lines of constant damping and constant natural frequency will be drawn on the root locus plot.

NOTE: The Root Locus function invoked in this way is the standard Xmath interactive root locus function; you may need to rearrange the screen windows to gain access to the Xmath Root Locus dialog box, from which you can change the interactive gain value. Refer to the online help or the *Xmath Control Design Module* for further details.

For example, consider the system from [Figure 10-2 on page 10-5](#) again. If the Plant Block were selected as input and output, then the system shown in [Figure 10-3](#) would be created and linearized, and the resulting root locus plot would be as shown in [Figure 10-11](#), subject to changes you may make in the Interactive Root Locus dialog box.

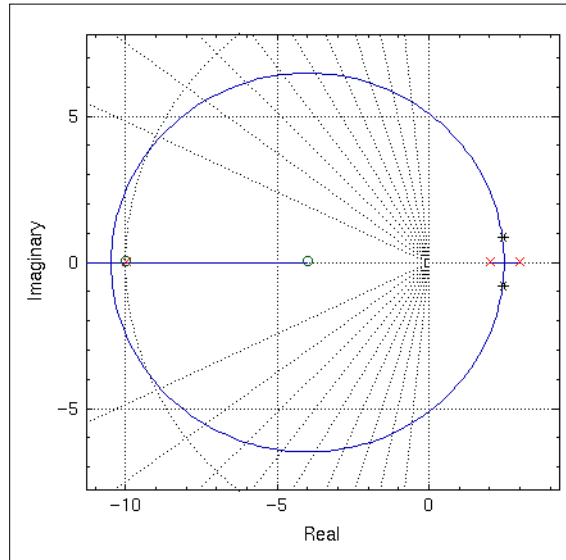


FIGURE 10-11 Root Locus Plot

NOTE: If your model has other dynamics that are not in the signal path of interest, these dynamics will appear in the root locus plot as unobservable-uncontrollable modes. They do not affect the root locus of interest, except in the case of a discrete system with uncontrollable or unobservable dynamics where these dynamics are of a different rate, necessitating multirate calculations that would influence the root locus calculations.

10.7 Parameter Root Locus

The Parameter Root Locus functionality differs from the Root Locus. When the user defines a SISO system, $G(s)$, the Root Locus option automatically creates a closed-loop system as described in [Section 10.6](#). This assumes that you are interested in using a feedback gain, K , to create a closed-loop system from $G(s)$. In many situations, you will have already closed the loop and are only interested in the roots of the specified system when internal parameters, which have to be defined as %Variables, are changed. Thus, there is no need for closing any loops, and the parameters are internal. The Parameter Root Locus option provides this capability; with it you to obtain root loci of nonlinear systems.

An example of a nonlinear system is shown in [Figure 10-12 on page 10-14](#), where the inner_loop, outer_loop, and Channel 1 Feedback gains are parameterized.

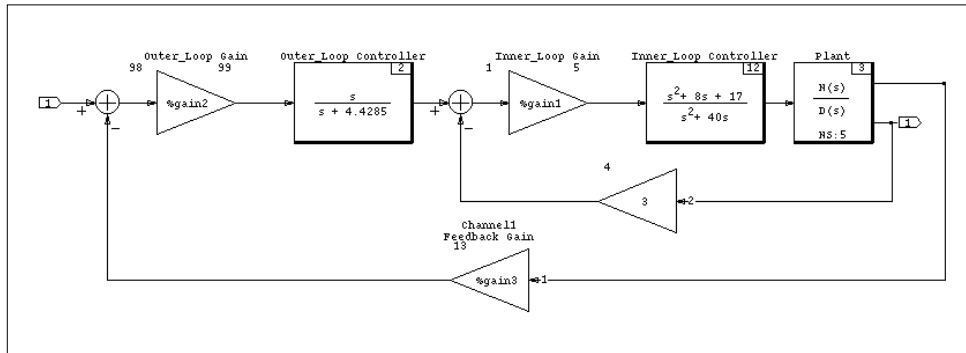


FIGURE 10-12 example1

To copy this model to your local directory, type:

```
copyfile "$SYSBLD/examples/classical_example/pr1.cat"
```

Load the file, open 'example1' in the Editor, and select Tools→Parameter Root Locus.

NOTE: The Parameter Root Locus option is enabled when there are no blocks selected.

The dialog shown in [Figure 10-14 on page 10-15](#) appears. It lists all scalar %Variables referenced in any block residing in the currently displayed SuperBlock and the hierarchy of SuperBlocks under it. One %Variable can be selected as well as a range for varying the parameter and the number of points.

The root locus is computed by varying the %Variable from MIN to MAX with a step-size of $(MAX-MIN)/NPTS$ and at each value plotting the real and imaginary parts of the eigenvalues of the linearized system. The plot is thus parameterized in the %Variable. The eigenvalues corresponding to the minimum and maximum parameter value are plotted in a different color. The %Variable is restored to its original value at the end of the analysis.

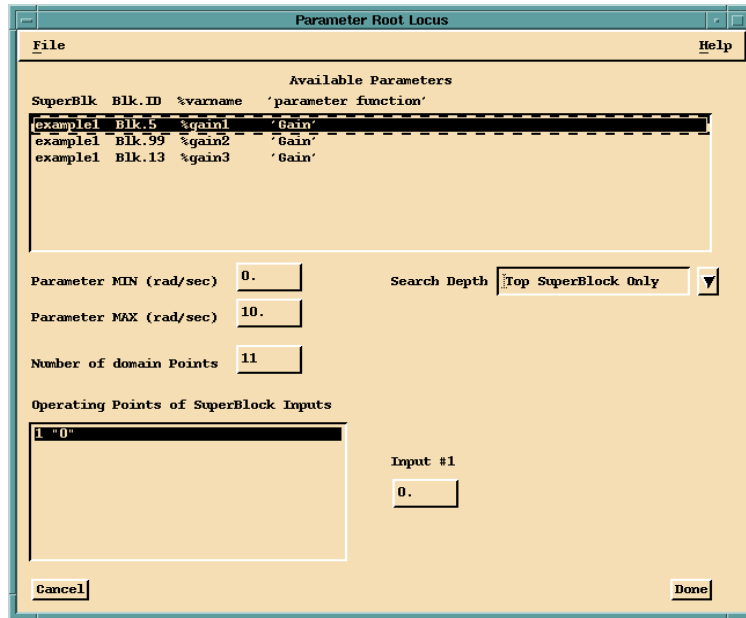


FIGURE 10-13 Parameter Root Locus Dialog Box

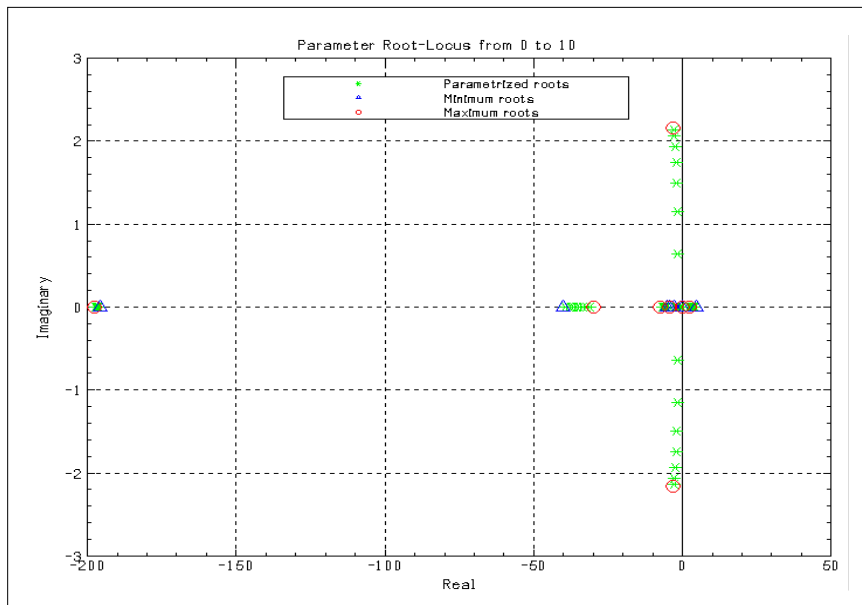


FIGURE 10-14 Parameter Root Locus Plot of the example1 Diagram

Multirate, Nonlinear System Root Locus

To illustrate some of the hidden steps that are executed for an analysis, let us consider a multirate, nonlinear system. To create the system, copy the following file to your local working directory:

```
copyfile "$SYSBLD/examples/classical_example/mws_demo.cat"
```

Load the file and edit the SuperBlock named BUILT_MODEL (see [Figure 10-15](#)).

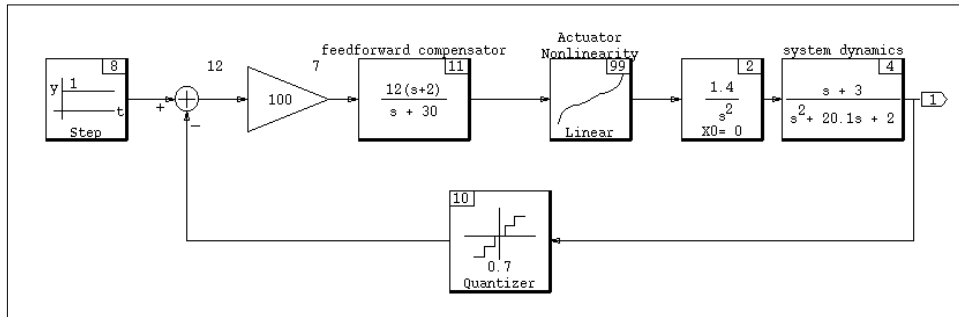


FIGURE 10-15 Built_Model SuperBlock

1. Select the feedforward compensator, then select Edit→Make SuperBlock.
2. Go to the Catalog Browser and select the new SuperBlock (_makesb). Select Tools → Transform, and in the Transform SuperBlock dialog, make the type discrete, and specify a sample period of 0.01,
3. In the Editor, select the Gain = 100 (ID 7) block as input, then control-click to select the system dynamics block (ID 4) as output.
4. Select Tools→Root Locus. Change the X Min to .5, the X Max to 1.1, the Y min to -.5, the Y Max to .5 and the Gain to 0.7, then click Done.

The root locus that is obtained, shown in [Figure 10-16 on page 10-17](#), corresponds to an equivalent single rate linear system with sample interval $T = 0.01$ and gain = 0.7. Therefore, the stability properties of the root locus must be interpreted as for a discrete system. See [Section 9.6 on page 9-7](#) for more multirate linearization.

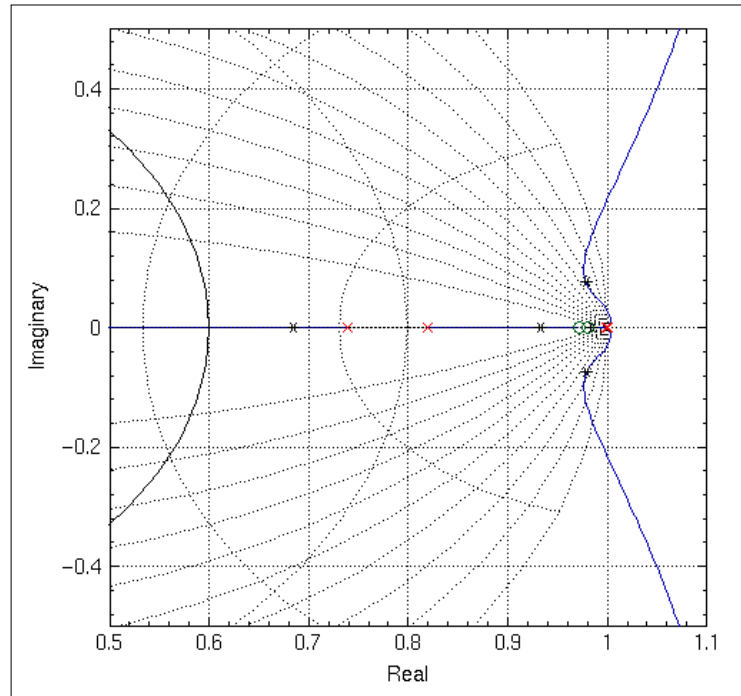


FIGURE 10-16 Root Locus Plot of the Built Model

The operating point has an important effect on the linearization of nonlinear systems. The operating point is defined by the external input, entered in the Tools men forms, and by the state initial conditions. See [Section 11.2 on page 11-10](#) for further explanation of operating points. The effect of the operating point on a linear analysis can be easily illustrated.

1. Load the original model `$SYSBLD/examples/mws_examples/mws_demo.cat` open the system `BUILT_MODEL`.
2. Select the Gain = 100 (ID 7) block as input, then control-click to select the system dynamics block (ID 4) as output.
3. Select Tools→Root Locus.
4. Change the root locations interactively. Take note of where the roots are when the gain is 100.

5. Make a SuperBlock out of the four blocks connected between Feedforward Compensator block to the System Dynamics (IDs 11, 99, 2, and 4). Select File→Update.

The new SuperBlock's name is the default name, `_makesb`. To rename `_makesb`, switch back to the Catalog Browser, expand the BUILT_MODEL subhierarchy in the left pane, and select `_makesb` SuperBlock; select Edit→Rename and name the new SuperBlock, for example, "newsb".

NOTE: If you do not rename the SuperBlock, Root Locus will fail.

6. Open newsb. In the Editor, do a root locus analysis with the same input and output blocks (IDs and 4), and launch the Root Locus tool. Note that this time there is an Opoint field. Set the gain to 100 and compare the values with those you noted in [step 4](#).

Note that now when the Gain = 100, the roots are not in the same place as the previous root locus plot. The reason is that in newsb, the extra blocks in the system are removed so that only the path from Feedforward Compensator to system dynamics remains. Therefore, the calculation of the operating point done by the function `opoint()` is different.

In the first case, the external input is specified by the step signal but in the second case it is defined by the external input entered in the Root Locus dialog box, with a default of zero. The nonlinear single variable interpolation block is therefore linearized at a different point, or, in other words, the equivalent gain for the block is different. Consequently, the open-loop gain of the system is different, affecting the closed-loop pole locations on the root locus plot.

11

Advanced Simulation

This chapter discusses:

- Explicit vs. implicit models
- Operating points
- Integration algorithms
- State events

11.1 Explicit vs. Implicit Models

Continuous simulation models can be implicit or explicit. Implicit blocks can be found on the Implicit palette. The majority of SystemBuild blocks are explicit, which means that the state derivative can be written as an explicit function of the state.

A model is implicit if it contains implicit blocks and/or algebraic loops.

- If a model contains algebraic loops, but no implicit blocks, it can be simulated using both implicit and explicit integration algorithms. The analyzer will however issue a warning message indicating that the simulation result is not reliable when explicit algorithms are used.
- Models that contain implicit blocks can only be simulated using implicit integration algorithms.

11.1.1 Explicit Models

Explicit models are defined by an underlying Ordinary Differential Equation (ODE) where the state derivative is explicitly computed from the state:

$$\dot{x}(t) = f_e(x(t), t) \quad \text{EQ. 11-1}$$

Note that the external input is not included as one of the right hand side arguments. This generally accepted formalism simplifies without loss of generality. Indeed, the dependence on the external input is really a time dependence defined by linear interpolation of the values in the input matrix as a function of time.

11.1.2 Implicit Models

Implicit models are defined by a residual (which is a function of time), the state, and the state derivatives:

$$\delta(t) = f_i(\dot{x}(t), x(t), t) \quad \text{EQ. 11-2}$$

The solution is defined by setting $\delta(t)$ to zero and solving for $x(t)$ and $\dot{x}(t)$. This type of equation is generally referred to as a Differential-Algebraic Equation (DAE), since it may imply non-dynamic or algebraic relations between elements of the state vector. Note that DAEs are inherently associated with constraints, because they require the residual to be zero. Also, any ODE can be reformulated as a DAE by defining the residual as the difference between the left and right hand sides of the ODE equation. Conversely, DAEs cannot generally be reformulated as ODEs.

In order to be able to solve the DAE, some additional assumptions have to be made. For example, the Stiff System Solver (DASSL) assumes nonsingularity of the Jacobian, a condition which is often hard to guarantee and verify. Despite these problems, implicit integrators are important because they are the only ones that can handle algebraic loops and implicit blocks correctly.

Implicit Model Constraints

Constraints can be used to prevent the solution from drifting away from any pre-defined manifold known for the solution. They can be imposed with implicit UCBs, constraint blocks and/or algebraic loops ([Example 14-1 on page 14-7](#) illustrates this).

SystemBuild differentiates between two types of constraints: required and auxiliary.

- Required constraints are defined as an arbitrary set of constraints that are necessary to solve the DAE, such that each variable required to solve the problem has a corresponding constraint. Any additional constraints are called auxiliary.
- Auxiliary constraints make a problem overdetermined since, generally speaking, they add more equations than there are variables. The notion of solvability goes mathematically beyond reasoning based on numbers of equations versus constraints. (For practicality, SystemBuild ignores these issues, since they are usually irrelevant and would add an unjustifiable degree of complexity.)

We distinguish between required and auxiliary constraints because auxiliary constraints make the Jacobian non-square, requiring different state update equations than those based on matrix inversion. The biggest difference is, that if auxiliary constraints are used, the δ vector is partitioned into two segments, which we refer to δ_r (required constraints) and δ_a (auxiliary constraints).

Implicit States and Implicit Outputs

The simulation state is described by explicit states, implicit states, and implicit outputs:

$$x = \begin{bmatrix} x_i \\ x_e \\ y_i \end{bmatrix}$$

The three components have the following meaning:

- x_i** Implicit states, introduced by implicit UCBs or implicit variable blocks.
- x_e** Explicit states, introduced by any other dynamic blocks.
- y_i** Implicit outputs, introduced by algebraic loops.

Note that this state vector is an augmentation of the implicit and explicit state vectors with implicit outputs which are discussed next. It is important to realize that in order to save the state of the simulation, the implicit outputs must be saved as well.

Implicit Outputs

Implicit outputs are block outputs in an algebraic loop selected by the SystemBuild analyzer as the starting point for direct evaluation of the loop equations. Implicit outputs are different from other outputs in the sense that the diagram cannot be evaluated without them. They are different from states in the sense that they can get instantly overwritten during the diagram evaluation. How this is done depends on the block sorting by the analyzer, and is generally unpredictable. In spite of this nondeterministic behavior, implicit integration algorithms can compute the numerically correct solution.

Initialization

Implicit model initialization is a little more complicated than initialization for explicit models. For Implicit models, both states and state derivatives must be initialized. In order for the simulator to compute the operating point at the start of the simulation, you must specify whether the search will be done over the states or the state derivatives. Both the `ImplicitUserCode` Block and the `ImplicitVariable` block have a combo box where this search mode can be set. It is important to realize that if the diagram contains no state derivatives, the search is done over the states, and vice versa. The operating point computation can be bypassed using `initmode = 4`. For more details, see [Section 14.2.4 on page 14-8](#).

11.1.3 Implicit Model Examples

We will illustrate the effect of implicit outputs and give examples of some of the available implicit blocks.

EXAMPLE 11-1: Algebraic Loop

A model containing an algebraic loop is shown in [Figure 11-1 on page 11-5](#). The effect of the feedback loop containing the gain block is twofold: (1) an implicit output is created after the summation block, and (2) the first input to the summer evaluates to zero.

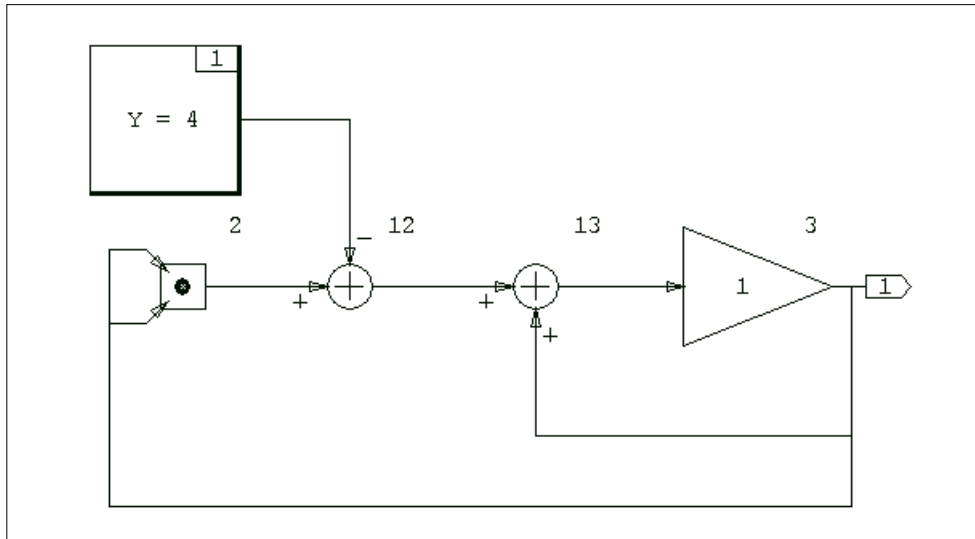


FIGURE 11-1 Algebraic Loop Solver of $x^2 = 4$

When evaluated, this diagram solves the equation $x^2 = 4$, resulting in the answer 2 or -2. We can obtain an answer of -2 (only) with the following commands:

```
t = [0:10]';
[,y] = sim("gallop", t, {ialg=6});
y?
```

Either `ialg=6` (DASSL) or `ialg=9` (ODAS), will produce the desired result. Note, feeding the output of the gain block into the DotProduct block, avoids singularity of the Jacobian, making it possible for O/DASSL to solve the equation without problems.

EXAMPLE 11-2: The Implicit Output Block

Disadvantages of solving the equation $x^{**2} = 4$ using the model in [Figure 11-1 on page 11-5](#) are:

- The analyzer arbitrarily decides the location of the implicit output. You cannot influence the analyzer's decision.
- In order to initialize the implicit output you must pass `yimp0` to the `sim` command. In more complicated situations, where there are several implicit outputs, you would need to know the composition of `yimp0`, which also depends on the analyzer.

The Implicit Output Block replaces the implicit output with an implicit state, allowing you to enter the initial condition in its block dialog. This change is shown in [Figure 11-2](#).

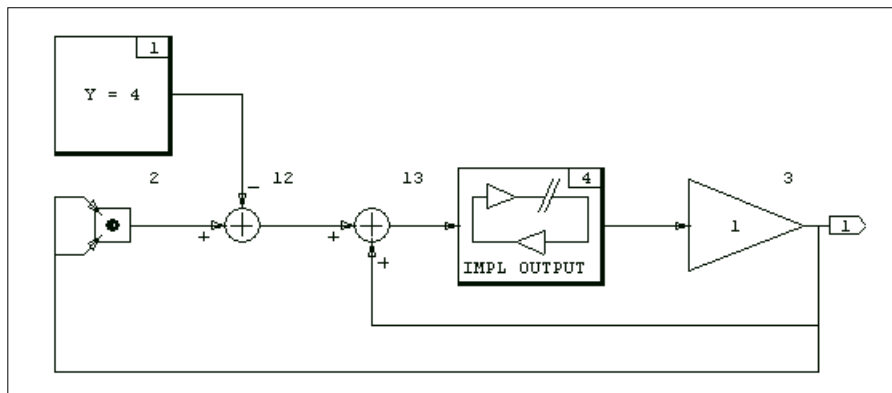


FIGURE 11-2 Solving $x^{**2} = 4$ using an Implicit Output Block

Because the model now contains an implicit block, it can only be simulated with implicit integration algorithms.

EXAMPLE 11-3: The Implicit Variable and Constraint Blocks

The most appropriate way of solving $x^{**2} = 4$ is to use a combination of an Implicit-Variable block and a Constraint block. The ImplicitVariable block has an output vector consisting of two segments, (1) the implicit state vector, and (2) the implicit state derivative vector.

For the ImplicitVariable block, the **Initialize Mode** field determines whether the search for initial values is done over the states (frozen states) or over the derivatives (frozen derivatives). In this case, the states must be frozen in order to simulate the diagram. The initial value cannot be zero, since this will lead to a singular problem. Any nonzero value will work.

For the ImplicitConstraint block, the **Constraint Type** field specifies whether the constraint type is Required or Auxiliary. Since the diagram has an equal number of implicit variables and constraints, we can assume the constraint in this diagram is required.

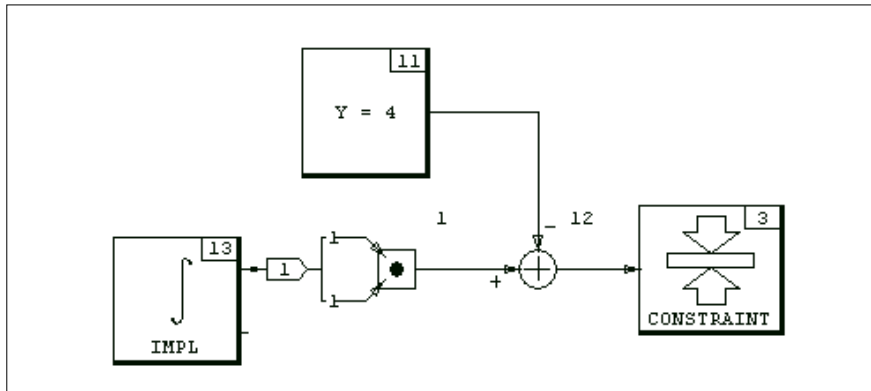


FIGURE 11-3 Solving $x^{**2} = 4$ using Implicit Variable and Constraint Blocks

EXAMPLE 11-4: An Exact PID Controller

A more useful example is the use of the ImplicitVariable and Constraint blocks to simulate an exact PID controller (Figure 11-4).

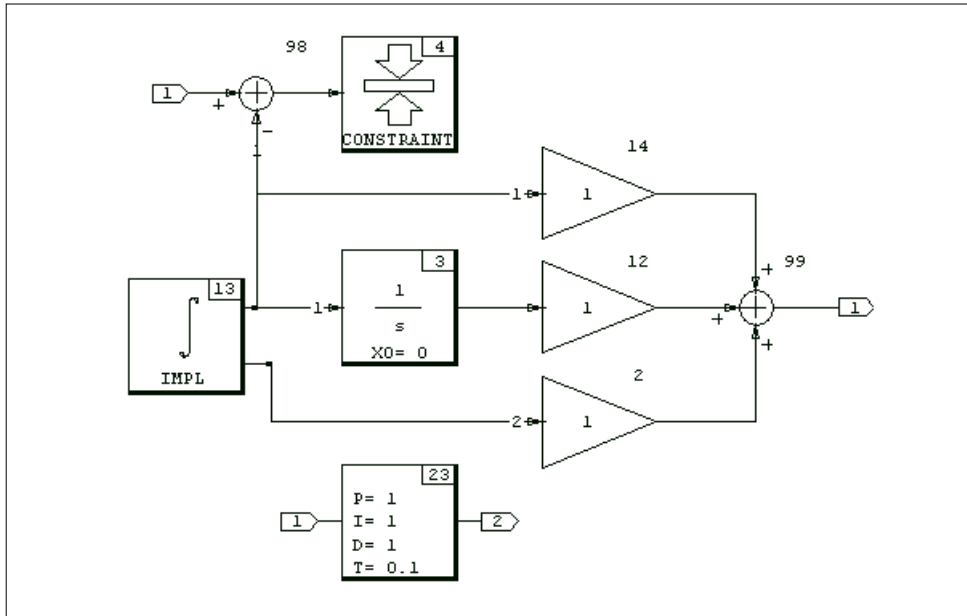


FIGURE 11-4 An Exact PID Controller

For the state derivative output of the ImplicitVariable block to represent the derivative of the input, the constraint is imposed so that its state output is equal to the external input. The command used to simulate this system and plot the output is:

```
t = [0:10000]'/1500;
[,y] = sim("imppid", t, sin(t.^3), {ialg=6});
plot(t, y)
```

Figure 11-5 on page 11-9 compares this output with the output of the PID Controller block on the dynamic palette using the default parameters.

Note that the comparison is done over frequencies far beyond where the derivative approximation of the (explicit) PID block is valid. The PID block performs much better if the default parameter tau = 0.1 is replaced with 0.01 instead.

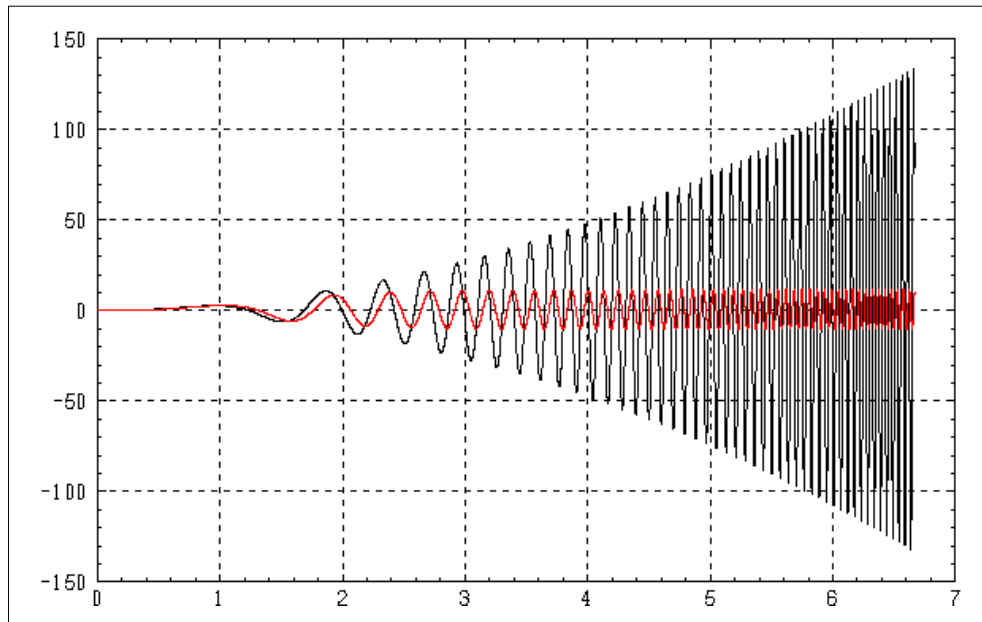


FIGURE 11-5 Comparison of Exact vs. Approximate PID Controllers.

EXAMPLE 11-5: The Inverted Pendulum

For this example we refer forward to [Chapter 14](#), where an inverted pendulum example is used to illustrate `ImplicitUserCode` block capabilities. This example can also be simulated using `ImplicitVariable` and `Constraint` blocks, so that a UCB is not required. An interesting aspect of that example is that it can be simulated under a variety of conditions, among which are the cases of one or two auxiliary constraints. The implicit block version of the inverted pendulum example can be found in `$$SYS-BLD/examples/pend_imp_no_iucb`. Load the catalog and run the `MathScript` found in that directory.

11.2 Operating Points

The operating point for each subsystem must be calculated at the beginning of every simulation, for all linearizations, and for all `simout` invocations. The objective of an operating point calculation is to determine consistent values of all states, state derivatives, block outputs and implicit algebraic loop variables. This section discusses the methods for finding the operating point.

11.2.1 Continuous Subsystem

The continuous operating point is found by evaluating all continuous block outputs, based upon the system initial states and inputs. If the system contains algebraic loops or implicit states (via the Implicit UserCode Block), the operating point is found by first applying a Newton-Raphson root solver. Then the other continuous block outputs are computed based on the algebraic loop outputs, implicit state values, external input values, and initial state values. Because the root solver finds a steady-state value for the algebraic loop outputs, initial transients in simulation due to an incorrect operating point will not occur.

If algebraic loop initial conditions are specified with the `yimp0` option, the operating point computation for the algebraic loops will utilize the `yimp0` values as initial conditions for the Newton-Raphson solver.[†]

11.2.2 Discrete Subsystems

Several levels of initialization are available for discrete subsystems, which you may select using the `initmode` keyword. By default, initialization is performed at the `initmode = 3` level.

The lowest level, corresponding to `initmode = 0`, simply sets all block output values for each subsystem to $-\sqrt{\epsilon}$.

At `initmode = 1`, after all outputs are set to $-\sqrt{\epsilon}$, outputs are computed for discrete subsystems that do not have enable or trigger flags. The computations are based on the initial states and inputs.

At `initmode = 2`, after the two lower levels of initialization described above are performed, outputs are computed for discrete subsystems that are enabled or triggered and “active” (because their enabling or triggering signals evaluate True). The computations are based on the initial states and inputs.

[†] Note that operating point computations for implicit states as well as algebraic loops are performed only for continuous subsystems.

At `initmode = 3`, the initialization is the same as `initmode = 2`, except that there is no sample and hold between subsystems.

At `initmode=4`, disables the operating point computation performed at the beginning of the simulation. Note that with `initmode=4`, consistent initial conditions have to be supplied for algebraic loop variables (if any), using the keyword `yimp0`. If this is not done, the implicit variables may have incorrect values at initialization time.

The order in which subsystems are executed is determined by their relative priorities. Subsystems with shorter sample intervals are executed before subsystems with longer sample intervals.

The initialization procedure may fail in the event that a high priority subsystem is dependent on the output of a low priority subsystem. In some cases, the problem can be avoided by setting `initmode = 0`, which may result in a different initial subsystem execution order, because the simulation scheduler considers other attributes that affect timing (initial time skew), which the initialization procedure does not. In extreme situations, you may need to modify or redesign your model to be insensitive to the bounds of values communicated between subsystems.

Note that code generated using AutoCode does not contain initialization procedures, such as those in the simulator. As a result, generated code behavior is similar to the case where `initmode = 0`.

CAUTION: Algebraic loops in discrete systems should be avoided.

If a discrete subsystem contains algebraic loops, a computational delay occur because some block outputs will be used in computations before they are calculated during each time interval. This computational delay will represent a pseudo-dynamic which may produce non-steady transients in the simulation results.

11.3 Inserting Initial Conditions

The `sim`, `simout`, and `lin` functions allow initial conditions to be set for dynamic blocks residing in the analyzed system, overriding the initial conditions defined in the individual block forms of the dynamic blocks. This allows you to determine quickly the effect of the initial operating point on a subsequent simulation or linearization.

The operating point can be changed by supplying `x0`, `xd0`, and `yimp0` values as optional arguments to the `sim`, `simout`, or `lin` functions. These keywords insert initial dynamic state values (`x0`) initial state derivatives for implicit UserCode blocks (`xd0`) and initial algebraic loop output values (`yimp0`).

If any of the initial condition options are set, the simulation data bus is loaded with the user-furnished vector. This action only initializes the run-time tables for the current execution of `sim`, `simout`, or `lin`. It does not overwrite the initial conditions that are stored on a block-by-block basis in the SystemBuild catalog. The catalog values can be changed only by editing the block diagram.

For the syntax and method of operation of the `sim`, `simout`, and `lin` initial condition features, see the online help.

States associated with the Padé approximation of continuous delay blocks cannot be accessed by the `x0` keyword under `sim`. These states are initialized to zero at the start of a `sim` or `lin`, unless the `sim` is being resumed, when the Padé states revert to the value they had at the end of the last simulation.

The simulator's `x0` initial condition option does not allow initialization of memory states. These include State Transition Diagram (STD) states, and DataStore initial conditions. These initial conditions are initialized using values stored in the SystemBuild catalog, except when a `sim` is resumed, in which case the memory states revert to the values they had at the end of the last `sim`. The `analyze` function only lists the states that `sim` insertion can access, and does not include the memory states.

Via the block parameter dialog boxes, SystemBuild allows you to specify initial *outputs* for the transfer function blocks, NumDen, PoleZero, and ComplexPoleZero. Using the internal state-space representations of these blocks, SystemBuild assigns appropriate initial conditions in these blocks so as to produce the desired initial outputs. However, this mapping is not necessarily unique and is not visible to the user, and therefore initialization of these states should be used with caution.

Also, to be consistent with the continuous case, SystemBuild initializes discrete integrator blocks at the integrator output point, rather than at the discrete delay output. Thus, as shown in Figure 11-6, if the initial condition x_0 is defined in the block parameter dialog, it will be applied at the output of the integrator. By contrast, the `sim x0` option initializes the integrator's state, \hat{x} .

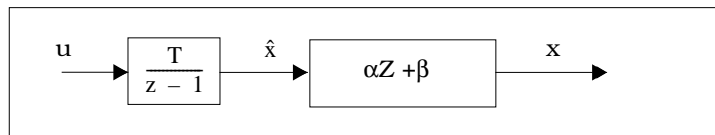


FIGURE 11-6 1/s Substitution in Discrete Integrator

11.4 Use of sim, lin, simout for Implicit UCBs

In the block dialog box of Implicit UCBs, there is a field that allows the definition of either states or derivatives as initial conditions. The interpretation of the `sim`, `lin`, or `simout` arguments `x0` and `xd0` for state and derivative initial conditions are *dependent on the dialog box definition*.

Dialog Definition	sim(), lin(), simout() argument	
	<code>x0</code>	<code>xd0</code>
states	<code>sim</code> initial condition	initial condition for operating point solver
derivatives	initial condition for operating point solver	<code>sim</code> initial condition

SystemBuild's operating point solver will compute the correct initial conditions for x or \dot{x} before a `sim`, `lin`, or `simout`, such that the implicit equations are satisfied. However, there may be cases in which the operating point is singular.

When the operating point is singular, the user must provide these initial conditions. The `simout` function has a special feature that facilitates this: when `simout()` is used with `initmode = 4`:

```
[x,xdot,y]=simout("model",{x0=x0_init, xd0=xd0_init,
    u0=u0_init, initmode=4})
```

then the operating point computation is bypassed and the `xdot` vector contains *not the derivatives*, but the residuals $\delta = f(x0, xd0, u0)$ in its corresponding entries.

You can use this feature to write algorithms that compute the correct initial conditions `x0`, `xd0`, or `u0` iteratively.

11.5 Matrix Blocks in the Simulator

While the matrix blocks carry out well-defined matrix-theoretic functions on their inputs, there may be a question as to what algorithm(s) they employ and what error bounds they should be expected to obey. The Constant and MatrixTranspose block are extremely simple and introduce no error. The ScalarGain, MatrixMultiply, LeftMultiply, and RightMultiply are implemented using sequential scalar multiplication (ScalarGain) or repeated dot product (the rest), and may introduce some error as a consequence of the floating-point or fixed-point multiplications being carried out. (See [Section 15.1.4 on page 15-7](#).)

This leaves us with the `MatrixInverse`, `MatrixLeftDivide`, and `MatrixRightDivide` blocks which compute $1/A$, or solve $AX=B$ or $XA=B$ for the unknown X . These blocks involve complicated algorithms which may produce a large relative error, depending upon the input. In Sim these blocks are implemented using the Gaussian elimination algorithm since this algorithm's behavior is well-known and definitive statements can be made about how much error the solution X or the inverse $1/A$ will contain (thanks to the reciprocal conditioning number, which is generated by the standard LINPACK implementation of Gaussian elimination). The simulator will halt in a `MatrixInverse`, `MatLeftDivide`, or `MatRightDivide` if the reciprocal condition number goes exactly to zero; this indicates it is impossible to invert or divide by the matrix in a meaningful manner.

11.6 Sim Integration Algorithms

Dynamic models created in SystemBuild can be broadly categorized as follows:

- Continuous
- Discrete
- Hybrid (i.e., a combination of continuous and discrete subsystems)

The procedure of simulating the SystemBuild model, or obtaining a sequence of solutions to the system equations given the user-defined initial conditions and input vector, is fairly straightforward for discrete systems. Starting from the given initial conditions the discrete state equations are iterated until the specified final time.

Finding a numerical solution for continuous and hybrid systems, on the other hand, requires a proper method of approximation. The purpose of an "integration algorithm" or differential equation solver is to calculate an accurate approximation to the exact solution of the differential equation. Then the solution is "marched" forward from a starting time and a given set of initial conditions.

Since all continuous integration algorithms are inherently approximations, there are a number of important points to consider in selecting a proper method: computational efficiency, truncation and round-off errors, accuracy and reliability of the solution, and stability of the integration algorithm. Here we discuss these issues and the advantages and disadvantages of each method included in the ISI repertoire of integration algorithms. Hints and recommendations for choosing suitable methods for various types of models are also provided.

These five terms are used in the following descriptions:

T	Current Time
DT	Time Step
ODE	Ordinary Differential Equation
DAE	Differential Algebraic Equation
ODAE	Overdetermined Differential Algebraic Equation

Integration methods may be divided into four classes: One-step, multi fixed-step, variable-step, and stiff system solvers.

11.6.1 Comparing Integration Algorithms

The following list enumerates the supported Integration Algorithms. The numbers correspond to the selection indices used in Xmath and SystemBuild to specify an algorithm; the abbreviations in parentheses are also accepted by SystemBuild.

1. Euler's method (euler)
2. Second-Order Runge-Kutta (RK2)
3. Fourth-Order Runge-Kutta (RK4)
4. Fixed-Step Kutta-Merson (FKM)
5. Variable-Step Kutta-Merson (VKM)
6. Differential-Algebraic Stiff System Solver (DASSL)
7. Variable-Step Adams - Bashforth - Moulton (ABM)
8. QuickSim (QSIM)
9. Over-determined Differential-Algebraic Stiff System Solver (ODAS)
10. Gear's method (GEAR)

The default integration algorithm is 5 (Variable Kutta-Merson). The algorithm may be set globally using the command:

```
setsbdefault, {ialg=alnumber}
```

ialnumber is taken from the list above.

You can also determine the current default number using the command:

SHOWSBDEFAULT

To set the integration algorithm for a given simulation run, use the Simulation dialog box in the SystemBuild Analysis Menu or the keyword in the sim function call:

```
y = sim(model,t,u,{ialg = alnumber})
```

11.6.2 Overview of the Algorithms

SystemBuild currently provides nine different integration algorithms. For numerically well-conditioned and non-stiff problems almost any of the algorithms can be expected to yield reliable answers, although the execution times will vary.

The process of “integrating” a system model is conceptually based on discretizing the differential equations that represent the model. That is, $\dot{x} = f(x, t)$ is replaced by a difference equation approximating the underlying continuous differential equation up to a certain order. The continuous variables x and t are replaced by their discrete equivalents x_n and t_n , while \dot{x} is substituted by $\frac{\Delta x}{\Delta t}$. Literally implementing this procedure yields Euler's method.

Euler Integration Method

Euler integration is an explicit, first-order method with one function evaluation per step:

$$x_{n+1} = x_n + hf(x_n, t_n) \text{ where } h = t_{n+1} - t_n$$

Note that the stepsize, h , is taken from the time vector supplied by the user.

This method is equivalent to approximating the area under the solution curve with a series of rectangles, as shown below.

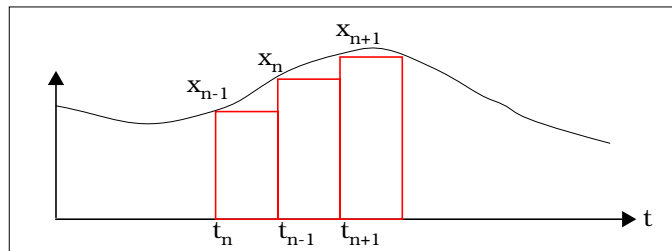


FIGURE 11-7 Forward Euler Integration

Euler's method is computationally inexpensive, but in practical applications it has a major drawback. As seen from the equation, this method corresponds to a simple linear extrapolation with a local truncation error of $O(h^2)$. Therefore, h must be made very small to obtain reasonable accuracy. Unfortunately, reducing the step-size increases the effect of round-off errors and compromises the computational accuracy and speed.

Second Order Runge-Kutta (Modified Euler) Method

This method requires two function evaluations per step, using the following equations:

$$\begin{aligned} k_1 &= hf(x_n, t_n) \\ k_2 &= hf(x_n + k_1, t_n + h) \\ \overline{x}_{n+1} &= x_n + \frac{k_1 + k_2}{2} \end{aligned} \quad \text{Eq. 11-3}$$

The second order Runge-Kutta method improves on the Euler method by fitting trapezoids under the solution curve instead of rectangles, as seen below. The local truncation error is $O(h^3)$.

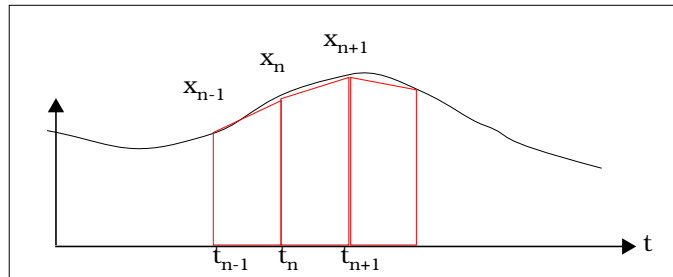


FIGURE 11-8 Second order Runge-Kutta Integration

Fourth Order Runge-Kutta Method:

The fourth order Runge-Kutta integration is an explicit method with four function evaluations per step:

$$\begin{aligned}k_1 &= hf(x_n, t_n) \\k_2 &= hf\left(x_n + \frac{k_1}{2}, t_n + \frac{h}{2}\right) \\k_3 &= hf\left(x_n + \frac{k_2}{2}, t_n + \frac{h}{2}\right) \\k_4 &= hf(x_n + k_3, t_n + h)\end{aligned}\tag{EQ. 11-4}$$

The solution is computed as:

$$x_{n+1} = x_n + \frac{k_1 + 2k_2 + k_4}{6}$$

The fourth order Runge-Kutta method has a truncation error of $O(h^5)$. This is a suitable method when not much is known about the nature of the problem or the solution. Although this algorithm has proven reliable for many types of problems and is widely used, computationally it is not among the most efficient methods, because it does not have any stepsize control or order adjustments.

Fixed-Step Kutta-Merson Method

The fixed-step Kutta-Merson method improves on the fourth order Runge-Kutta method by adding a fifth evaluation step. This method is more accurate than the fourth order Runge-Kutta algorithm with a slight trade-off in computational speed.

It is implemented with the following equations:

$$\begin{aligned}
 k_1 &= hf(x_n, t_n) \\
 k_2 &= hf\left(x_n + \frac{k_1}{3}, t_n + \frac{h}{3}\right) \\
 k_3 &= hf\left(x_n + \frac{k_2 + k_1}{6}, t_n + \frac{h}{3}\right) \\
 k_4 &= hf\left(x_n + \frac{3k_3 + k_1}{6}, t_n + \frac{h}{2}\right) \\
 \tilde{x} &= x_n + \frac{4k_4 + 3k_3 + k_1}{2} \\
 k_5 &= hf(\tilde{x}, t_n + h) \\
 x_{n+1} &= x_n + \frac{k_5 + 4k_4 + k_1}{6}
 \end{aligned}$$

Variable-Step Kutta-Merson Method

As shown in reference 3 on page 11-39, in the fixed Kutta-Merson method shown above, the difference between terms \tilde{x} and x_{n+1} yields an accurate estimate of the local truncation error.

The Variable-Step Kutta-Merson method uses this information to adjust the integration step. The equations are identical to the fixed Kutta-Merson method with additional computations to decide on the stepsize.

The local error is computed as:

$$ERR = \frac{\tilde{x} - x_{n+1}}{5}$$

The maximum stepsize is limited by the time increment $t_{n+1} - t_n$ or dt_{max} . Because of its accuracy, reliability, moderately efficient speed, and its successful performance in a wide range of problems, Variable-Step Kutta-Merson method has been chosen as the default integration algorithm in SystemBuild.

Stiff System Solver (DASSL)

Two types of problems are usually difficult or impossible to solve with the conventional integration algorithms:

- Stiff systems, which have both very fast and very slow dynamics,
- Differential algebraic equations (DAEs).

SystemBuild has an implicit stiff system solver, DASSL, that can handle both types of problems. In stiff systems, conventional algorithms will not be able to capture both the fast and the slow dynamics present in the same system.

DAEs occur when the SystemBuild model results in an implicit system; a system with algebraic loops or implicit UserCode Blocks. The simulator analyzer detects algebraic loops and informs you of their existence with the message `There are algebraic loops in the system`. If you attempt to solve such a system with one of the explicit methods, a delay is automatically added to the model to resolve the algebraic loop. (When such a system is analyzed, SystemBuild provides a message indicating the location of the delay to be inserted.) This may not always be desirable because you do not have control over where the delay is inserted, and if the loop gain is greater than unity, the system becomes unstable.

The Implicit Stiff System Solver applies a Newton-Raphson solver at each time step, and attempts to find an operating point consistent with the user-supplied initial conditions. This process will succeed in most instances. If an operating point cannot be found, however, this will mean that either the problem, or the given initial conditions, are ill-posed (physically unrealizable), or the true solution cannot be reached from the current estimate of its value. DASSL will present one of the following messages when this occurs:

```
DASSL-- AT TIME (=...) AND STEPSIZE H (=...) THE ERROR TEST FAILED
REPEATEDLY OR WITH ABS(H) = HMIN.
```

```
DASSL-- AT TIME (=...) AND STEPSIZE H (=...) THE CORRECTOR FAILED
TO CONVERGE REPEATEDLY OR WITH ABS(H) = HMIN.
```

```
DASSL-- AT TIME (=...) AND STEPSIZE H (=...) THE ITERATION MATRIX
IS SINGULAR.
```

In such cases, the SystemBuild model and the initial conditions should be inspected carefully to locate inconsistencies or errors in the model.

DAEs usually arise in systems of equations resulting from dynamic analysis of mechanical systems. Generally, these types of models are a mixture of nonlinear algebraic and differential equations and they are numerically stiff. A system of DAEs has the form:

$$\begin{aligned} g(\dot{x}, x, t) &= 0; & \dot{x}(t_0) &= \dot{x}_0 \\ & & x(t_0) &= x_0 \end{aligned}$$

The equations are numerically solved as follows:

1. The last converged value of the solution x_n is used as an initial guess at the current time point t_{n+1} .
2. \dot{x} is approximated by a backward differentiation formula of order up to 5. The approximation is substituted for every occurrence of \dot{x} to yield a nonlinear algebraic equation.

The nonlinear algebraic equation is solved by a Newton-Raphson iteration method. This method computes a Jacobian of the form

$$J = \frac{\partial g}{\partial x} + c \frac{\partial g}{\partial \dot{x}}$$

where the constant c is determined by the backward differentiation formula. The Newton-Raphson iteration equation

$$x_n^{k+1} = x_n^k - J(x_n^k)^{-1} g(\dot{x}_n^k, x_n^k, u_n)$$

is solved at $t = t_{n+1}$ using the Jacobian calculated above, approximated by the backward differentiation formula, and initial guess computed from a predictor polynomial that fits the past solution curve. If the Jacobian is not invertible, the algorithm will fail.

The algorithm in SystemBuild is based on the DASSL Stiff System Solver developed by Linda Petzold at Sandia National Laboratories. [Section 7.10 on page 7-18](#) discusses the types of systems for which the implicit stiff system solver is suitable. Also see [Computing the Maximum Integration Stepsize in Variable-Step Integration Algorithms](#) for a discussion of stepsize computations.

Variable-Step Adams-Bashforth-Moulton Method

The Adams-Moulton method is implemented with a variable-step, variable-order algorithm (note that “variable-step” refers to the time step, while “variable-order” pertains to the order of the polynomial that is fitted to the solution curve of the differential equation). Since it generally requires only two function evaluations per time step (one for predictor, one for corrector), the execution time is usually faster than other algorithms (except Euler’s method, which requires only one function evaluation).

The Variable-Step Adams-Moulton method is especially suitable for smoother problems with continuous higher derivatives, since it uses the information from the

higher derivatives. The Adams-Bashforth explicit method is used as a predictor, and the Adams-Moulton implicit method is used as the corrector step:

The Adams-Bashforth Predictor step is computed as:

$$\dot{x}_{n+1} = x_n + \frac{h(55f_n + 59f_{n-1} + 37f_{n-2} - 9f_{n-3})}{24}$$

f is the differential equation evaluated at step n :

$$f_n = \dot{x}_n = f(x_n, t_n)$$

The Adams-Moulton Corrector step is computed as:

$$x_{n+1} = x_n + \frac{h(9\dot{f}_{n+1} + 19f_n + 5f_{n-1} + f_{n-2})}{24}$$

In the above equation \dot{f}_{n+1} is the differential equation computed at step $n+1$ using \dot{x}_{n+1} from the predictor step. As before, $h = t_{n+1} - t_n$. The solution is started at the beginning of the simulation with a second order Runge-Kutta algorithm. The local truncation error of this method is $O(h^5)$. A discussion of how the stepsize is adjusted in the Variable-Step Adams-Moulton algorithm is presented in [Section 11.7](#).

QuickSim Method

Explicit fixed-step integration algorithms are inefficient for numerically solving stiff differential equations because the stability of the method depends on the smallest time constant. QuickSim is an explicit, A-stable, fixed-step integration algorithm that is more efficient and has good accuracy for linear or nearly linear systems.

Given: $\dot{x} = f(x(t), u(t))$ with initial condition $x(t_k)=x_k$, a Taylor series expansion for $f(x,u)$ around $x(t_k), u(t_k)$ is obtained:

$$\begin{aligned} \dot{x} = f(x, u) \approx & f(x(t_k), u(t_k)) + \left. \frac{\partial f}{\partial x} \right|_{x(t_k), u(t_k)} (x - x(t_k)) \\ & + \left. \frac{\partial f}{\partial u} \right|_{x(t_k), u(t_k)} (u - u(t_k)) + \text{h.o.t.} \end{aligned} \tag{EQ. 11-5}$$

Defining:

$$\begin{aligned} \mathbf{A} &= \left. \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right|_{\mathbf{x}(t_k), \mathbf{u}(t_k)}, \quad \mathbf{B} = \left. \frac{\partial \mathbf{f}}{\partial \mathbf{u}} \right|_{\mathbf{x}(t_k), \mathbf{u}(t_k)} \\ \mathbf{x}_{k+1} &= \mathbf{x}(t_{k+1}) \\ \mathbf{x}_k &= \mathbf{x}(t_k) \\ \mathbf{h} &= t_{k+1} - t_k \\ \Delta \mathbf{u}(t) &= \mathbf{u}(t) - \mathbf{u}(t_k) \end{aligned}$$

we obtain:

$$\dot{\mathbf{x}}_{k+1} = \mathbf{A} \mathbf{x}_{k+1} + \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{B} \Delta \mathbf{u}(t_{k+1}) - \mathbf{A} \mathbf{x}_k \quad \text{EQ. 11-6}$$

and solving for \mathbf{x}_{k+1} :

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \int_0^{\mathbf{h}} e^{\mathbf{A}(h-\tau)} [\mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{B} \Delta \mathbf{u}(\tau)] d\tau \quad \text{EQ. 11-7}$$

The method is A-stable, with the same stability properties as trapezoidal integration.

To implement the algorithm, a secant approximation of $\Delta \mathbf{u}$ is used:

$$\Delta \mathbf{u}(t) = \frac{\mathbf{u}_{k+1} - \mathbf{u}_k}{\mathbf{h}} t \equiv \mathbf{m}_k t \quad \text{EQ. 11-8}$$

Substituting, we obtain:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \int_0^{\mathbf{h}} e^{\mathbf{A}(h-\tau)} \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) d\tau + \int_0^{\mathbf{h}} e^{\mathbf{A}(h-\tau)} \mathbf{B} \mathbf{m}_k \tau d\tau \quad \text{EQ. 11-9}$$

The final form of the algorithm is:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{I}_1 \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{I}_2 \mathbf{m}_k \quad \text{EQ. 11-10}$$

where $\mathbf{I}_1, \mathbf{I}_2$ are the two integrals above. These integrals are precomputed at the beginning of the simulation.

Local Truncation Error

The method has a local truncation error that is $O(h^2)$. Let the actual solution at time t_k be $x(t_k)$ and the computed solution x_k . Assume that $x(t_k) = x_k$ and write expressions for $x(t_{k+1})$ and x_{k+1} :

$$\begin{aligned} x(t_{k+1}) &= x(t_k) + \dot{x}(t_k)h \\ &= x_k + f(x_k, u_k)h \end{aligned} \tag{EQ. 11-11}$$

and

$$\begin{aligned} x_{k+1} &= x_k + \int_0^h e^{A(h-\tau)} [f(x_k, u_k) + Bm\tau] d\tau \\ &= x_k + \int_0^h \left[I + A(h-\tau) + \frac{A^2(h-\tau)^2}{2!} + \dots \right] [f(x_k, u_k) + Bm\tau] d\tau \\ &= x_k + f(x_k, u_k)h + \frac{1}{2}Ah^2 + \text{h.o.t.} \end{aligned}$$

so:

$$\|x_k - x(t_{k+1})\| = O(h^2)$$

Over-determined Differential Algebraic System Solver (ODASSL)

ODASSL (Overdetermined Differential Algebraic System Solver) is useful in solving DAEs or ODEs with constraints (problems with more equations than state variables). All continuous dynamic systems in SystemBuild are in ODE form. In order to incorporate DAEs into SystemBuild, an Implicit UserCode block (UCB) has been created. The Implicit UCB ([Section 14.2.2 on page 14-2](#)) also allows constraints to be defined.

The equations of motion for multi-body dynamics often result in systems of differential-algebraic equations. These DAEs may sometimes possess additional constraints that the physical system must satisfy. In some cases, these equations can be reduced to explicit form with algebraic manipulations. However, reduction by analytical or numerical methods may require strong simplifications or serious analytical and numerical difficulties. For such problems, formulation and numerical solution of the equations of motion in the DAE or overdetermined DAE form offers the most convenient approach.

Consider the implicit differential equation (IDE):

$$f(\dot{x}, x, t) = 0$$

The system has index 0 if and only if $\frac{\partial f}{\partial \dot{x}}$ is not singular. Note that this means that the equation can be (locally) transformed into an explicit form $\dot{x} = f_2(x, t)$ without any differentiations. If at least one differentiation of the IDE is required to transform the DAE into explicit form, then the DAE is said to have index 1. In general, an index k DAE requires k differentiations to transform it into explicit form.

In order for SystemBuild to solve a DAE with DASSL or ODASSL, the DAE must have an index of 0 or 1. This is because these algorithms are not designed to handle systems of index greater than 1. In particular, DASSL and ODASSL will fail if the Jacobian is singular:

$$J = \frac{\partial f}{\partial x} + c \frac{\partial f}{\partial \dot{x}}$$

For a DAE defined using the implicit UCB, when a simulation is started, the initial conditions $\dot{x}(t_0)$ and $x(t_0)$ must be consistent. They must satisfy the following equation, otherwise the algorithms may fail when the integration process is started.

$$f(\dot{x}(t_0), x(t_0), u(t_0)) = 0 \quad \text{EQ. 11-12}$$

In exactly the same way, any constraint equations that may be part of the implicit UCB should also be satisfied by the initial conditions:

$$g(\dot{x}(t_0), x(t_0), u(t_0)) = 0 \quad \text{EQ. 11-13}$$

When using ODASSL, SystemBuild first calculates the rank of the matrix $\frac{\partial f}{\partial \dot{x}}$. If there are derivatives that do not explicitly appear in the equations, then the equations that are associated with the variables are de-emphasized in the local error calculations. This is usually encountered when Lagrange multipliers are used to formulate the equations of motion for a dynamic system. For an example of this procedure, see [Figure 11-10](#).

The technique used by ODASSL in incorporating the constraints into the DAE is numerically equivalent to the Gear Stabilization technique. If a DAE is integrated without its constraints, the solution tends to “drift away” from the correct answer. The constraints act as corrections that stabilize the solution.

The most common type of DAEs or ODAEs appear in multibody system dynamics, such as vehicles, satellites, and robots. Typically, the DAEs obtained from the Lagrangian formulation yield the following equations of motion for holonomic systems:

$$\begin{aligned} \dot{p} &= v \\ M(p)v &= f(p, v, u) - \frac{\partial}{\partial p}g_p(p)^T\lambda \\ 0 &= g_p(p) \end{aligned} \quad \text{FIGURE 11-9}$$

where:

- p represents generalized position variables
- v represents generalized velocity variables
- M is the inertia matrix
- f is the function of Coriolis, Centrifugal, and gravitational forces and external inputs, u
- g_p represents position constraints
- λ represents the generalized constraint forces, also called Lagrange multipliers

Differentiating these equations shows that the DAE in [Equation 11-9](#) has an index of 3. In order to successfully solve this system in SystemBuild, an index 1 formulation must be obtained (higher index formulations may fail during integration).

Since the constraint $g_p(p)$ is a function of positions, it can be differentiated to obtain constraints on velocity and acceleration:

$$\begin{aligned} g_v(p, v) &= \dot{g}_p(p) = 0 \\ g_a(p, v, \dot{v}) &= \dot{g}_v(p, v) = 0 \end{aligned} \quad \text{EQ. 11-14}$$

Thus, three types of formulations are possible with this example:

1. Unconstrained DAE formulations:

a. Index 3 formulation:

$$0 = \dot{p} - v$$

$$0 = M(p)\dot{v} - f(p, v, u) + \frac{\partial}{\partial p}g_p(p)^T\lambda$$

$$0 = g_p(p)$$

There are $n_p+n_v+n_i$ states, and the same number of equations.

b. Index 2 formulation:

Instead of $0=g_p(p)$, use constraint $0=g_v(p,v)$.

c. Index 1 formulation:

Instead of $0 = g_p(p)$ use constraint $0 = g_a(p, v, \dot{v})$

2. Index 2 formulation, with one constraint:

DAEs:

$$0 = \dot{p} - v$$

$$0 = M(p)\dot{v} - f(p, v, u) + \frac{\partial}{\partial p}g_p(p)^T\lambda$$

$$0 = g_v(p, v)$$

Constraints: $0 = g_p(p)$

3. Index 1 formulation with two constraints:

DAEs

$$0 = \dot{p} - v$$

$$0 = M(p)\dot{v} - f(p, v, u) + \frac{\partial}{\partial p}g_p(p)^T\lambda$$

$$0 = g_a(p, v, \dot{v})$$

Constraints:

$$0 = g_v(p, v)$$

$$0 = g_p(p)$$

The most reliable numerical results are usually obtained from the index-1 constrained formulation (formula 3) above.

EXAMPLE 11-6: Pendulum Example

The above formulations will be demonstrated by the simple pendulum example illustrated below:

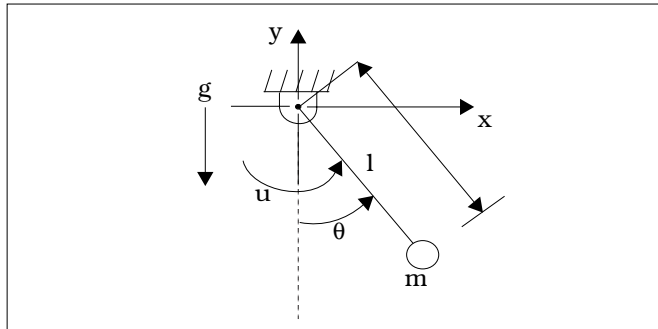


FIGURE 11-10 Pendulum Example Diagram

The pendulum is connected to ground via a pivot. It is assumed to have mass m concentrated at the endpoint with link l having zero mass. An input torque u excites the motion, applied at the pivot point. The equation of motion, using the generalized coordinate θ , is:

$$\ddot{\theta} = \frac{g}{l} \sin \theta + u$$

which is an ODE. For purposes of illustration, the equations of motion will be derived using the coordinates x and y . The position constraint, (the pendulum oscillates on a circle of radius l).

$$g_p = \frac{x^2 + y^2 - l^2}{2} = 0 \quad \text{EQ. 11-15}$$

where:

$$\begin{aligned} m\ddot{x} &= -\frac{y}{l^2} - x\lambda \\ m\ddot{y} &= -\frac{x}{l^2}u - mg - y\lambda \\ 0 &= \frac{1}{2}(x^2 + y^2 - l^2) \end{aligned} \quad \text{EQ. 11-16}$$

The above set of equations constitutes an index-3 unconstrained formulation.

For index-2 and index-1 formulations, we use the velocity constraint,

$$\mathbf{g}_v = \dot{\mathbf{g}}_p = x\dot{x} + y\dot{y} = 0 \quad \text{Eq. 11-17}$$

and the acceleration constraint,

$$\mathbf{g}_a = \dot{\mathbf{g}}_v = x\ddot{x} + \dot{x}^2 + y\ddot{y} + \dot{y}^2 = 0 \quad \text{Eq. 11-18}$$

in place of the position constraint above.

Thus, an index-1, two-constraint formulation of this problem (as in case 3 above) would be as follows:

Let $\begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \lambda \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$ (External input was taken to be zero in this example)

Then:

D.A. (with algebraic level constraints) $\left\{ \begin{array}{l} \dot{x}_1 - x_3 = 0 \\ \dot{x}_2 - x_4 = 0 \\ m\dot{x}_3 + x_1x_5 = 0 \\ m\dot{x}_4 + x_2x_5 + mg = 0 \\ x_1\dot{x}_3 + x_3^2 + x_2\dot{x}_4 + x_4^2 = 0 \end{array} \right.$

Velocity level constraint: $x_1x_3 + x_2x_4 = 0$

Position level constraint: $\frac{1}{2}(x_1^2 + x_2^2 - l^2) = 0$

This example is coded as an implicit UCB in the SystemBuild examples directory. To run the example, go to Xmath and copy the files from the examples directory to your current directory:

```
copyfile "$SYSBLD/examples/pendulum_imp/*"
```

In the Xmath command area type:

```
execute file = "pend_imp.ms"
```

This will load and simulate the implicit UCB. The script prompts you interactively.

Gear's Method

Most Backward Difference Formulations (BDF), for example, DASSL and ODASSL, are based on Gear's method. The original algorithm, DFASUB, was developed at the University of Illinois at Urbana-Champaign. Gear's method is a variable-step, variable-order algorithm and uses a polynomial-based predictor followed by a Newton-based correction at every step (just like (O)DASSL). It was designed to handle a mixture of ordinary differential equations, nonlinear equations, and linear equations.

In Integrated Systems' implementation of Gear's method, the dedicated linear equation handling in the original implementation has been omitted; linear equations are simply handled by the general nonlinear equation solver. In addition, a least-squares extension to the Newton corrector has been implemented to make the algorithm suitable for overdetermined systems. Gear's method is a good choice for implicit systems, and is a good alternative where ODASSL fails.

The main differences between Gear's method and (O)DASSL are summarized herey:

- (O)DASSL bases its polynomial order choice on stability, whereas Gear's uses the largest possible step size as a criterion to determine the order.
- (O)DASSL uses internal logic to determine when re-evaluation of the Jacobian is required, whereas Gear's simply re-evaluates the Jacobian at every time step. Since computation of the Jacobian is a CPU-intensive operation, (O)DASSL runs several times faster than Gear's; conversely, Gear's Jacobian is more accurate.
- ODASSL imposes any auxiliary constraints (that is, any equations that make the system overdetermined) in an exact manner. Gear's imposes them with a weighting function that is implicitly determined by the least-squares enhancement of the Newton step.

To make the associated residuals arbitrarily small, multiply them with a suitably large constant. In cases where the residuals are defined by a Constraint block, this can be done by feeding them through a Gain block first.

References for Gear's Method

R.L. Brown, C.W. Gear — "Documentation for DFASUB - A Program for the Solution of Simultaneous Implicit Differential and Nonlinear Equations", University of Illinois at Urbana-Champaign, report UIUCDCS-R-73-575, July 1973.

C.W. Gear — "Simultaneous Numerical Solutions of Differential-Algebraic Equations", IEEE Transactions on Circuit Theory, CT-18, January 1971.

C.W. Gear — "Numerical Initial Value Problems in Ordinary Differential Equations", Prentice Hall, 1971.

11.7 Absolute and Relative Tolerances

The Variable-Step Kutta-Merson, Stiff System Solver, and Variable-Step Adams-Moulton methods all make use of user-definable absolute and relative tolerances to determine whether the order and/or the stepsize needs to change. Each of these methods has a different technique for local error computation. In the following description of these computations, we first define some terminology:

x_i	Approximate solution for the states at the i_{th} step.
ERR	Approximate local error in the state x .
RELTOL	Relative tolerance (default = 10^{-3}). The <code>reltol</code> variable is specified in the <code>sim</code> function.
ABSTOL	Absolute tolerance (default = $\sqrt{\epsilon}$, where ϵ is the machine epsilon). The <code>abstol</code> variable is specified as a <code>sim</code> keyword.
h	The time step. Just as with the fixed-step methods, this value is originally taken from the time vector supplied by the user, but each variable-step integration algorithm will modify h as part of its process.
$\ \cdot\ $	The Euclidean norm (2-norm) of a vector (except as indicated otherwise).

All dynamic blocks have two fields to specify multiplication factors to scale the `sim` command tolerance parameters `abstol` and `reltol`. These multiplication factors are stored in row vectors and have the default value of one. They are used only by variable step integration algorithms, and can help improve the speed and/or accuracy of the simulation results.

11.7.1 Variable-Step Kutta-Merson Method

The Variable-Step Kutta-Merson algorithm uses the following test:

$$\text{ERR} = \frac{\bar{x} - x_{n+1}}{5}$$

$$\text{If } \|\text{ERR}\| \leq \text{RELTOL} \times \|x\| \text{ or } \|\text{ERR}\| \leq \text{ABSTOL}$$

then the solution for x is good (see the [Fixed-Step Kutta-Merson Method](#) on page 11-18 for definition of \bar{x}). Otherwise, the stepsize h is decreased and the Kutta-Merson equations are recalculated.

11.7.2 Stiff System Solvers (DASSL and ODASSL)

The Stiff System Solver uses a backward differentiation formula to estimate *ERR*. The test to check if the solution is accurate is as follows:

First, let

$$v(i) = \frac{ERR(i)}{RELTOL \times |x(i)| + ABSTOL} \quad (i=1, \dots, N)$$

where N is the number of equations. Then, if $\|v\| \leq 1$ the solution is good. The default norm routine in DASSL is one that finds the Root Mean Square (RMS) norm of the vector:

$$\|X\|_2 = \sqrt{\sum_{i=1}^n v_i^2}$$

Alternatively, the Infinity norm of the vector may be used: $\|v\|_\infty = \max_i |v(i)|$

The Infinity norm is a more conservative bound for the error computation, and choosing this norm will yield more accurate answers. The RMS norm is less accurate, but the algorithm usually executes faster with this norm. The `sim` keyword `dnorm` controls this error computation. `dnorm=1` (RMS norm) and `dnorm=2` (Infinity norm).

It is advisable to use a smaller `reltol` value with the RMS norm to get more accurate answers, since convergence accuracy is not the same as with the Infinity norm. Note that stiff systems are more expensive to solve in terms of computation than other systems. In DASSL the expense is more strongly dependent on the tolerance than it is with other algorithms.

11.7.3 Variable-Step Adams-Bashforth-Moulton Method

In the Variable-Step Adams-Moulton algorithm, the stepsize is chosen so that the local truncation error *ERR* satisfies,

$$|ERR| < h_{n+1} TOL$$

where:

$$TOL = RELTOL \times \|x\| + ABSTOL.$$

In particular, the local truncation error ERR is computed as:

$$\text{ERR} = |h_n(g_{k+1,1} - g_{k,1})\phi_{k+1}(n)|$$

The index k is the order of the method and n is the time step index. Terms $g_{k+1,1}$ and $g_{k,1}$ are coefficients related to past stepsizes, and f_{k+1} is a quantity related to a modified divided difference approximation of the solution derivative at the current and past times.

With this estimate of the error ERR, the next stepsize $h_{n+1} = r h_n$ is chosen so that:

$$r = \left(\frac{\text{TOL}}{2 \times \text{ERR}} \right)^{\frac{1}{k+1}}$$

11.7.4 Computing the Maximum Integration Stepsize in Variable-Step Integration Algorithms

At the end of each integration step, the simulation scheduler decides the stepsize for the next step. It uses five factors to compute the maximum stepsize the variable-step integrators can take, or the actual stepsize the fixed-step integrators will take:

1. The user-defined time vector defines the times when the output is posted. The stepsize can be no larger than the difference between the current time and the next output posting time, as modified by `dtout`, below
2. `dtout`, a `sim` keyword, gives control over the spacing of posted outputs, and thus over the integration stepsize. It may override factor 1, because the integration stepsize can be no greater than the difference between the current time and the next `dtout` posting time.
3. `dtmax`, also a keyword for `sim`, places an absolute upper limit on the stepsize.
4. The sampling of discrete subsystems may affect the length of an integration step. At a given time, the scheduler checks for any upcoming discrete events. The continuous integration stepsize is limited by the difference between the current time and the next discrete event.
5. If there is a state event within any given integration step, the stepsize is reduced to the time instant of the state event.
6. Factors 1 through 5 will define a maximum possible stepsize. The variable-stepsize integrators may internally choose a smaller stepsize in order to satisfy error requirements. fixed-stepsize integrators will use exactly the stepsize dictated by items 1 through 4, above.

11.8 Sample Simulation

In this example, a simple model is simulated with each of the integration algorithms to compare the execution times and the errors in the solutions. The effect of relative tolerance `reltol` on the variable-step algorithms (Variable Kutta-Merson, Stiff System Solver and Adams-Moulton) is demonstrated. Finally, the performance of the Stiff System Solver is tested with the two different error norm computations.

For evaluating and comparing the speed and accuracy of the integration algorithms in SystemBuild, a mass-spring system was modeled with viscous damping proportional to the position multiplied by the velocity and with a nonlinear cubic spring:

$$\ddot{x} + 3xx\dot{x} + x^3 = 0; \quad x_0 = -5; \quad \dot{x}(t_0) = 0.$$

Reference [7] gives the exact solution for the response of the above system as:

$$x(t) = 2 \left(\frac{t + \frac{1}{x_0}}{\left(t + \frac{1}{x_0}\right)^2 + \left(\frac{1}{x_0}\right)^2} \right)$$

The SystemBuild block diagram model for this system is shown in [Figure 11-11 on page 11-35](#). You can copy the data file for this model to your local directory with the following Xmath command.

```
copyfile "$SYSBLD/examples/ialgs/cubic_spring"
```

The remainder of this example compares the various errors exhibited by each integration algorithm, where the error was computed by taking the difference of the ex-

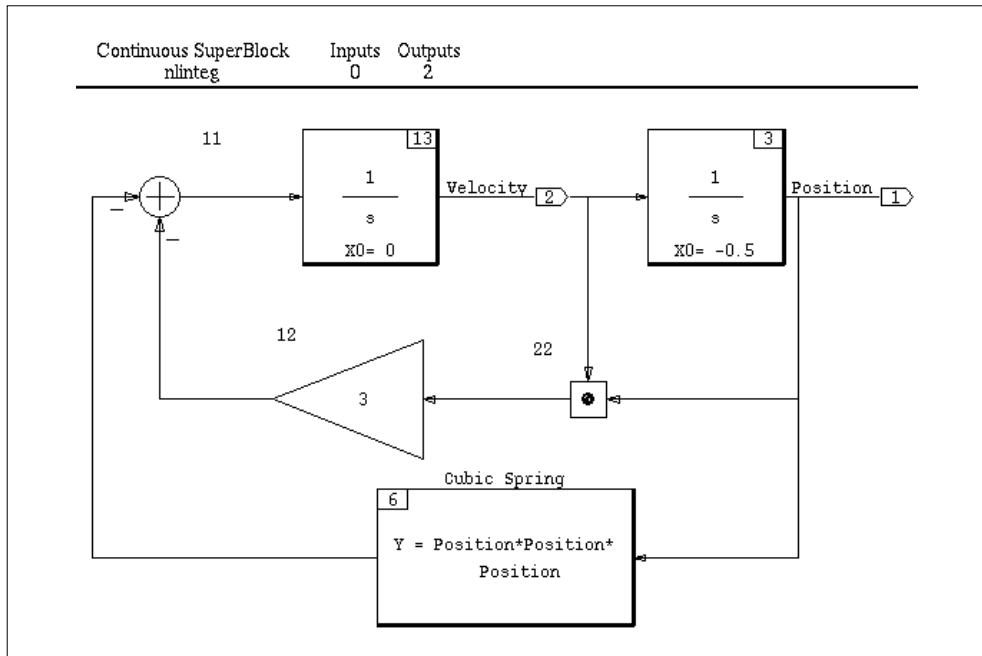
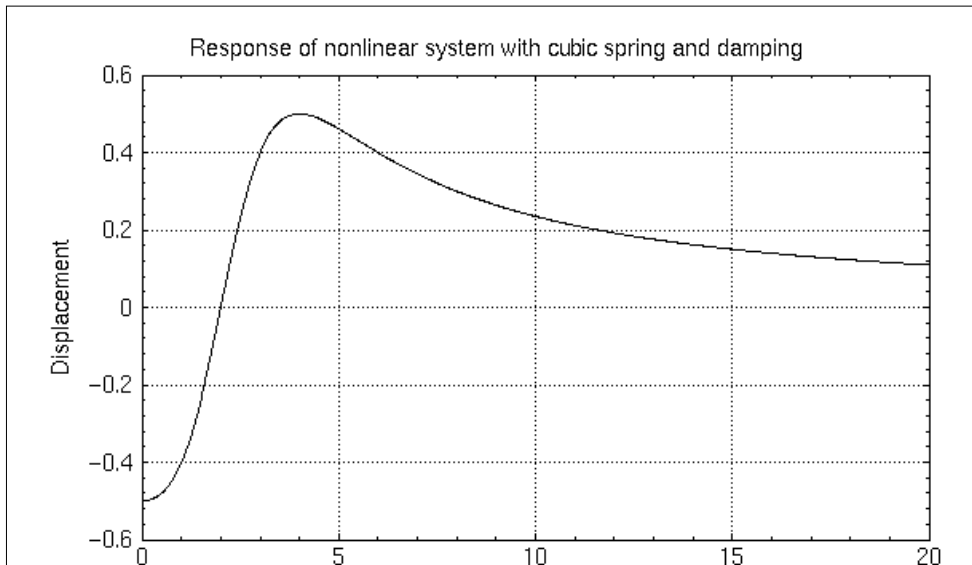


FIGURE 11-11 Block Diagram of the Cubic Spring Model

act value and the value yielded by the algorithm of interest. The position response for this system is shown in [Figure 11-12 on page 11-36](#)



The integration error for the Euler algorithm is shown in [Figure 11-13 on page 11-36](#). This algorithm displays the largest errors, with a deviation of up to 10% of the maximum amplitude of the response.

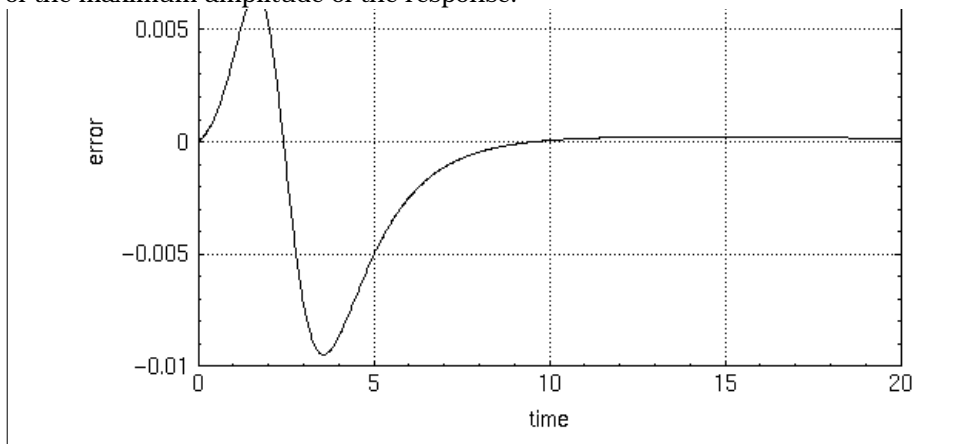


FIGURE 11-13 Errors in Euler Integration

Figures 11-14 and 11-15 present the errors of the remaining six algorithms. In Figure 11-15 on page 11-38, the amount of error displayed by the Stiff System Solver and the Variable-Step Adams-Moulton method are the same order of magnitude as the Second Order Runge-Kutta method.

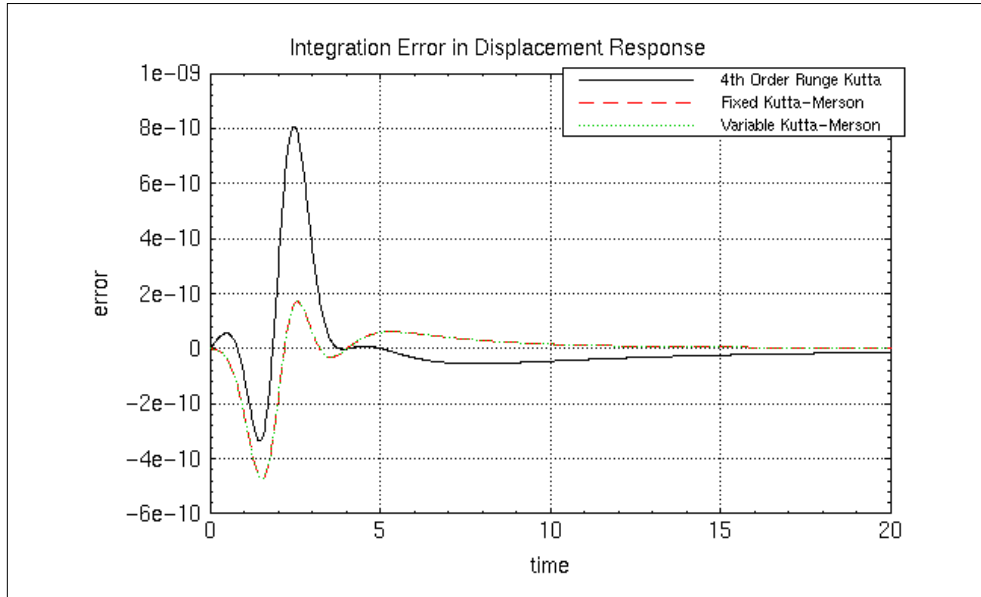


FIGURE 11-14 Errors in Fixed and Variable-Step Integration Algorithms (1)

This example is presented strictly for the purpose of testing the algorithms; these error magnitudes are not significant for most problems. The most accurate method in SystemBuild for the example shown is the Variable-Step Kutta-Merson method. (In general, the Variable-Step Kutta-Merson method is the most accurate among the built-in methods in SystemBuild). All fourth order Runge-Kutta based methods yield excellent performance, as seen in Figure 11-14. Note the vertical scale; the error magnitudes are less than one millionth of the response magnitude.

When `reltol` was varied, it had no observable effect for the Variable-Step Kutta-Merson method for this specific example. Figure 11-16 presents the effect of changing the relative tolerance `reltol` for Variable-Step Adams-Moulton. Figure 11-17 and Figure 11-18 show the effect of `reltol` for Variable-Step Kutta-Merson, and for the Stiff System Solver.

For this example, changing certain options can improve accuracy without adding too much computational burden. Depending on the type of problem, the performance of the Stiff System Solver can be improved by changing the norm computa-

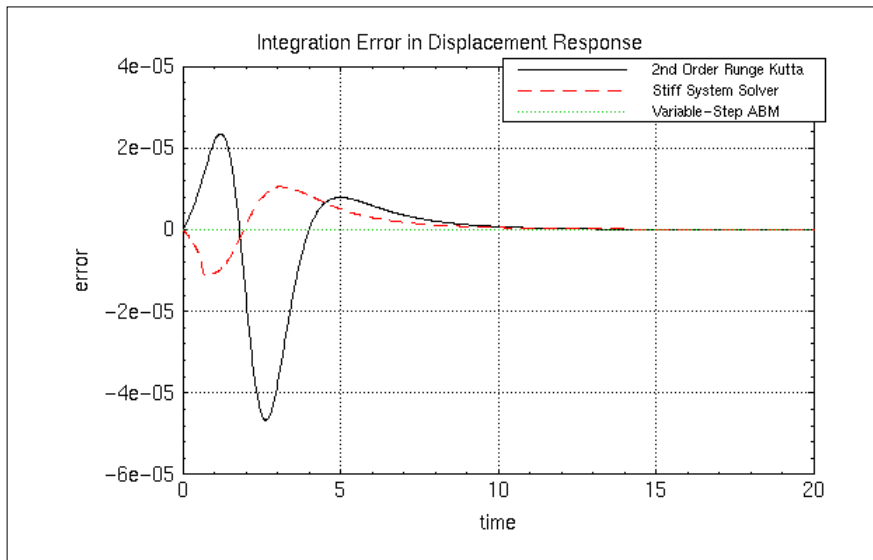


FIGURE 11-15 Errors in Fixed and Variable-Step Integration Algorithms (2)

tion method. [Figure 11-19](#) shows the effect of this change where simulations using the RMS norm ($dnorm=1$) and infinity norm ($dnorm=2$) are plotted. These results may differ slightly, depending on the platform SystemBuild is running on. In particular, the machine epsilon of your specific platform will effect some of these results. (The machine epsilon for these simulations was $\epsilon = 2.2204E-16$.)

In conclusion, it should be pointed out that choosing the right integration method is as much an art as it is a science. As in everything else, experience is the best guide in selecting an integration algorithm.

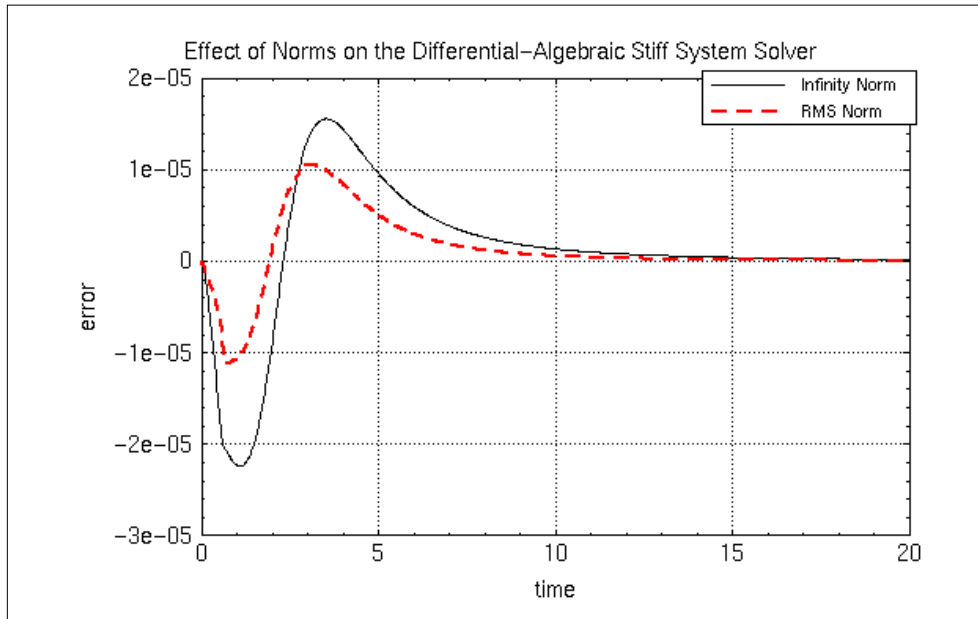


FIGURE 11-19 Norms and the Stiff System Solver

Bibliography

1. Brenan, K.E., S.L. Campbell, and L.R. Petzold, *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations*, North-Holland, New York, 1989.
2. Conte, S.D., and C. deBoor, *Elementary Numerical Analysis*, McGraw-Hill, New York, 1980.
3. Fox, L., *Numerical Solution of Ordinary and Partial Differential Equations*, Pergamon Press, 1962.
4. Gear, C. W., *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice Hall, Englewood Cliffs, N.J., 1971.
5. Hamming, R.W., *Numerical Methods for Scientists and Engineers*, McGraw Hill, New York, 1973.
6. Kahaner, D., C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice Hall, Englewood Cliffs, N.J., 1989.

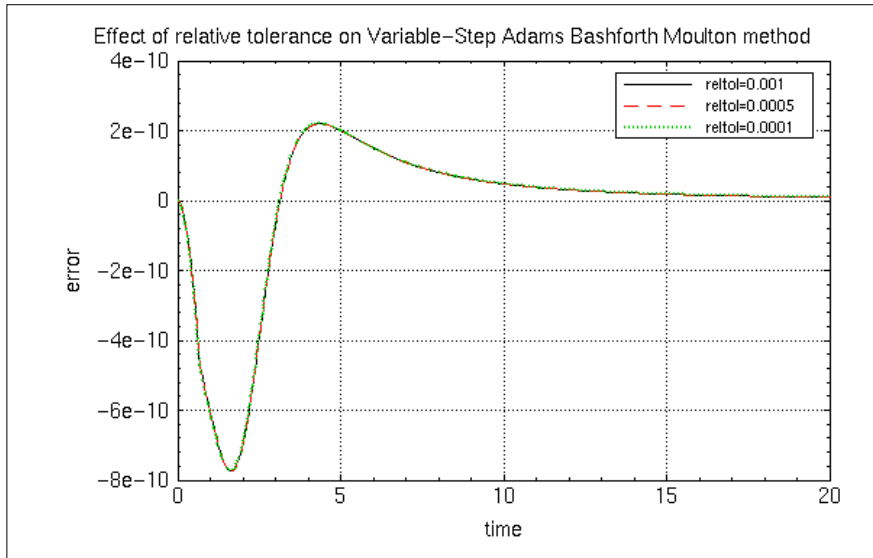


FIGURE 11-16 reltol and Variable-Step Adams-Moulton Method

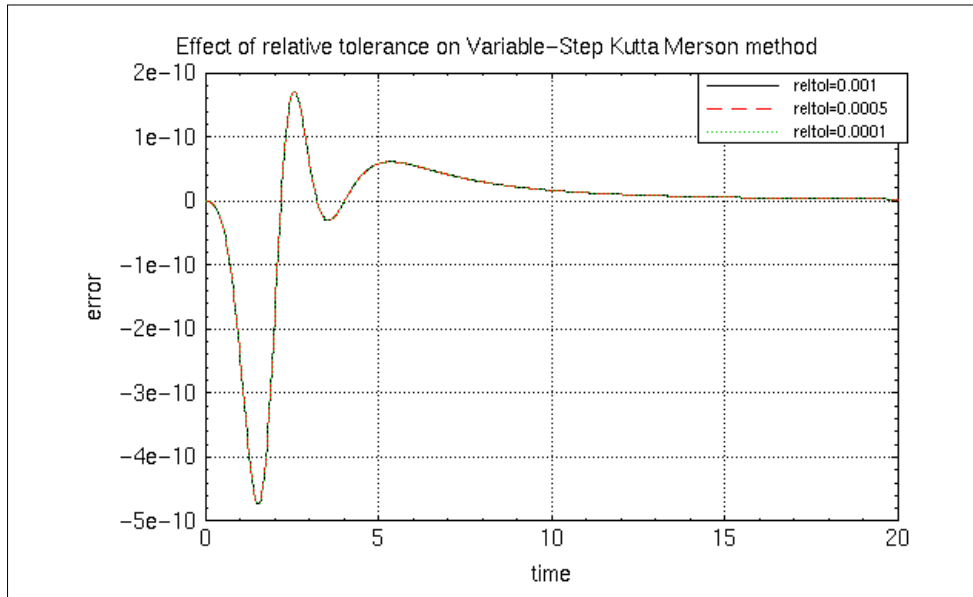


FIGURE 11-17 reltol and Variable-Step Kutta-Merson; All Curves Superimposed

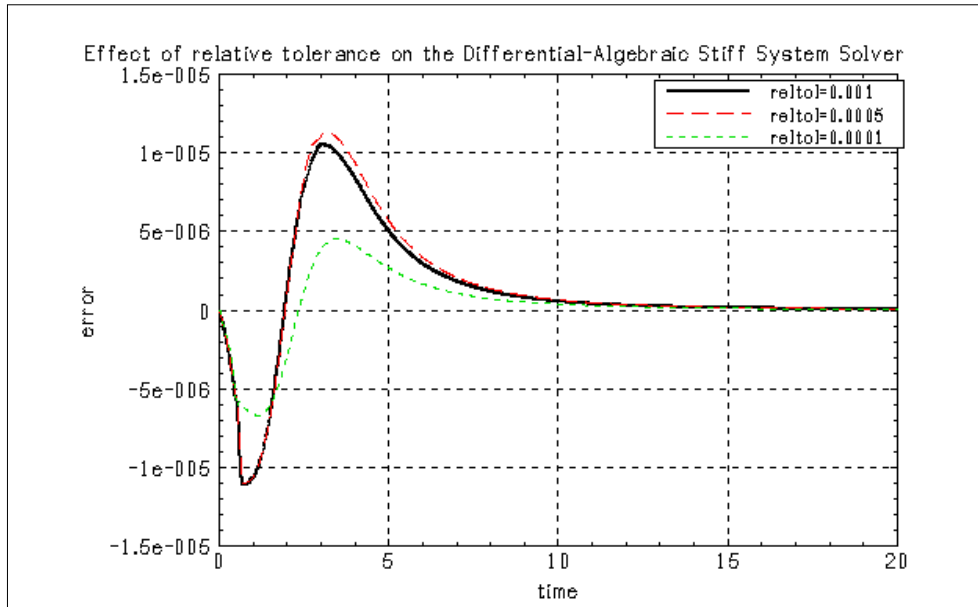


FIGURE 11-18 reltol and the Stiff System Solver

7. McLachlan, N.W., *Ordinary Non-Linear Differential Equations in Engineering and Physical Science*, Oxford University Press, Glasgow, 1950.
8. Milne, E.M., *Numerical Solution of Differential Equations*, Dover Publications, New York, 1970.
9. Petzold, L.R., "A Description of DASSL: A Differential/Algebraic System Solver," in *Scientific Computing*, eds. R.S. Stepleman et al, North Holland, Amsterdam, 1983, pp. 65-68.
10. Press, H.W., B.P. Flannery, S.A. Tevkolsky, and W.T. Vetterling, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge, 1989.
11. Shampine, L.F., and C.W. Gear, "A User's View of Solving Stiff Ordinary Differential Equations," *SIAM Review*, Vol.21, N.1, Jan. 1979.
12. Führer, C., and B. J. Leimkuhler, "Numerical solution of differential-algebraic equations for constrained mechanical motion," *Numerische Mathematik*, Vol 59, pp. 55-69, 1991.

13. Broucke, M., and S. Karahan, "Efficient Method for Integration of Stiff, Nearly Linear Systems," Proc. of the American Control Conference, V.3, pp. 2635-6, San Francisco, California, June 1993.

11.9 State Events

A difficulty in modeling of continuous systems occurs when a system shows any of a class of local discontinuities (*state events*[†]), which may interfere with the operation of continuous integration methods.

To solve this problem, a state-event modeling capability is provided in SystemBuild through the ZeroCrossing block, resettable Integrator blocks, and continuous User-Code blocks (UCBs). The purpose of the feature is to handle discontinuities in state values and switching between system equations. Applications for state events arise in modeling mechanical systems with impact, stiction/friction, and nonlinear systems with variable structures; *i.e.*, potential changes in either the model or the controller.

State events are also used to simulate interrupts associated with Asynchronous Trigger SuperBlocks (see [page 5-14](#)).

The objective of state event modeling is to circumvent attempts by the numerical integration code to integrate over drastic changes (*i.e.*, instantaneous changes of state values in system equations) in the system. Usually such attempts will cause the local error criterion of the integration algorithm to fail, or have convergence difficulties.

These problems are most acute when the time the event occurs is not known a priori. Observe that if state events are handled properly, they do not disturb the numerical integration of the continuous system because they are handled outside of the integrators; in the following discussions this process is referred to as restarting the integrators. After the state event, the operating point of the continuous system is recomputed and the integration can proceed without convergence problems.

SystemBuild provides two ways of dealing with state events: ZeroCrossing blocks and Continuous UserCode blocks (UCBs). If at all possible, we recommend that you try to use ZeroCrossing blocks rather than writing your own UCB to deal with State Events, since ZeroCrossing blocks provide a simpler, consistent and optimized solution to this problem.

† In some literature such events are called *switch events*.

11.9.1 ZeroCrossing Block

The ZeroCrossing block is located on the User Programmed palette. A signal connected to the ZeroCrossing block will be monitored for a sign change[†], and the exact instant of the sign change is located[‡]. Once the zero crossing is found, the output of this block changes from 0 to 1 or from 1 to 0, depending on its previous state (*i.e.* $y(k+1) = 1 - y(k)$ for the k 'th zero crossing). The block output is always initialized to 0 at the beginning of a simulation.

Thus, given a sinusoid signal as input, the block's output will change as shown in Figure 11-20:

```
createsuperblock "sinewave",{inputs=0,outputs=2}
createblock "sinwave",{id=1}
createblock "zerocrossing",{id=2}
createconnection 1,2;createconnection 2,0
t = [0:.01:1]';
y = sim("sinewave",t,{extend,graph});
```

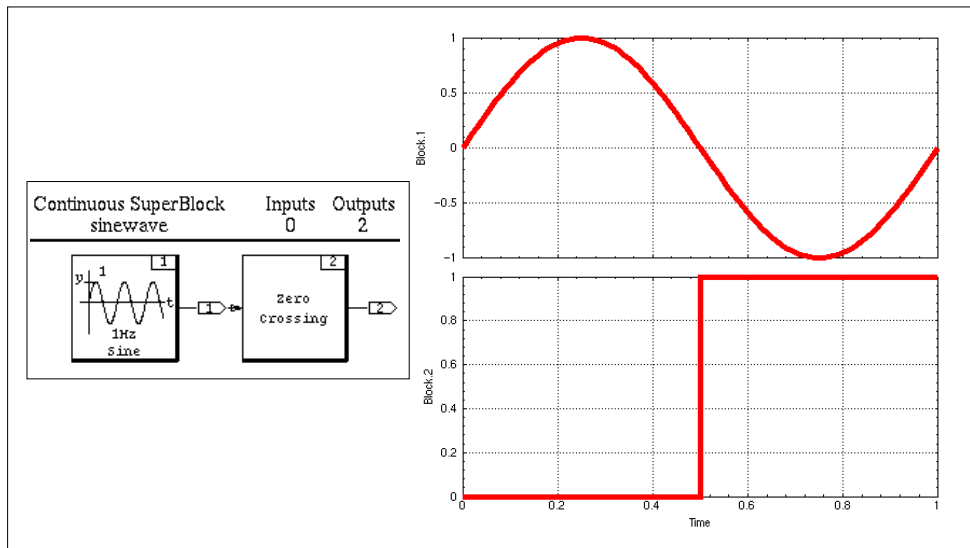


FIGURE 11-20 Zero Crossings of a Sine Wave

† In this block, the sign change is detected only on a change from > 0 to < 0 or from < 0 to > 0 , not on a change to or from 0.

‡ A time point will be generated at the zero-crossing time.

Note that the simulator will always restart (and the operating point will be recomputed) when a zero crossing is found. Therefore, the output of the Zero Crossing Block can be used to detect discontinuities and trigger various events elsewhere in the model (e.g. resetting a resettable integrator to avoid integrating over a discontinuity.) Note that Asynchronous Trigger SuperBlocks whose trigger signal is the output of the ZeroCrossing will execute (and post outputs) before the simulation time is restarted.

In order to offer more flexibility, the Resettable Integrator block has been designed to reset either on a single edge (a transition from *zero* or *negative* to *positive*) or a double-edge (**either** a transition from *zero* **or** *negative* to *positive*, or a transition from *positive* to *zero* or *negative*). See the Integrator block in the online help.

The interface of the Zero Crossing block is simple: the number of input signals monitored is the same as the number of output signals for selection of zero crossings.

11.9.2 Continuous UserCode Blocks

The dialog boxes for both explicit and implicit UCBs include a field for entering the number of state event signals. The mechanism for finding the exact time of an event is as follows:

Associated with each UCB is a user template for defining state events. This code defines two things: the signal to monitor for occurrence of a state event, and the actions to be executed at the instant of the event.

Usually, the state event problem is formulated as follows:

$$\dot{x} = f(x, u, t) \quad \text{EQ. 11-19}$$

$$y = g(x, u, t) \quad \text{EQ. 11-20}$$

$$z = h(x, u, t) \quad \text{EQ. 11-21}$$

where [Equation 11-19](#) and [Equation 11-20](#) describe the system, and [Equation 11-21](#) defines a set of monitoring functions whose roots (or zero crossings) signal a possible state event.

During simulation, the equations defined by [Equation 11-21](#) should be contained in the UCB's `monit` section and are monitored at every integration step. A test is done for a sign change of $z = h(x,u,t)$. If a sign change occurred in the last step, the integration code executes a root solver to find the instant of zero crossing.

Once the time of the event is found, the integrators will take a step just up to the time of the event, and then the event section code described in the template will be executed. This code can reset state values, change model parameters, and/or switch between system equations (see the examples for a demonstration of how this can be done). After the event is executed, the operating point is recomputed with the integrators restarted; i.e., continued from the reset values.

The UCB template divides the user-written program into six sections, each invoked using its own flag. The sections are:

- `init` (perform program initialization activities)
- `state` (perform state updates)
- `output` (perform output updates)
- `monit` (check for state events)
- `event` (perform state event activities)
- `last` (perform program termination activities)

Thus, two flags are provided to be used for state-event simulations: `monit` and `event`. In the `monit` section, the monitor function (i.e., the zero-crossing signal) should be calculated.

The UCB is called with `monit=1` during every integration time step. The event section defines the actions to be taken when a state event is found. When SystemBuild finds a zero crossing in one of the monitor signals, it makes a call to the UCB with `event=i`, where `i` is the i^{th} monitor signal that has signaled a zero crossing. Note that the `i` count starts from 1, not from 0. In the `event` section, states can be reset to new values, or `rpar` and `ipar` values can be set for other purposes, depending on the application.

To help with building UCBs for state events, the following template files are provided:

```
$$SYSBLD/src/usr01.c
$$SYSBLD/src/iusr01.c
```

NOTE: `iusr01.c` is intended for implicit UCBs only. See [Section 14.2.2 on page 14-2](#) for further details.

- In all state-event applications, the activation of the event depends on the precise location of a zero-crossing of the signal being monitored. For this reason, the zero-crossing detection fails if the signal becomes degenerate (i.e. stays at zero

at more than one integration step). Care must be taken to avoid this situation; *i.e.*, you must make sure that the location of the zero-crossing of the signal is unique.

- It is possible to have multiple zero-crossing signals and corresponding events in a model. Each event in the state events blocks will be handled independently of others, even when the zero crossing locations coincide. The new operating point is computed *after* all events have been detected for that time step.
- When the Zero Crossing block is used to simulate a model, every zero crossing is considered to be an event. Only in UCBs are the `monitor` and `event` sections separate.
- Time-based events can easily be handled using the output time feature of the AlgebraicExpression block (*i.e.*, $y = T$), and feeding the signal ($T_{\text{event}} - T$) into a ZeroCrossing block.
- The set of equations simulated by SystemBuild can be switched during a simulation using a ZeroCrossing block in conjunction with some additional logic (*e.g.*, the DataPathSwitch block). Also, within a UCB, user-written code in the event section may activate a flag called by user code in the `states` section when determining which state update equations to use.

Restrictions and Limitations

1. State events are only defined for continuous UCBs.
2. No zero crossings can be detected at time zero.
3. For any given signal, only odd zero-crossings can be detected within an integration step. This means that when an integration algorithm takes a forward time step, if the signal changes signs twice, the zero crossings will not be detected.
4. Degenerate zeros (*i.e.*, successive zero values) will not be detected as zero crossings. Degenerate zeros will result in a failure of the simulation.
5. Fixed step integrators will not calculate the zero crossing instant as accurately as the variable step algorithms.
6. Quicksim (`ialg = 8`) will not detect any zero crossings, because this algorithm calculates its solution based on a linearization at $t = 0$.

11.9.3 Example

This example uses a bouncing ball model with a ZeroCrossing block.

EXAMPLE 11-7: Impact of a Bouncing Ball

In this example, a bouncing ball is modeled. Copy the example data to your local directory:

```
copyfile "$SYSBLD/examples/bouncer_example/bouncingball.dat"
```

The ZeroCrossing block detects the moment when the ball hits the ground. Note that the height of the ball needs to become slightly negative in order to locate a zero crossing. However, the height is reset to zero by the resettable integrator as soon as the impact instant is found, and the simulation is backed up to the impact time. Locating the zero crossing in this way allows the use of larger steps in the time vector.

The conservation of the impulse is modeled by resetting the velocity integrator to the last velocity (multiplied by a restitution coefficient).

When the kinetic energy of the ball becomes too small, the bouncing frequency increases, and the zero crossing becomes degenerate. To represent this, a boundary layer on the position of the ball has been added to the model; see [Figure 11-21](#).

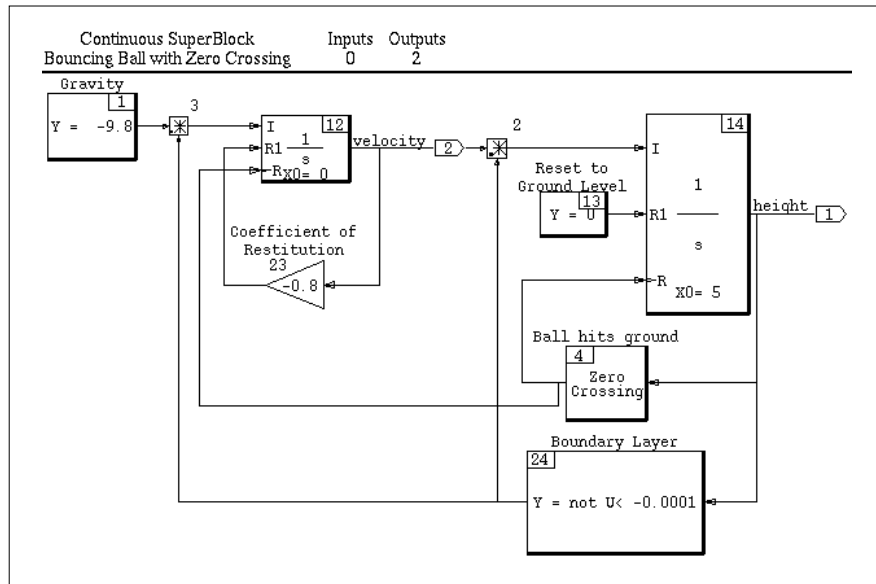


FIGURE 11-21 Bouncing Ball Model

In [Figure 11-22](#), the system was simulated with:

```
t = [0:.05:10]';  
y = sim("Bouncing Ball with Zero Crossing", t, {graph, extend})
```

Note that by using the `extend` keyword option in `sim`, we obtain the exact instants of impact (zero crossings) in the output PDM `y`. This allows a very accurate plot of the bouncing ball simulation.

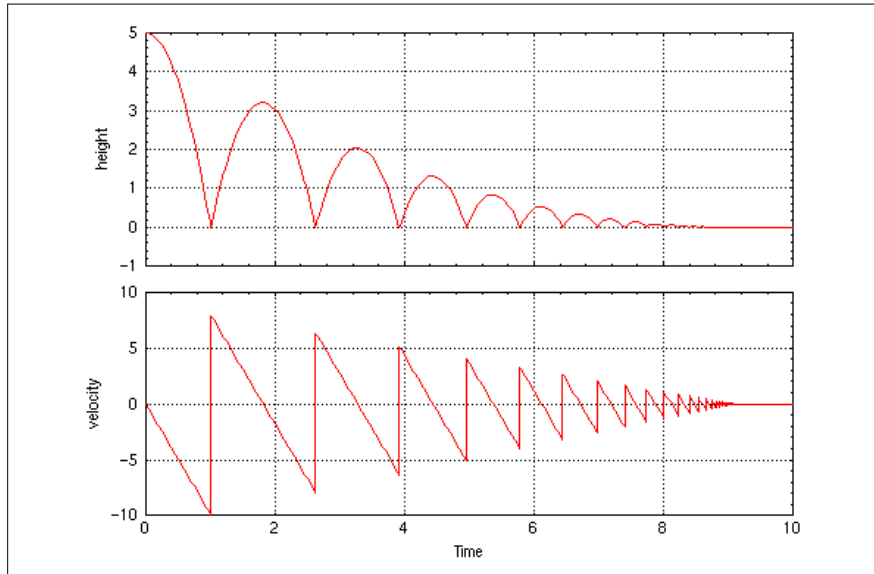


FIGURE 11-22 Plot of Bouncing Ball Example

12

BlockScript

This chapter focuses on the structure and syntax of the BlockScript language. For an explanation of the BlockScript block, see the online help.

12.1 Introduction

BlockScript provides a generalized programming capability for defining SystemBuild blocks for simulation and code generation. The BlockScript block extends the concepts used in SystemBuild's AlgebraicExpression and LogicalExpression blocks.

A BlockScript program allows you to define block inputs, outputs, and parameters, specify their datatypes and dimensions, and write the update equations that process the inputs and parameters to produce the outputs.

12.1.1 The Block Paradigm

This section explains the Integrated Systems block paradigm (Figure 12-1) and shows how the structure of the BlockScript program supports it.

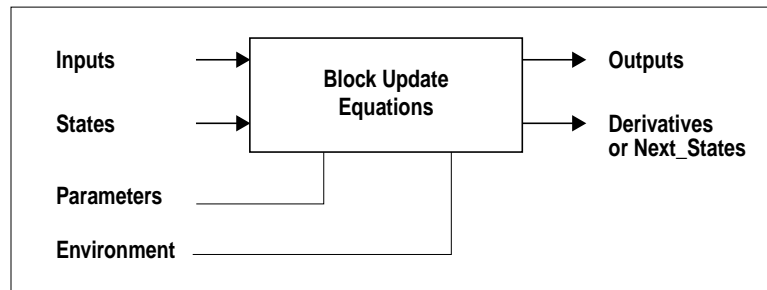


FIGURE 12-1 Block Structure Paradigm (Continuous Shown)

The block update equations are programmed to accept:

- Block inputs
- States (information from the previous cycle)
- Parameters (information from the block dialog box)
- Environment information, such as time and certain universal and platform-dependent constants

The block update equations produce two types of outputs:

- Block outputs
- State derivatives (Continuous) or Next_States (Discrete)

12.1.2 BlockScript Program Structure

BlockScript programs employ two kinds of variables, block variables and local variables:

- Block variables correspond to data flow and parametric entities in the block dialog for the block being defined. These are the inputs and outputs of the block update equations illustrated in [Figure 12-1 on page 12-1](#).
- Local variables take their datatyping and meaning from the program context in which they are defined.

WARNING: Local variables cannot be used to pass data between the INIT, OUTPUT, or STATE phases. Recompute the data, use a parameter to store the data, use a state variable, or use a block output.

All the inputs and outputs in [Figure 12-1](#) are defined using lists of block variables, and the updating of the outputs and derivatives is performed using the equations in the BlockScript program. The design of the BlockScript program lets you choose variable names that are descriptive in the context of the block equations.

The general structure of a BlockScript program is as follows:

```
# Block variable names           : Category: (Var1, Var2, ...);
# Datatype and dimensions definitions: Type Var (Dimension);
# Block output update equations  : ...
```

1. The structure of a block variable name definition is:

```
Category: (Var1, Var2, ...);
```

Categories are reserved keywords in BlockScript; a complete list of the supported categories is shown in [Section 12.2.1](#).

2. BlockScript supports three data types: integer, float, and logical. The format of a datatype and dimension definition is:

```
Type Var (Dimension)
```

Block variables must receive a definition in this section; you may also assign a datatype and dimension definition to any local variables.

3. Block output update equations. The format of equation statements are given in [Section 12.3](#).

The three sections must be presented in the order shown. The sections are defined by the formats of the statements in context, such that the first datatype and dimension definition marks the end of the block variable names, and the first statement with the format of a block output update equation marks the end of the datatype and dimension definitions.

In the following example, a simple addition block is programmed with two input variables, A and B, and one output variable, C. Also note that the input list implies an ordering, which will appear in the block defined from this code. A is the first input while B is the second input.

```
Inputs: (A, B);
Outputs: C;
C = A + B;
```

12.2 BlockScript Variables

BlockScript variables include block variables and local variables. Note the following:

- Language operators, and function names, and keywords are case-insensitive.
- Variable names are case-sensitive.
- Environment variables must be fully capitalized.

12.2.1 Block Variable Declarations

Block variables are declared with the following list construct:

```
Category: (Var1, Var2, ...);
```

- Category can be one of the predefined list category names in [Table 12-1](#).
- If there is only one variable in the list, parentheses are not required. If there are no variables in the list, then the parentheses are required but contain nothing.
- For all lists, order is significant. The first variable maps to the first input/output/state, etcetera, and the last variable maps to the last element.

The name list mechanism can be dispensed with altogether if the programmer is willing to accept the default names for each category, as listed in [Table 12-1](#).

TABLE 12-1 Default Variable Names

List Category Name	Default Var Name	Definition
Inputs	u	A list of input variable names.
Outputs	y	A list of output variable names.
States	x	A list of state names.
State_Derivatives	xdot	A list of state derivatives. This declaration is only valid for continuous dynamic blocks. The dimension of this list must agree with the dimension of the States list.
Next_States	xnext	A list of next state variable names. This declaration is only valid for discrete dynamic blocks. The dimension of this list must agree with the dimension of the States list.
Parameters	(none)	A list of parameter names. This list implies order; if AutoCode maps the variables into Rpar and Ipar vectors, mapping will duplicate the order in the Parameters list. If a list of parameters is supplied, additional fields will be added to the block dialog in the order specified in the Parameters list.
Environment	See Section 12.3.5	The SystemBuild and AutoCode environment provides predefined variables that can be imported into the BlockScript code through this Environment list. The variables in Section 12.3.5 are available regardless of environment (simulation or generated code).

For example, a simple signal generator might be coded.

```
Environment: TIME;
Inputs: ();
Outputs: y;
y= Sin(TIME);
```

```
Parameter Phi:
Parameter Theta:
Parameter Psi:
```

You can enter hard default values for these parameters in the dialog box. Also, you can provide a %Variable name for each parameter. The maximum number of parameterized variables for a BlockScript block is 10.

12.2.2 DataTypes and Dimensions

BlockScript supports three datatypes: float, integer, and logical. Datotyping can be performed according to the rules in [Table 12-2](#).

TABLE 12-2 Datotyping Rules

Variable	Default Type	OK to Datatype Variable?
Inputs	float	Yes
Outputs	float	Yes
States	float	Yes
State_Derivatives	float	Yes; type must agree with States.
Next_States	float	Yes; type must agree with States.
Parameters	(user-defined)	Yes, required
Environment	(predefined)	No. Note, TIME is always float scalar.

If local variables are not explicitly datotyped, then they are defined as scalar variables whose datatype agrees in context with the first statement that defines them in the BlockScript code. Consider [Example 12-1 on page 12-6](#), which defines a non-linear breakpoints block.

EXAMPLE 12-1: BlockScript for Non-linear Breakpoints Block

```

Inputs: U;
Outputs: Y;
Parameters: (UBrk, YBrk);
Float U, Y(:);
Float UBrk(:), YBrk(Y.size,:);
J = 1;
K = UBrk.size;
Uval = U;
While J < K-1 Do
    M = (J + K)/2;
    If Uval < UBrk(M) Then
        K = M;
    Else
        J = M;
    EndIf;
EndWhile;

Alpha = (Uval - UBrk(J)) / (UBrk(K) - UBrk(J));
For I = 1:Y.size Do
Y(I) = (1.0-Alpha)*YBrk(I,J) + Alpha*YBrk(I,K);
EndFor;

```

- The input, U, is a scalar.
- The output, Y, is a vector that has a wildcard dimension (the colon operator); see [Wildcard Dimensions and Dialog Imported Information](#). Its dimension will be imported from the Outputs field in the block dialog. The breakpoints are specified as parameters with two variables, UBrk and YBrk. Parameters can also be given wildcard characters for their dimensions, so that they can be scaled from user inputs in the block dialog.
- A variable's size can be used as a dimension in any name space except Environment (because environment variables have predefined sizes). Note the use of the compile time variable `Y.size` to obtain the current dimension of a vector variable. For matrix variables, you can use the variables `Var.rows` and `Var.columns` to specify a dimension.
- Parameters and local variables can be scalars, vectors, or matrices.
- Inputs, outputs and states can use names that are scalars or vectors.

Wildcard Dimensions and Dialog Imported Information

The `:` wildcard character can be used for any parameter dimension. This is possible because the dimensions are not constrained in the block dialog box. For example:

```
Parameters: (F,G,H);
Float F(:), G(:, :), H(:, :);
```

The `:` wildcard character may also be used for dimensioning Inputs, Outputs, States, and/or State_Derivatives (Next_States if discrete).

A colon wildcard can only be used with a signal name if there is just one name in the list. This is because the block dialog provides only the total number of signal values and does not accommodate a list of names.

```
Inputs: U;
Float U(:);
```

Any variable's total size, (`Var.size`), number of rows, (`Var.rows`), or number of columns, (`Var.columns`) may be used as a dimension for any other variable (excluding itself). The size of a variable may be used before the size and datatype are defined. For example,

```
Inputs: U;
Float Workspace(U.size, Pivot.size);
Float U(:), PivotVector(5);
```

Other dimensions can be described by `Var.size` for vectors and `Var.rows` and `Var.columns` for matrices. This constrains the specified dimension to follow the dimension of `Var`. Any constrained dimension (either hardcoded or described by `Var.size`), is not free; it cannot be changed in the block dialog.

Dimensions that are specified with the wildcard character can later be changed by the user from the BlockScript block dialog. If the user decreases the dimension, information referenced outside that dimension will be discarded. If the user increases the dimension, the last value of the vector is repeated to extend the vector. If you extend a matrix, the extended area filled with zeros.

The variables, `Var.size`, `Var.rows`, `Var.columns`, as well as a generic casting function, `Var.type()` can be used in the code.

```
For I = 1:A.rows Do
    For J = 1:A.columns Do
        Y(I) = Y.type(A(I,J)*U(J));
    EndFor;
EndFor;
```

`Var.size` has special meanings for differing variable shapes. For scalars it is one, for vectors it is the dimension specified (wildcarded or not), and for matrices it is the product of the two specified dimensions.

Method for Implied Datotyping

As stated before, not all datatypes must be explicitly specified. In [Example 12-1 on page 12-6](#), the variable `J` is an integer because it is assigned the integer literal `1`. (The decimal point and/or `E` in scientific notation are used to specify float literals). `K` is also an integer because `UBrk.size` returns an integer. `M` is an integer because $(J+K)/2$ evaluates to an integer. `Uval` and `Alpha` are float variables because they are evaluated with float expressions. `I` is an integer because `1:Y.size` is an integer range expression. (It is possible to code floating point `For` loop ranges such as `For Angle = -Pi : Pi/10.0 : Pi Do`.)

Integers must be used so that `AutoCode` can generate efficient algorithms for the blocks. If you mix floats and integers in expressions, `BlockScript` will promote the entire expression to float at the operation that mixes the two datatypes. The result of the operation is then a float, and the float datatype is then propagated through the outer expressions. Consider the following equation.

```
Integer I, J, K;  
I = (J + K)*3.14 + 255 / (L + M);
```

Both $(J + K)$ and $255 / (L + M)$ are evaluated as integer expressions. Furthermore since integer division causes truncation towards zero, $255 / (L+M)$ will contain that truncated value. Next, multiplication by `3.14` makes $(J + K)*3.14$ a float which when added to the integer expression $255 / (L+M)$ makes the resultant right-hand-side (RHS) of the equation a float expression. Finally since `I` is an integer, the RHS float expression is again truncated towards zero before storing the result in variable `I`. In short, the only difference between integer and float is the implied truncation towards zero when dividing two integers or when assigning an integer with a float expression. It is more disciplined and left to the user to explicitly mix arithmetic with the casting functions `Integer()` and `Float()`.

It should also be noted that although `logical` is a special form of integer, and although the C language treats both the same in its syntax, other languages such as `FORTTRAN` do not. Therefore, logical variables must be declared and used when they are intended to hold logical results; see [Example 12-2 on page 12-9](#).

EXAMPLE 12-2: Declaring Logical Variables

```

Logical Negative, InRange, OK;
Negative = A < 0.0;
InRange= A > B & A < C;
OK = InRange & ! Negative;
If OK Then
...
EndIf;

```

BlockScript Datatypes and Code Generation

If you intend to generate code, only the float datatype should be used within the BlockScript block. In most situations, if the Typecheck feature is disabled, all signal types are forced to be float. The BlockScript block is an exception; its datatypes are unchanged; any mismatch has no effect on simulation.

12.3 The BlockScript Language

As defined in BlockScript, a primitive block is an entity that accepts inputs and produces outputs at times that are scheduled by the simulation or AutoCode generated scheduler.

12.3.1 Operators and Precedence

BlockScript's precedence of operators is similar to those in the C language. One difference is that, in C, the logical datatype is an integer and therefore logical operators combine integer values. In BlockScript, logical and integer data are different. BlockScript makes a distinction between numeric equivalence, ==, and logical equivalence, ~, but places them next to each other in the table to provide the same precedence as in C. However, C puts the precedence for Bitwise XOR, ^ in C, between AND and OR. Since XOR is also NEQV, !~, BlockScript places it with EQV, ~. [Table 12-3 on page 12-10](#) illustrates the BlockScript operators and precedence.

12.3.2 Assignment Statements and Expressions

A variable may be assigned using an expression to the right of the assignment operator. By default all block variables are float scalar data. If a local variable is not explicitly datatyped, then it is automatically assigned the datatype of the right-hand-side expression that first defines it within its function body. Once you have chosen the datatype, integer variables may be assigned to float expressions and vice-versa.

TABLE 12-3 Operator Precedence

Operator Type	Operators	Associativity	Precedence	
Primary	(), (Subexpressions, Functions, Arrays)	Left-to-Right	Highest 	
Power	^ or ** (Power)	Left-to-Right		
Multiplicative	* (Multiply) / (Divide)	Left-to-Right		
Unary	+ (Unary Plus) - (Unary Minus) ! (Not, Complement)	Right-to-Left		
Additive	+ (Plus) - (Minus)	Left-to-Right		
Shift	<< (Shift Left) >> (Shift Right)	Left-to-Right		
Range	: (Define Range)	Right-to-Left		
Relational	< (Less Than) <= (Less Than or Equal) > (Greater Than) >= (Greater Than or Equal) <> (Not Equal) == (Equal)	Left-to-Right		
EQV	~ (Equivalence, Eqv, Nxor) !~ (Not Equiv, Neqv, Xor)	Left-to-Right		
AND	& (And, Intersection) !& (Nand)	Left-to-Right		
OR	(Or, Union) ! (Nor)	Left-to-Right		
assignment	= (Variable Assignment)	Right-to-Left		Lowest

Logical variables can only be assigned relational or logical expressions. There are five kinds of expressions:

Arithmetic Expressions

Arithmetic expressions use only arithmetic operators. These expressions use the operators listed in [Table 12-3](#). Note that the bitwise operators use the same symbols as the logical operators. Bitwise operators only take integer expressions for their operands.

Relational Expressions

Relational expressions compare two arithmetic expressions to form a logical result. Relational expressions use the following operators:

< <= > >= <> ==

Logical Expressions

Logical expressions combine logical expressions and/or relational expressions with logical operators to produce logical results. Logical operators are as follows:

primary:	()	
unary:	!	
logical eqv:	~	!~
logical and:	&	!&
logical or:		!

Range Expressions

Range expressions combine arithmetic values or expressions with the Define Range operator (:) to specify a set of values. A range expression has the following format:

```
Range := Start : Increment : End
```

If the increment is omitted then its value is 1. Ranges may be either integer or float.

Set Expressions

Set expressions combine range expressions with the Union operator (|) to define sets of values. If ranges are used with the float datatype, sets are composed with a discrete number of continuous regions of values.

The syntax for a Set expression is shown below:

```
Set := Region | Region | Region | ...
Region := { Range | Value }
```

The vertical bar enclosed in braces in the syntax represents a choice between the enclosed identifiers. The vertical bar used in the Set description is the Union operator, |, and is required in the syntax. The Intersection operator, &, and parentheses (), are not used in Set expressions. All identifiers describing a Set must be the same datatype. The set expressions are used in the Select clause.

12.3.3 Looping Constructs

BlockScript provides four constructs for looping and decision-making.

For Loop

The For loop can be used when the body of the loop should be executed a known number of times. BlockScript describes the loop counting as an arithmetic progression with a Start, Increment and End value. These values can be either integer or float, but should be consistent. The syntax for the For loop is shown below:

```
For LoopVar = LoopRange Do
    LoopBody;
EndFor;
```

The LoopBody is any number of BlockScript statements. The LoopRange is in either of the following two formats:

```
Start : End
Start : Increment : End
```

The default Increment is 1.

While Loop

The While loop is used when the loop body should be executed until some condition is met. The syntax for the While loop is shown below:

```
While LogicCondition Do
    LoopBody;
EndWhile;
```

LoopBody is any number of BlockScript statements. The LogicCondition is any valid scalar logical expression.

NOTE: The expression may use any number of previously defined variables. However, BlockScript all input variables used inside While loops must be scalars or subscripted with literals.

If Clause

The If clause is used to conditionally execute one of several bodies of code depending on a True evaluation of some condition. The If clause syntax is shown below:

```
If LogicCondition Then
    ConditionBody;
ElseIf LogicCondition Then
    ConditionBody;
Else
    ConditionBody;
EndIf;
```

There may be any number of ElseIf clauses. ConditionBody can consist of any number of BlockScript statements. LogicCondition is any valid scalar logical expression; it may use any number of previously defined variables. Both the ElseIf and Else clauses can be omitted.

Select Clause

The Select clause is used to conditionally execute one or more bodies of code depending on a variable whose value matches the values in the corresponding sets specified with Case statements. The following illustrates the syntax for the Select clause:

```
Select ChoiceVar ClauseForm
Case ConstSet
    CaseBody;
Case ConstSet
    CaseBody;
Otherwise
    CaseBody;
EndSelect;
```

ChoiceVar	ChoiceVar is any integer or float variable previously defined.
ClauseForm	<p>BlockScript provides two forms of Select clauses. In the above syntax, <i>ClauseForm</i> can be either <i>OneOf</i> or <i>Allof</i>. There must be at least one case in the Select clause.</p> <ul style="list-style-type: none"> ■ The <i>OneOf</i> keyword instructs BlockScript to execute only the first Case that matches. ■ The <i>Allof</i> keyword allows BlockScript to execute all cases that match. ■ The optional <i>Otherwise</i> case will only be executed if no cases match. Note that at the end of each <i>CaseBody</i> there is an automatic break to the next Case that matches (if <i>Allof</i>) or <i>EndSelect</i> (if <i>OneOf</i> or in <i>Otherwise</i>). This case can be omitted.
CaseBody	The <i>CaseBody</i> appears in any number of BlockScript statements.
ConstSet	<p><i>ConstSet</i> is a set of values specified by scalar constant values and constant ranges.</p> <p>A vertical bar () represents a choice between one or more identifiers. In the case of <i>ConstSet</i>, it functions as the Union operator.</p> <ul style="list-style-type: none"> ■ The datatypes for all <i>ConstSets</i> must agree within the set and must be the same type as <i>ChoiceVar</i>. ■ If the <i>ChoiceVar</i> is type float, a range cannot be used. For example 1.0:3.0 will not be accepted. To achieve the same thing, specify 1.0 2.0 3.0. <p>The <i>ConstSet</i> syntax shown below is recursive, such that subsets within <i>ConstSet</i> can be ranges or values, specified as floats or integers, if appropriate.</p> <pre> ConstSet Subset Subset Subset ... Subset Range Value Range StartValue : EndValue Value IntegerValue FloatValue </pre>

Exit Statement

The *Exit* statement is used to break out of loops. If used in a *For* loop or *While* loop, execution resumes just after the matching *End* keyword. Note that unlike the C language's *break* statement, *Exit* is not used to break out of *Case* statements.

Iterate Statement

The `Iterate` statement is used to invoke the next iteration of the corresponding current `For` or `While` loop. Execution resumes where the loop variable is incremented in `For` loops or where the logical condition is tested in `While` loops.

12.3.4 Functions

This section describes all of BlockScript intrinsic functions.

`Var.rows`, `Var.columns`, `Var.size`

These variables return the size of a variable. `Var.rows` and `Var.columns` should be used on matrix variables while `Var.size` should be used for vectors. In the matrix case `Var.size` returns the product of row and column size. All of these variables return integer values.

`Integer(a)`, `Float(a)`, and `Var.type(a)`

These functions provide explicit casting operations for converting float to integer and vice-versa. The `Integer` casting function truncates the value towards zero as is the case for Fortran, C, and Ada. `Var.type` is a general casting function that produces a resulting datatype that agrees with `Var`. If `Var`'s datatype is an integer, then integer truncation would occur.

`Abort(n)`

`Abort` is a `Void` function. Its output cannot be assigned to a variable, but rather, it is used as a procedure call. It must be passed an integer literal value which encodes a severity level and a message index. Its purpose is to stop (or raise an exception) during the simulation or AutoCode code generation. The values for the integer are the same error message variables as those defined for `UserCode` blocks; these are negative values. See *Simulation Errors* on page 14-49 for details.

`Abs(a)`

This function takes the absolute value of its argument. The resultant datatype is the same as that of the argument.

`Acos(a)` and `Asin(a)`

`ACOS` and `ASIN` return the arccosine and arcsine, respectively, of the argument. The argument must be float and if it is larger than 1 or less than -1 a run-time error will occur. `ACOS` returns a float value in the range 0 to Pi. `ASIN` returns a float value in the range: -Pi/2 to Pi/2.

Atan(a) and Atan2(y,x)

Both of these functions return the arctangent of their input argument(s). `Atan2` returns the arctangent of (y/x) which is a float value in the range $-\pi$ to π dependent upon which quadrant (x,y) maps in the Cartesian coordinate frame. `Atan`, on the other hand, returns a float value in the range $-\pi/2$ to $\pi/2$. If `Atan2` is passed two zero values, a run-time error will occur.

Bset(a,b), Bclear(a,b), Btest(a,b) and Btoggle(a,b)

These functions set, clear, test or toggle bit b in integer word a . The bit position, b , is 0 for the low-order bit. This definition is consistent with the Digital FORTRAN intrinsics: `IBSET`, `IBCLR`. `Btest` returns a logical result and is consistent with the Digital FORTRAN intrinsic with the same name. Digital FORTRAN does not have a `Btoggle` function, which we provide here for convenience.

BitLshift(a,b) and BitRshift(a,b)

`BitLshift(a,b)` shifts integer word a left b bits while `BitRshift(a,b)` shifts integer word a right b bits. The output type is an integer.

BitNot(a), BitOr(a,b) and BitAnd(a,b)

`BitNot` performs a bitwise complement of integer word a . `BitOr` and `BitAnd` perform bitwise And and Or, respectively, for their input arguments. The output type is integer.

Bound(a,b,c)

`Bound` returns b if a is less than b ; c if a is greater than c ; otherwise, a is returned. The arguments must be all float or all integer. The returned value is the same data type as the arguments to `Bound`.

Exp(a)

`Exp` returns the value e raised to the power a where e is the natural number ($\approx 2.7183\dots$). The argument to `Exp` must be a float and the returned value is a float.

Log(a) and Log10(a)

`Log` returns the base e logarithm of its input argument while `Log10` returns the base 10 logarithm. Both functions require a float input argument and produce a float result. If the input is negative a run-time error will result.

Max(a,b) and Min(a,b)

Max returns the larger of the two arguments while Min returns the smaller of the two. Both arguments must agree in datatype, which is the datatype of the returned value.

Mod(a,b)

This function takes two arguments. It performs the operation:

```
a - b*Integer(a/b)
```

Both *a* and *b* must be the same datatype. The resultant datatype is the same as that of its arguments.

Quad(a, w, x, y, z)

Quad accepts float arguments and returns a float result. The function is evaluated as follows. If *a* is in the interval [*x*,*y*] then the output is 1.0. If *a* is less than or equal to *w* or greater than or equal to *z* then the output is 0.0. Otherwise, if *a* is in the interval (*w*,*x*) then the output is an interpolated value between 0.0 and 1.0; or, if *a* is in the interval (*y*,*z*) then the output is an interpolated value between 1.0 and 0.0. The values, *w*, *x*, *y*, and *z* must be increasing. Note that *w* may be equal to *x* and/or *y* may be equal to *z*.

Round(a), Truncate(a), Floor(a) and Ceiling(a)

These functions quantize the float input and produce a float output. Round quantizes to the nearest integer. Truncate quantizes to an integer in a direction towards zero. Floor quantizes to an integer value in the direction of negative infinity. Ceiling quantizes to an integer value in the direction of positive infinity.

Sign(a)

Sign computes the signum function of its input. It is defined to be +1 when *a* > 0, -1 when *a* < 0, and 0 when *a* == 0. The resulting datatype is the same as that of its argument.

Sin(a), Cos(a), and Tan(a)

Sin computes the sine of its input while Cos computes the cosine of its input. Both functions require a float input and return a float result in the range: -1 to 1. Tan computes the tangent of its argument. If *a* is a multiple of π , Tan will overflow. The output of Tan is float.

Sinh(a), Cosh(a) and Tanh(a)

These functions compute the respective hyperbolic functions. `Sinh` returns a float value. `Cosh` returns a value that is greater than or equal to unity. `Tanh` returns a value greater than -1 and less than +1.

Sqrt(a)

This function returns the square root of its input argument. A run-time error will occur if the input argument is negative. Both the argument and the returned value are float.

Swap(a,b)

This function swaps the values referenced by `a` and `b`. `a` and `b` must be simple variable name references and can either be both float or both integer.

Trg(a, x, y, z)

`Trg` accepts float arguments and returns a float result. The function is evaluated as follows. If `a` is equal to `y` then 1.0 is the result. If `a` is less than or equal to `x` or greater than or equal to `z`, then 0.0 is the result. Otherwise, if `a` is in the interval (`x,y`) then the output is an interpolated value between 0.0 and 1.0; or, if `a` is in the interval (`y,z`) then the output is an interpolated value between 1.0 and 0.0. The values `x`, `y`, and `z` must be increasing. Note that `x` may be equal to `y` and/or `y` may be equal to `z`.

Urand(s,v), Nrand(s,v), OUrand(s, ouLast, timeInterval, timeConst,v)

These functions generate random numbers. The first argument, `s`, is an integer seed. The seed must be declared as a parameter, so it can be changed by the function. (Directly specifying an integer will not work as expected, but a parameter will.)

- `Urand` is a uniform random number generator that returns a float value in the range 0.0 to 1.0 in the `v` argument.
- `Nrand` is a normal random number generator that returns a float Gaussian value that has zero mean and unit variance in the `v` argument.
- `OUrand` implements the Ornstein-Uhlenbeck process for generating band-limited white noise. It is correlated with past history given the float values `ouLast`, `timeInterval` and `timeConst`. The `timeInterval` should be the delta time between the current and previous function call. `ouLast` is the last value returned from the previous function call. The random value is returned in the `v` argument.

12.3.5 Environment Variables

Environment variables must be all upper-case. They are read-only values.

ABSTOL

ABSTOL is the absolute tolerance specified in the `sim({abstol=value})` function call.

EPSILON

EPSILON is the smallest float value that can be added to unity and change its value. This value is machine dependent.

INIT

For blocks without states, INIT is set True the first time the BlockScript program is called during simulation or code generation, and False all the other times.

For blocks with one or more states, INIT is called twice, once the first time that OUTPUT is set True, the other the first time that STATE is set True. Note that each time the block is called with INIT and either OUTPUT or STATE True, all the *other* statements in the IF INIT part will be duplicated into the IF STATE and IF OUTPUT segments of the generated code.

OUTPUT

OUTPUT is set True to request the BlockScript program to perform output update computations.

PI

PI (=3.14159...) is the circumference of a circle divided by its diameter.

RELTOL

RELTOL is the relative tolerance specified in the `sim(...,{reltol=value})` function call.

STATE

STATE is set True to request the BlockScript program to perform state update computations.

TIME

TIME is the current absolute value for time. It is a float scalar value.

TSAMP

TSAMP is a float value that is the sample time of the parent Discrete SuperBlock.

TSTART

TSTART is always equal to t_0 , the first time point of the first sim call.

12.4 Debugging Tips

With minor modifications, you can include all or part of the body of your BlockScript program in a MathScript function, to be executed from Xmath, and run it with the MathScript debugger.

12.5 Compiling BlockScript Blocks

During simulation, BlockScript statements are interpreted for execution. Other types of blocks are not interpreted, and are evaluated by built-in functions. As a result, simulation speed is reduced when BlockScript blocks are used.

A solution is available for AutoCode customers. This method involves placing the BlockScript block or blocks inside a Procedure SuperBlock, generating stand-alone Procedure code from the SuperBlock, compiling and linking the generated code, and finally invoking the stand-alone procedure as a UserCode block (UCB).

NOTE: This procedure gives enhanced performance at the expense of flexibility. You cannot use %Variables inside a stand-alone procedure.

1. Write and debug a BlockScript block (named, for example, MYBLOCK) as explained in this chapter.
2. Place the BlockScript block inside a Procedure SuperBlock named MYPROC. An easy way to do this is to:
 - Select the BlockScript block, then select Edit→Make SuperBlock.
 - Open the SuperBlock block dialog and name the block; for example, MYPROC. Be sure to remember this name; it becomes the name of your stand-alone procedure. Click **OK**.
 - Double-click on the new SuperBlock (MYPROC) to open it. Single-click in the SuperBlock ID bar to raise the SuperBlock properties dialog.
 - On the Attributes tab, go to the **Type** field and select Procedure. Click **OK**.

3. Now, create a new SuperBlock. Go to the Catalog Browser and select File→New→SuperBlock. Name it (for example, MYSUPER), make its type Discrete; and specify at least one output. Click **OK**.
4. Position the Catalog Browser and the SuperBlock editor so that you can see both. In the Catalog Browser, click the SuperBlock hierarchy heading (in the left pane) so that all SuperBlocks are displayed in the Contents view (the right pane).

Locate the SuperBlock MYPROC in the Contents view. Drag MYPROC from the Catalog Browser into the Editor. Select File→Update to make sure the new information appears in the Catalog Browser.

5. Go to the Catalog Browser SuperBlock hierarchy and select MYSUPER. Select Tools→AutoCode. In the Generate Real-Time Code dialog **Code Style** field, select Procedures, then press **OK**. By default, the code will go to MYSUPER.C.

The file that is generated is the source code for your stand-alone procedure. Alter the model that originally contained the BlockScript block to use the newly generated wrapper (which takes the form of a UCB).

6. Open the original model in the Editor. To replace the BlockScript block, raise the Palette Browser and drag a UCB icon so that it covers the BlockScript block; when you release the mouse the UCB will take the BlockScript block's place.
7. Open the UCB for editing; in the **Name** field, type MYSUPER.C. In the Function Name field, type MYPROC. Make sure that the UCB Inputs, Outputs, and States are consistent with the original BlockScript block settings. Click **OK**.

The first time the new SuperBlock is simulated, the procedure code will be compiled and linked into your simulator, creating a local version of the simuch shared library file. Every subsequent time you run the simulator, the local version will be used.

CAUTION: Any time you simulate a model with a new stand-alone procedure present, the system will automatically compile, link, and simulate using the new procedure. Old local copies of the simuch shared library are overwritten. If the project directory still contains the code objects from which the shared library was made, they will be included in the new library. If these code objects are not present, the simuch shared library should be removed before starting this process so that new objects will be created.

12.6 Examples

The following sections contain examples designed to demonstrate BlockScript capabilities. The examples in Sections 12.6.1 and 12.6.2 show how an equation can be expressed as BlockScript and included in a model. The remaining examples are scripts that demonstrate BlockScript solutions for a variety of problems.

12.6.1 Bessel Equation BlockScript Block

This example uses a BlockScript block to model and solve a nonlinear differential equation, also known as a Bessel equation of order zero:

$$y'' + \frac{1}{u} y' + y = 0$$

To use the equation in BlockScript, it must be transformed to state-space representation:

$$\begin{aligned}\dot{x}_1 &= x_2 \\ \dot{x}_2 &= -x_1 - \frac{1}{u} x_2 \\ y &= x_1\end{aligned}$$

1. Load the Catalog file from the Xmath command area:

```
load file="$SYSBLD/examples/blockscript_example/blkscrip_ex1.cat"
```

2. From the Catalog Browser open BlockScript_Example1. The BlockScript block Bessel_eq_BScript displays the script used to implement the Bessel equation.
3. In Xmath, enter the time and input vectors:

```
t = [0:.1:10]'; u = ones(t);
```

4. Enter the initial values for the %vars (shown on the Parameters tab in Bessel_eq_BScript).

```
x0_1 = 2.2;  #-- Initial condition for state #1
x0_2 = 0;    #-- Initial condition for state #2
```

5. From the Xmath command area, simulate the model and plot the results.

```
[ ,y] = sim( "BlockScript_Example1", t, u, {vars} );
plot( t, y, {title = "Solution of Bessel eq. order = 0",
        x_lab = "time [s]"} )?
```


12.6.2 Discrete PID Controller BlockScript Block

This example illustrates the BlockScript implementation of a discrete PID controller. Of course this controller is available as a standard block, however we will demonstrate it can also be modeled successfully using the BlockScript block. In some instances you may want a PID controller with dynamically scheduled gains that can be adjusted during actual simulation; the BlockScript implementation is a good solution in this case.

To keep the example simple we will not modify the gains, however, the BlockScript implementation of the PID controller we present is ready to be used with dynamically adjusted gains.

This example uses the following equations for the proportional, integral, and derivative components (equations are represented in Z domain):

$$\begin{aligned}
 y_p &= K_p u && \text{proportional component} \\
 y_i &= \left(\frac{K_i T_s z}{z - 1} \right) u && \text{integral component (Backward Euler Integrator)} \\
 y_d &= \left(\frac{K_d(z - 1)}{T_s z} \right) u && \text{derivative component}
 \end{aligned}$$

The actual output will be: $y = y_p + y_i + y_d$

The state-space representation of the above dynamic system will be:

$$\begin{aligned}
 x_1[k + 1] &= x_1[k] + T_s u && \text{the integral state} \\
 x_2[k + 1] &= u[k] && \text{the derivative state} \\
 y[k] &= K_p u[k] + K_i x_1[k] + \frac{K_d u[k] - x_2[k]}{T_s}
 \end{aligned}$$

Where:

T_s Sample period for the Discrete PID controller.

K_p Proportional component gain.

K_i Integral component gain.

K_d Derivative component gain.

1. Load the Catalog file from the Xmath command area:

```
load file="$SYSBD/examples/blockscript_example/blkscript_ex1.cat"
```

2. From the Catalog Browser open BlockScript_Example2. The BlockScript block PID_Ctrl_BScript contains the script for the PID controller.
3. In Xmath, enter the time and input vectors:

```
t = [0:.001:.04]'; u = ones(t);
```

4. Enter the initial values for the %vars (shown on the Parameters tab in PID_Ctrl_BScript).

```
ts = 0.001 #-- Sample period for the Discr PID controller [s]
x0_1 = 0; #-- Initial value for integral state #1
x0_2 = 0; #-- Initial value for derivative state #2
kp = 2; #-- Proportional component gain
ki = 100; #-- Integral component gain
kd = 0.002; #-- Derivative component gain
pid_gains = [kp, ki, kd];
```

5. From the Xmath command area, simulate the model and plot the results.

```
[, y] = sim( "BlockScript_Example2", t, u, {vars} );
plot( t, y, {marker, x_lab = "time [s]",
         title = "Cl. Loop step resp. (PID controller -> Motor)" } )?
```

12.6.3 Three-Cycle Delay Script

This example implements a three-cycle delay block. The standard delay block implementation in SystemBuild uses states and next states/derivatives. Although the SystemBuild implementation is a complete solution, it may be expensive for much simpler needs. Here we use a BlockScript block to develop a custom algorithm that is highly efficient.

EXAMPLE 12-3: Three-Cycle Delay

```

inputs: u;
outputs: y;
parameters: (DelayBuffer, Index);

float    u(y.size), y(:);
y.type  DelayBuffer(y.size, 3);
integer Index;

for i = 1:y.size do
    y(i) = DelayBuffer(i, Index);
    DelayBuffer(i, Index) = u(i);
endfor;

Index = Mod(Index+1, 4);
if (Index == 0) then
    Index = 1;
endif;

```

The parameter `DelayBuffer` is used for holding the input value and is copied into the output variable when it is appropriate to do so. This buffer is 2 dimensional with the number of rows equal to the number of outputs and number of columns equal to the number of delay stages (3 in this example). Actual delay is accomplished by treating this buffer as a circular buffer and moving the read/write index in a circular fashion.

The parameter `Index` is used to record the circular indexing details. This example also illustrates the use of parameters for remembering values from one cycle to another. Using states for such simple application would be a burden because states are double-buffered.

`DelayBuffer` is initialized to an initial value specified on the BlockScript block parameters tab. Similarly, the parameter `Index` can also be initialized to an appropriate value.

12.6.4 Linear Interpolation Algorithm Script

This example implements a simple linear interpolation algorithm. In the System-build implementation of linear interpolation, the interpolation tables are parameters to the block. There could be circumstances where there is a need to interpolate among input values, i.e., the interpolation tables could be dynamic. This can be efficiently implemented in BlockScript using the input variable to represent both the actual input and the interpolation table.

EXAMPLE 12-4: Interpolating Among Input Values

```
inputs: u;
outputs: y;
parameters: (Gain);
float u(:), ulocal(u.size-1), y;
integer index, length;
float slope;

for i = 1:u.size-1 do
    ulocal(i) = u(1+i);
endfor;
length = (u.size - 1) / 2;

found = false;
index = 0;
while (!found) do
    if (u(1) < ulocal(index+1)) then
        found = true;
    else
        index = index + 1;
    endif;

    if (index == length) then
        found = true;
    endif;
endwhile;

if (index == 0) then
    yout = ulocal(length+1);
elseif (index == length) then
    yout = ulocal(length*2);
else
    slope = (ulocal(index+length+1) - ulocal(index+length)) /
            (ulocal(index+1) - ulocal(index));
    y = ulocal(index+length) + slope * (u(1) - ulocal(index));
endif;
```

The first location of the input vector `u` represents the actual input while the remainder represents the interpolation input and output tables. The input and output tables are of the same size. The input variable `u` is copied into a local variable `ulocal` because only local variables can be indexed with a while loop. Based on these tables, `slope` is calculated and the `slope` is used along with the actual input value to determine the output value.

12.6.5 Hysteresis Script

Consider the following BlockScript script for the Continuous Hysteresis (Backlash) block in SystemBuild. To write this file to your current working directory, enter the following in the Xmath command area:

```
copyfile "$SYSBLD/examples/blockscript_example/hysteresis.txt"
```

Note that all vector sizes are inherited from the **Outputs** field in the dialog box. This means that dimension changes in the **Inputs** and **States** fields are ignored. Likewise, the sizes for the parameters are fixed to match the Outputs dimension. In the script, ω is the cutoff frequency from the block dialog. Note that this program uses `estate` and `halfw`, two local variables that are defined when they are first used.

EXAMPLE 12-5: Hysteresis Script

```
inputs: u; outputs: y;
states: x; State_derivatives: xdot;
parameters: (omega, width, slope);

float y(:), u(y.size), x(y.size),
xdot(y.size);

float omega(y.size),
width(y.size), slope(y.size);

for i = 1:y.size do
    y(i)=slope(i)*x(i);
    halfw=width(i)/2.0;
    estate = u(i)-x(i);
    if estate>halfw then
        xdot(i) = omega(i)*(estate
            - halfw);
    elseif estate < -halfw then
        xdot(i) = omega(i)*(estate
            + halfw);

    else
        xdot(i) = 0.0;
    endif;
endfor;
```

13

SystemBuild Access (SBA)

13.1 Overview

SystemBuild Access (SBA) is a subset of Xmath commands and functions that access the SystemBuild catalog database from the Xmath command area. SBA allows you to create blocks, and modify, query, and delete SystemBuild objects. Almost every SuperBlock editor function has an SBA equivalent. To see a listing of SBA commands and functions, go to the Xmath command area and type:

```
help SBA
```

SBA commands and functions can be entered directly into the Xmath command window command area, executed from within MathScripts, or called from an Xmath UCI; in all cases SystemBuild must be running. You can use SBA to automate editing, check model consistency, query for model content, and enable interoperability between SystemBuild and other vendor supplied tools. When you combine SBA commands with Xmath commands and functions to create looping and branching constructs, you can create scripts to automatically create SystemBuild models with scalable structures.

NOTE: Most blocks are supported by SBA, even custom blocks you create yourself ([Section 18.3.1 on page 18-15](#)). However, IA icon blocks ([Section 8.5 on page 8-7](#)), are not supported.

[Figure 13-1 on page 13-2](#) shows how SBA fits into the MATRIX_x paradigm. A MathScript containing SBA commands and functions is executed and interpreted by Xmath. The interpreted SBA outputs are the same as the outputs of the SystemBuild editor, including SuperBlocks, blocks, connections, STDs, and other SystemBuild block diagram objects, used to create or modify SystemBuild models.

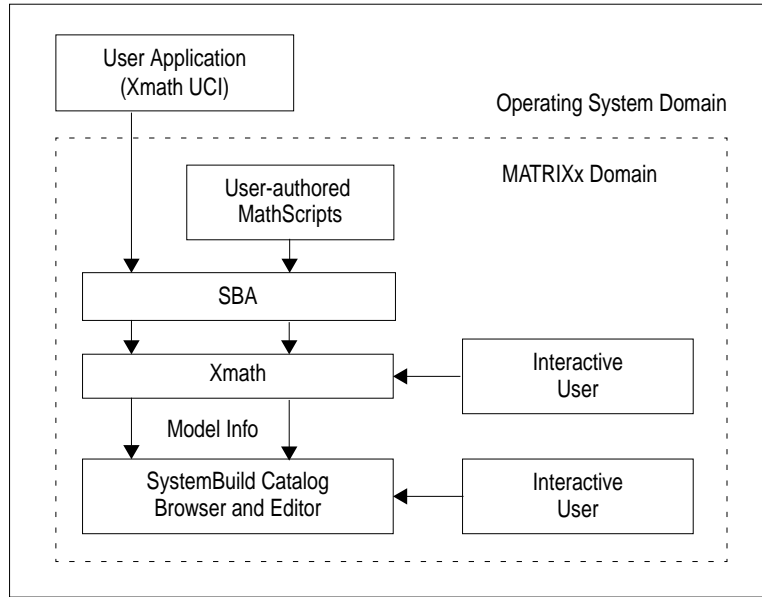


FIGURE 13-1 SBA MathScript Blocks in the MATRIX_x Context

Figure 13-2 shows how SBA can be used to extend the SystemBuild paradigm to allow the results of a simulation to change the model. In this scenario, a MathScript executes a simulation. The MathScript can evaluate `sim` results and then execute SBA code that modifies the model before the next simulation is started.

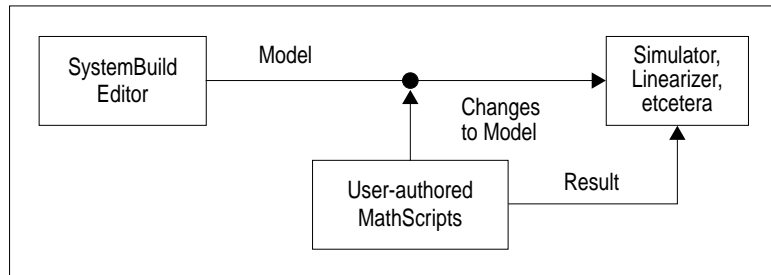


FIGURE 13-2 Typical SBA Program Used in a SystemBuild Application

13.2 Xmath Syntax Review

This section describes SBA syntax and other conventions and notation used in this chapter. SBA syntax is exactly the same as Xmath command and function syntax (see the *Xmath Basics*). The basic syntax, and the behavior of inputs, keywords, and outputs is the same.

13.2.1 Command Syntax

Xmath commands process inputs to perform operations, evaluate expressions, access intrinsic (or user-supplied) utilities and facilities, and more. By definition, commands do not return results as Xmath values. In SBA, commands are used to create, copy, delete, or modify SystemBuild objects in the Editor:

```
Command arg1, arg2, ... argn, {kwd=parameter_value}
```

A SystemBuild object is described by the values of its parameters, which closely correspond to all enabled fields in the relevant SystemBuild dialog. A created object is instantiated in the SystemBuild editor.

13.2.2 Function Syntax

Xmath functions operate on a list of input parameters without modifying them. SBA uses functions to query Xmath objects. They have the following syntax:

```
[v1=kwd, ..., vN=kwd] = function(in1,in2,{kwd=parameter_value})
```

The results of query function calls are returned to Xmath; any parameter that can be queried can be assigned to a variable. Because a SystemBuild object may have dozens of parameters that you may want to query, SBA functions take advantage of keyword output assignment. (While this capability is available in Xmath MSFs, it is not commonly used, because Xmath functions generally return few outputs). For an example, see [Section 13.3.2 on page 13-4](#).

13.2.3 Inputs, Optional Inputs, and Keywords

Each command or function has one or more required inputs, possible optional inputs, and keywords. These have the following properties for both commands functions:

Inputs Inputs are required; these must be ordered as shown in the syntax statement for each command or function. If a command syntax does not appear to specify inputs, it is because a default is assumed.

- Optional Inputs** An optional input appears after required inputs, and before keywords. If specified, the order is important.
- Keywords** Keywords always appear within curly braces {}. They are optional, as each is assigned a default value which will be used if the keyword is not called. Keyword order is not important unless it violates the logical rules of the Editor. See [Section 13.4](#).

13.3 Basic SBA Tasks

This section show some basic examples of SBA syntax. Any object you can create and modify in the Editor can be reproduced with SBA. This section shows SBA commands to create a simple model, then refers to it throughout the section.

13.3.1 Create

The following example creates a SuperBlock, blocks within the SuperBlock, and connections between them.

```
createsuperblock "cmd_createconnection", {inputs=5, outputs=2}
createblock "sin", {id=1, inputs=3}
createblock "elementproduct", {id=2,inputs=2}
con0_1=[...
  1,3;
  2,2;
  3,1];
con1_2=[...
  2,1;
  3,2];
createconnection 0,1,con0_1
createconnection 1,2,con1_2
createconnection 1,0,[1,1]
createconnection 2,0,[1,2]
```

13.3.2 Query

You can query an existing block to determine what its settings are, or you can query block options to determine the available keywords for a particular block. You must identify a SuperBlock by its name, but you can identify a primitive block by its name or its ID. Let's see what the valid options are for the ElementProduct block.

```
[optlist] = queryblockoptions("elementproduct");optlist?
```

```
optlist (a column vector of strings) =
```

```
BlockType
Name
Id
Inputs
Outputs
States
Comment
Location
Size
Color
Faces
OutputLabel
OutputName
InputName
InputPins
OutputPins
Labels
IconType
Border
OutputUserType
OutputDataType
OutputRadix
OutputMinimum
OutputMaximum
OutputAccuracy
OutputUnit
OutputComment
OutputScope
OutputAddress
CustomHelp
Container
PropagateLabels
CustomIcon
```

Given the available parameters, we can query an existing block for selected values by using keyword output assignment.

```
[l=location,s=size,ip=inputpins]=queryblock(1); l? s? ip?
```

```
l (a row vector) = 180 0
s (a row vector) = 80 80
ip (a string) = Scalar
```

13.3.3 Modify

These commands modify the model created above.

```
modifysuperblock "cmd_createconnection",  
    {inputlabel=["U1";"U2";"U3"]}  
  
modifyblock 1,{labels="on"}
```

13.3.4 Display

The SBADISPLAY command allows you to refresh the editor window after changes and control a diagram's size on the model level (individual block size is controlled by the `size` keyword definition for each block). We can try this on the existing model.

```
SBADISPLAY, {fit}  
SBADISPLAY, {normal}  
SBADISPLAY, {refresh}
```

13.3.5 Delete

You can delete any of the elements created so far. For example,

```
deleteconnection 0,1  
deleteblock 2  
deleteSuperBlock "cmd_createconnection"
```

13.3.6 Sample Scripts

For examples of some sophisticated SBA scripts, see `$(SYSBLD)/examples/export` or `%SYSBLD%\examples\export`. These unsupported SBA commands and functions read all or part of an existing model and create an SBA script that will reproduce the model. `Exportsuperblock.msc` is able to do this for an entire SuperBlock hierarchy. Other examples of SBA scripts can be found in `$(SYSBLD)/scripts`. You can use any of the scripts as long as the directory is in your path; alternatively, you can copy them locally and modify them to suit your purposes.

13.4 Using SBA

This section gathers a variety of tips that will help you get the results you want when using SBA.

13.4.1 Keyword Ordering

In general, Xmath keywords, because they have defaults, can be specified in any order. However, SBA does not always keep this rule, because Xmath cannot fully mimic the behavior of the Editor. For example, when working interactively it often does not matter which fields you define first as long as the inputs are found to be compatible when **OK** is pressed. If inputs are compatible, the Editor provides defaults, even for fields you did not edit; for example, if you specify three outputs, the editor will automatically give you the opportunity to specify three output names, three output labels, etc.

When in doubt, look at the block dialog. If you plan to define any of the values shown above the tabs, they should be defined first, and in the order shown from left to right. Inputs on the Parameter tab should be defined later. Let's apply this principle to the following block dialog.

StateSpace Block

Name: Inputs: Outputs: States: ID:

Parameters | Inputs | Outputs | States | Document | Comment | Icon | Display

Parameter	Value	% variable
System Matrix	[...]	
Zero Initial States	Yes	
Initial States	0, 0, 0	
Reltol Multipliers	1.0, 1.0, 1.0	
Abstol Multipliers	1.0, 1.0, 1.0	

System Matrix Rows: 6 Columns: 4 Value: xxx

	1	2	3	4
1	0	0	0	1.0
2	0	0	0	0
3	0	0	0	0
4	1.0	0	0	0
5	0	0	0	0

OK Cancel Help

The Name and ID are optional so it doesn't matter when they are defined. Of the remaining items, Inputs must be defined first, then Outputs, then States. The dimension of the System Matrix is determined from these values.

13.4.2 Block Parameters

Many blocks have parameter dependencies, or special parameters. The `query-blockoptions` function will return all parameters specific to the queried block, however, it is possible for legal parameters to be mutually exclusive. To be sure that you are properly using block parameters, consult the online help for the specific block before creating your script.

13.4.3 Error Handling

Xmath interprets SBA inputs literally; an error may result if there are dependencies between fields and the parameters are undefined, or incompatibly defined. In some cases, however, Xmath will pass a call to the Editor; if it does not make sense, it will attempt to create compatible settings. This is consistent with the editor's interactive behavior, but it makes script debugging difficult, as no error messages are returned to Xmath when SystemBuild encounters the improper values. For example, the following is legal command that will create a block in SystemBuild; unfortunately you may not get what you expect:

```
# legal (but undesirable) syntax:  
  
createblock "gain", {outputlabel=["A", "B", "C"], outputs=3}
```

Regardless of the fact that you have specified three labels, this command generates a block that has one output and one output label. When it is interpreted the output-label is encountered first so the default number of outputs (1) is assumed and one outputlabel is created. The final output definition is ignored because a value has already been assumed. As stated in [Section 13.4.1](#), inputs and outputs should always be defined first.

13.4.4 Input Formats

The online help details all SBA commands and functions. In the Xmath command area, type `help SBA` to display a list of links to SBA commands and functions. Look at different SBA commands; you will see that the input format (integer, matrix of strings, etc., is specified for each parameter).

The following table shows samples of possible input formats.

TABLE 13-1 Possible Input Formats

Format	Comment or Example
Float	Argument requires a single float value.
Float, Vector, or Matrix	Argument requires a matrix of floats. Size and matrix orientation is command dependent. <code>modifyblock "test", {outputaccuracy=[.01, .001, .0001]}</code>
Integer	Argument requires a single integer value. <code>queryblock(98)</code>
Integer Matrix	Argument requires a matrix of integers. Size and matrix orientation is command dependent. <code>createconnection 2,0,[1,2]</code>
String	The string length may be context sensitive. <code>createblock "timedelay", {name="td"}</code> <code>modifyblock "td", {id=2}</code>
Matrix or Vector of Strings	The number of strings and the matrix orientation are context sensitive. <code>modifyblock 99, {outputcomment=["com1", "com2", "com3"]}</code>
Boolean	Indicates the keyword is a simple yes/no or on/off parameter. In these cases, the presence/absence of the keyword defines the value.

Multiple Input/Output Specification

A number of SystemBuild objects have multiple input and/or multiple output (MIMO) capabilities. In these cases, certain parameters are constrained to have dimensions proportional to either the number of inputs, outputs or both. These inputs can be specified as an array of values, or, each value can be specified separately by appending the index value to the keyword.

For example, use an array of strings to define the output names.

```
createblock "gain", {id=13,inputs=3,
  outputs=3,outputname=["gain_A","gain_B","gain_C"]}
```

To change one name after the fact, append the index number to the keyword:

```
modifyblock 13, {outputname2="changed_gain_B" }
```

13.4.5 SuperBlock Editor Coordinate System

As shown in [Figure 13-3](#), the SuperBlock Editor coordinate system occupies the fourth quadrant of the coordinate plane. In this quadrant, X values are positive and Y values are negative. A full-sized Editor window thus lies between (0,0) and (1200,-900). By default, all blocks except containers are 80 x 80; containers are 100 x 100.

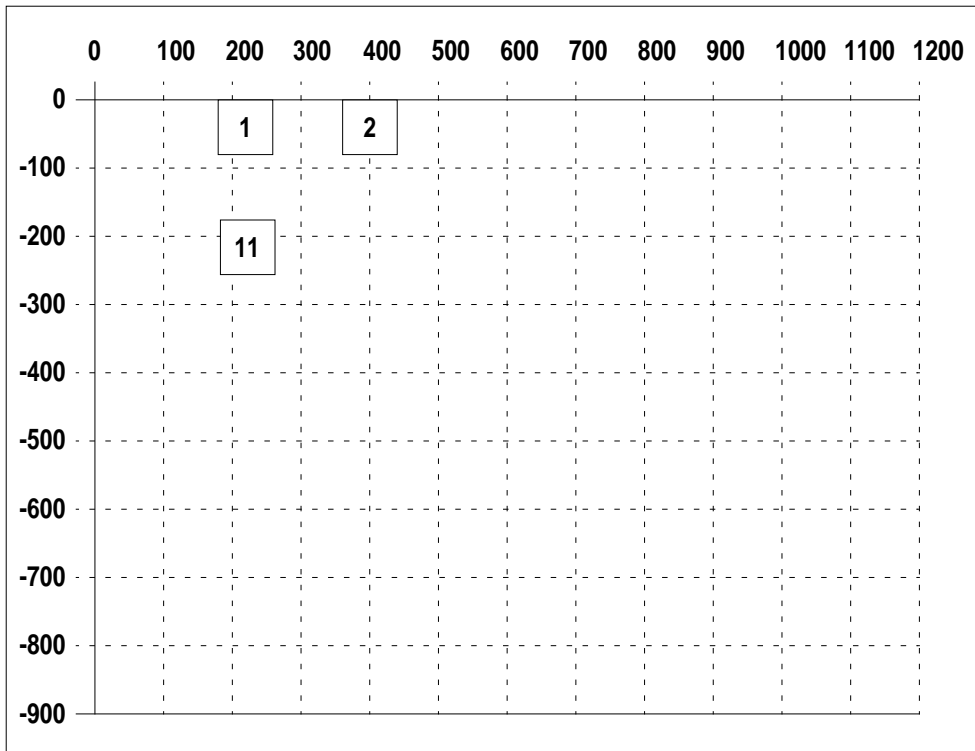


FIGURE 13-3 SystemBuild Coordinate System

When SBA places new blocks in an empty diagram, the default location of the first block 1 is (180,0), and if no ID is specified the default ID is 1. If the default is assumed for subsequent blocks they will be placed from left to right and numbered sequentially; if the number of blocks is greater than 10, a new row will be started at (180,-180). Note that block 11 starts a new row. You can position blocks in the dia-

gram by specifying a block ID. For example, assigning a block ID of 14 will place a block at (720,-180), as long as that position is unoccupied.

You should not specify a Y value greater than 0; although SystemBuild will place your block, unpredictable numbering can result.

13.5 Tutorial

As is explained above, SBA allows you to write MathScript files that contain commands to build, modify, and query a SystemBuild model. Also, Xmath allows you to execute commands that simulate, linearize, and perform other operations on models. These two types of operations can be combined in a MathScript to build and execute models.

13.5.1 Building the Predator-Prey Model

This model is from the field of population demographics, illustrating the interaction of a predator and a prey species in a competitive environment. You can copy it from `$SYSBLD/examples/pred_prey/predprey.ms`.

```
# Create predator prey model using SBA
# _____
#
#
# Build window must be opened first
build
#
# Create the top level SuperBlock
createsuperblock "predator_prey",
{inputs=1,outputs=2,type="continuous"}
#
# Create the prey integration loop
createblock "integrator",
  {name="x_prey",id=1,location=[100,0],initialstates=1}
createblock "gain",
  {name="c_times_xprey",gain=1,id=2,location=[300,0]}
createblock "summer", {name="xdot_prey",id=3,location=[500,0]}
#
# Connect the prey integration loop
createconnection 1,2
createconnection 2,3, [1,1]
createconnection 3,1
#
# Create the predator integration loop
createblock "integrator", {name="x_pred",id=4,
location=[100,-300],initialstates=1}
```



```

createblock "elementproduct", {name="a_xpred",id=5,
location=[300,-300]}
createblock "summer", {name="xdot_pred",id=6,
location=[700,-300],icontype="special"}
#
# Connect the predator integration loop
createconnection 4,5, [1,1]
createconnection 5,6, [1,2]
createconnection 6,4
#
# Create the blocks for the interaction between predator and prey
createblock "gain",
  {name="b_times_xprey",gain=2,id=7,location=[250,-150]}
createblock "elementproduct",
  {name="b_xpred_xprey",id=8,location=[400,-150]}
createblock "gain", {name="k",gain=.5,id=9,location=[600,-150]}
#
# make the connections between the two loops
createconnection 1,7
createconnection 7,8, [1,1]
createconnection 4,8, [1,2]
createconnection 8,9
createconnection 8,3, [1,2]
createconnection 9,6, [1,1]
#
# Make the external connections
createconnection 0,5, [1,2]
createconnection 1,0, [1,1]
createconnection 4,0, [1,2]

```

Note that the most natural way of operating within the SystemBuild editor may differ sometimes from the obvious way of doing things in SBA. When working interactively it is typical to create a few blocks, then perhaps duplicate blocks in order to build the model quickly and efficiently; you may connect a few blocks, then create more, putting in the external inputs and outputs as you go, and connecting things whenever it seems convenient. By contrast, in this MathScript we first create a loop, then form the connections; we then create the second loop and form its connections. We create the blocks that work between the two loops, then connect them; finally, we add the external inputs and outputs. You can organize your script as you see fit.

13.5.2 Simulating the Predator-Prey Model

This MathScript file is available on your system as `$$SYSBLD/examples/pred_pre/predprey_driver.ms`, which in turn calls (executes) the previous file, `$$SYSBLD/examples/pred_pre/predprey.ms`. Next, we create the time- and input data vectors, run the simulation and plot the output.

```
# Create the model
execute file = "predprey.ms"
# Create the t and u input vectors
t = [0:.1:20]';
u = ones(t);
#
# Run the simulation
y = sim("predator_preym",t,u);
# plot the results
plot(t,y)?
#
# modify the model
modifyblock 9, {gain=.2}
# rerun the simulation
y2 = sim("predator_preym",t,u);
#
# plot outputs from both simulation runs
plot(t,[y,y2],{linecolor=["black","black","red","red"],
                 linestyle=[1,2,1,2]})?
# save the output data
save t y "predprey.out" {matrixx,ascii}
```

14

UserCode Blocks

14.1 Introduction

The SystemBuild UserCode block (UCB) interface allows you to call your own (or any external) C and FORTRAN subroutines from within the SystemBuild program. A UCB may be connected and manipulated within your model, just like any block in the standard block library. The hand-written source code instructions for UCBs are referred to as UserCode functions.

Before using UCBs, you must familiarize yourself with programming, compilation, and linking procedures for your hardware platform and environment.

In SystemBuild, there are two types of UCBs: explicit and implicit. For most cases, you should use the explicit version. The implicit version is only required when implicit equations are used; implicit equations require using the DASSL, ODASSL or GEARS integration algorithm. The ImplicitUserCode block is restricted to continuous systems, whereas the UserCode block has no restrictions.

Note that SystemBuild UCBs differ from the hand-written UCBs used for AutoCode. The differences reflect the varying needs of simulation as contrasted with generated real-time code. In a simulation environment, provision must be made for continuous and hybrid considerations such as linearization, implicit equations, and state events. AutoCode does not support these features and has strict performance requirements and a fixed calling sequence. The Integrated Systems template files (`usr01.c` or `usr01.f` for simulation, `sa_user.c` for AutoCode) reflect these differences. SystemBuild UCBs cannot be used in AutoCode, however, you can link AutoCode hand-written UCBs into the SystemBuild simulation engine and simulate them, but their functionality will be limited to what AutoCode supports.

This chapter deals with the programming and linking aspects of UCBs. See the on-line help for the UserCode block for information on the block dialog itself.

In this chapter, the notation for indexing arrays (particularly the `IINFO` array) follows FORTRAN conventions: indexing starts with 1, index values appear in parentheses (). For the C language, the corresponding indexes would start with 0, and the index values would appear in square brackets []. See the `$SYSBLD/src/usr01.c` and `iusr01.c` files for comparative lists of the C and FORTRAN conventions.

14.2 The Structure of UCBs

The design of the UCB interface emphasizes a paradigm of accepting inputs and returning updates of states (optional memory elements that carry information from one cycle to the next) and required outputs.

14.2.1 Explicit UCBs

In continuous systems, the explicit UCB can represent a set of first-order ordinary differential equations (ODE) of the form:

$$\begin{aligned}\dot{x} &= f(x, u) \\ y &= g(x, u)\end{aligned}$$

In discrete systems, it can represent a set of first-order difference equations of the form:

$$\begin{aligned}x_{k+1} &= f(x_k, u_k) \\ y_k &= g(x_k, u_k)\end{aligned}$$

In the above equation, u is the input vector, x is the state vector, and y is the output vector.

The UserCode function is called from the simulator to compute \dot{x} or x_{k+1} and y . The simulator will calculate x and pass it to the UserCode function; this value should not be modified except during initialization. There is one exception: in continuous UCBs, x can be modified during the event call for a state-event simulation (see [Section 14.2.6 on page 14-13](#)).

14.2.2 Implicit UCBs

For continuous systems only, the implicit UCB can represent the more general class of differential algebraic equations (DAE); that is, models described by both implicit

differential and algebraic equations. In the most general form, DAE systems are mathematically described by equations of the form:

$$\begin{aligned} 0 &= f(x, \dot{x}, u) \\ y &= g(x, \dot{x}, u) \end{aligned} \tag{Eq. 14-1}$$

f is a vector-valued function with dimension equal to the number of states, and g is the output equation vector, with dimension equal to the number of outputs.

In the `init` section of the implicit UCB, both the x and \dot{x} values may be initialized.

If, for example, an ODE is expressed as a DAE, the right-hand side of the equation must be in the proper form:

$$f(x, \dot{x}, u) = f(x, u) - \dot{x} \tag{Eq. 14-2}$$

Note that the DAE integrator calculates both x and \dot{x} . The UserCode function simply evaluates the implicit equation $f(x, \dot{x}, u)$ in the variable f with the supplied x and \dot{x} values. The integrator uses f as the local residual error and attempts to maintain it below a certain threshold.

The implicit UCB may also represent overdetermined differential algebraic equations (ODAE), that is, models that have more equations than unknowns. These types of systems are mathematically described by equations of the form:

$$\begin{aligned} 0 &= f(x, \dot{x}, u) \\ 0 &= f_c(x, \dot{x}, u) \\ y &= g(x, \dot{x}, u) \end{aligned} \tag{Eq. 14-3}$$

where f has dimension n_x and the constraint equation f_c has dimension n_c , which is the number of additional constraints. The number of constraints cannot exceed the number of states. In this case, make sure that the number of constraints, n_c , is specified in the implicit UCB block parameter dialog.

14.2.3 Implicit UCB Implementation

The equations of motion for multibody dynamics often result in systems of DAEs that sometimes possess additional constraints that the physical system must satisfy. In some cases, these equations can be reduced to explicit form with algebraic manipulations. However, reduction by analytical or numerical methods may require strong simplifications or serious analytical and numerical difficulties may result. For such problems, formulation and numerical solution of the equations of motion in the DAE or ODAE form offers the most convenient approach.

Consider the implicit differential equation: $f(\dot{x}, x, t) = 0$

The system has an index of 0 if and only if $\frac{\partial f}{\partial \dot{x}}$ is not singular. Note that this means that the equation can be (locally) transformed into an explicit form $\dot{x} = f_2(x, t)$ without any differentiations. If at least one differentiation of the implicit differential equation is required to transform the DAE into explicit form, then the DAE is said to have an index of 1. In general, assuming an index k , a DAE requires k differentiations to transform it into explicit form.

In order for SystemBuild to solve a DAE with DASSL, ODASSL, or GEARs (`ialg` options 6, 9, and 10, respectively), the DAE must have an index of 0 or 1. This is because these algorithms are not designed to handle systems of index greater than 1. In particular, DASSL and ODASSL will fail if the Jacobian is singular:

$$J = \frac{\partial f}{\partial x} + c \frac{\partial f}{\partial \dot{x}} \quad \text{EQ. 14-4}$$

See [Section 14.2.4](#) for information concerning how to use `simout` for a workaround when the operating point computation fails due to a singularity of the Jacobian.

For a DAE defined using the implicit UserCode block, when a simulation is started, the initial conditions $\dot{x}(t_0)$ and $x(t_0)$ must be consistent, that is, they must satisfy:

$$f(\dot{x}(t_0), x(t_0), u(t_0)) = 0 \quad \text{EQ. 14-5}$$

If [Equation 14-5](#) is not satisfied, the algorithms may fail when the integration process is started.

In exactly the same way, any constraint equations that may be part of the implicit UCB should also be satisfied by the initial conditions:

$$f_c(\dot{x}(t_0), x(t_0), u(t_0)) = 0 \quad \text{EQ. 14-6}$$

When using ODASSL, SystemBuild first calculates the rank of the matrix $\frac{\partial f}{\partial \dot{x}}$. If there are derivatives that do not explicitly appear in the equations, then the equations that are associated with the variables are de-emphasized in the local error calculations. For an example of this procedure, see [Example 14-1](#).

The ODASSL technique for incorporating the constraints into the DAE is numerically equivalent to the Gear Stabilization technique. If a DAE is integrated without its constraints, the solution tends to “drift away” from the correct answer. The constraints act as corrections that stabilize the solution.

The most common type of DAEs or ODAEs appear in multi-body system dynamics, such as vehicles, satellites, and robots. Typically, the DAEs obtained from the Lagrangian formulation yield the following equations of motion for holonomic systems:

$$\begin{aligned} \dot{\mathbf{p}} &= \mathbf{v} \\ \mathbf{M}(\mathbf{p})\dot{\mathbf{v}} &= \mathbf{f}(\mathbf{p}, \mathbf{v}, \mathbf{u}) - \frac{\partial}{\partial \mathbf{p}} \mathbf{g}_p(\mathbf{p})^T \boldsymbol{\lambda} \\ 0 &= \mathbf{g}_p(\mathbf{p}) \end{aligned} \tag{Eq. 14-7}$$

where:

- \mathbf{p} Generalized position variables.
- \mathbf{v} Generalized velocity variables.
- \mathbf{M} The inertia matrix.
- \mathbf{f} Function of Coriolis, centrifugal, and gravitational forces, and external inputs, \mathbf{u} .
- \mathbf{g}_p Position constraints.
- $\boldsymbol{\lambda}$ Generalized constraint forces, also called Lagrange Multipliers. By differentiating these equations, we can see that the DAE defined above has index 3.

In order to successfully solve such a system in SystemBuild, an index 1 formulation must be obtained (higher index formulations may fail during integration).

Since the constraint $\mathbf{g}_p(\mathbf{p})$ is a function of positions, it can be differentiated to obtain constraints on velocity and acceleration:

$$\begin{aligned} \mathbf{g}_v(\mathbf{p}, \mathbf{v}) &= \dot{\mathbf{g}}_p(\mathbf{p}) = 0 \\ \mathbf{g}_a(\mathbf{p}, \mathbf{v}, \dot{\mathbf{v}}) &= \ddot{\mathbf{g}}_p(\mathbf{p}, \mathbf{v}) = 0 \end{aligned} \tag{Eq. 14-8}$$

Thus, three types of formulations are possible with this example:

1. Unconstrained DAE formulations:
 - a. Index 3 formulation:

$$\begin{aligned} 0 &= \dot{\mathbf{p}} - \mathbf{v} \\ 0 &= \mathbf{M}(\mathbf{p})\dot{\mathbf{v}} - \mathbf{f}(\mathbf{p}, \mathbf{v}, \mathbf{u}) + \frac{\partial}{\partial \mathbf{p}} \mathbf{g}_p(\mathbf{p})^T \boldsymbol{\lambda} \\ 0 &= \mathbf{g}_p(\mathbf{p}) \end{aligned} \tag{Eq. 14-9}$$

where there are $n_p+n_v+n_\lambda$ states, and the same number of equations.

b. Index 2 formulation:

Instead of $0 = g_p(p)$, use equation $0 = g_v(p, v)$.

c. Index 1 formulation:

Instead of $0 = g_p(p)$, use equation $0 = g_a(p, v, \dot{v})$

2. Index 2 formulation, with one constraint:

DAEs:

$$\begin{aligned} 0 &= \dot{p} - v \\ 0 &= M(p)\dot{v} - f(p, v, u) + \frac{\partial}{\partial p}g_p(p)^T \lambda \\ 0 &= g_v(p, v) \end{aligned} \tag{EQ. 14-10}$$

Constraints:

$$0 = g_p(p) \tag{EQ. 14-11}$$

3. Index 1 formulation with two constraints:

DAEs

$$\begin{aligned} 0 &= \dot{p} - v \\ 0 &= M(p)\dot{v} - f(p, v, u) + \frac{\partial}{\partial p}g_p(p)^T \lambda \\ 0 &= g_a(p, v, \dot{v}) \end{aligned} \tag{EQ. 14-12}$$

Constraints:

$$\begin{aligned} 0 &= g_v(p, v) \\ 0 &= g_p(p) \end{aligned} \tag{EQ. 14-13}$$

The most reliable numerical results are usually obtained from the index-1 constrained formulation (number 3) above.

EXAMPLE 14-1: Pendulum Example

The above formulations will be demonstrated by a simple pendulum example; see [Figure 14-1](#).

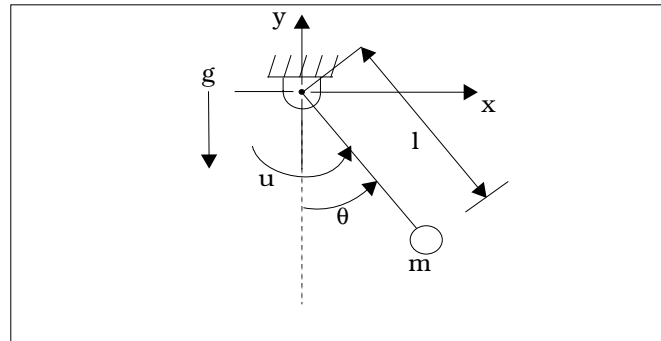


FIGURE 14-1 Pendulum Example Diagram

The pendulum is connected to the ground with a pivot. It is assumed to have mass m concentrated at the endpoint with link l having zero mass. An input torque u may excite the motion, applied at the pivot point.

The equation of motion, using the generalized coordinate θ is:

$$\ddot{\theta} = \frac{g}{l} \sin \theta = u \quad \text{EQ. 14-14}$$

which is an ODE. For purposes of illustration, the equation of motion is derived using the coordinates x and y .

$$g_p - (x^2 + y^2 - l^2) = 0 \quad \text{EQ. 14-15}$$

[Equation 14-15](#) is used in the following equations:

$$m\ddot{x} = -\frac{y}{l^2}u - x\lambda \quad \text{EQ. 14-16}$$

$$m\ddot{y} = -\frac{x}{l^2}u - mg - y\lambda \quad \text{EQ. 14-17}$$

$$0 = (x^2 + y^2 - l^2) \quad \text{EQ. 14-18}$$

The above set of equations constitutes an index-3 unconstrained formulation. For index-2 and index-1 formulations, we use the velocity constraint:

$$\mathbf{g}_v = \dot{\mathbf{g}}_p = x\dot{x} + y\dot{y} = 0 \tag{EQ. 14-19}$$

and the acceleration constraint

$$\mathbf{g}_a = \dot{\mathbf{g}}_v = x\ddot{x} + \dot{x}^2 + y\ddot{y} + \dot{y}^2 = 0 \tag{EQ. 14-20}$$

in place of the position constraint in [Equation 14-15](#).

Thus, an index-1, two-constraint formulation of this problem (as in case 3 above) would be as follows:

With external input taken to be 0, let:

$$\begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \lambda \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \text{ then } \left\{ \begin{array}{l} \text{D.A. (with algebraic level constraints)} \\ x_1 - x_3 = 0 \\ x_2 - x_4 = 0 \\ m\dot{x}_3 + x_1x_5 = 0 \\ m\dot{x}_4 + x_2x_5 + mg = 0 \\ x_1\dot{x}_3 + x_3^2 + x_2\dot{x}_4 + x_4^2 = 0 \end{array} \right. \tag{EQ. 14-21}$$

Velocity level constraint $x_1x_3 + x_2x_4 = 0$

Position level constraint $(x_1^2 + x_2^2 - l^2) = 0$

This example is coded as an implicit UserCode block in \$SYSBLD/examples/pendulum_imp/pend_imp.c. To run it, copy files pend_imp.c, pend_imp.dat, and pend_imp.ms from \$SYSBLD/examples/pend_imp/ into your local directory. Then enter:

```
execute file = "pend_imp.ms"
```

This will load the SuperBlock in pendulum.dat and simulate the implicit UCB.

14.2.4 Implicit UCBs and sim, lin, or simout

Implicit UCBs contain equations of the form $f(x, \dot{x}, u) = 0$. When these equations are not exactly satisfied, there is a residual $\delta = f(x, \dot{x}, u)$. At the beginning of

each `sim`, `lin`, or `simout` operation, the SystemBuild software tries to compute a valid operating point where the residual δ is zero.

Initialize Mode

In the Implicit UCB block dialog, the field **Initialize Mode** sets the initial conditions to be either `states` or `derivatives`. The interpretation of the `sim`, `lin`, or `simout` input arguments `x0` and `xd0` for state and derivative initial conditions *is dependent on this dialog box definition*.

TABLE 14-1 State and State Derivative Initial Conditions

Dialog Definition	sim, lin, or simout argument	
	X0	Xd0
states	sim initial condition	initial condition for operating point solver
derivatives	initial condition for operating point solver	sim initial condition

If they are specified, the `sim`, `lin`, or `simout` arguments `x0` or `xd0` override the dialog defaults.

Implicit Integration Algorithm Operating Point

Implicit implementation algorithms (DASSL, ODASSL and GEARS) require that the operating point be consistent (*i.e.*, the residual δ is 0 or very small) at the beginning of a simulation. Under normal conditions, SystemBuild's operating point solver will compute the correct initial conditions for x or \dot{x} before the integration starts. However, there may be cases in which the operating point is singular; that is, the Jacobian of the equations:

$$\frac{\partial}{\partial x} f(x, \dot{x}, u) \quad \text{EQ. 14-22}$$

or

$$\frac{\partial}{\partial \dot{x}} f(x, \dot{x}, u) \quad \text{EQ. 14-23}$$

(or a mixed partial if different implicit UCBs have different initial condition definitions for states/derivatives) is singular. Under these conditions it is still possible to start the simulation (or to perform `lin` or `simout`, provided that the residual δ is zero, or very small.

When the operating point is singular, it is up to the user to provide initial conditions, because the SystemBuild software cannot compute them. The `simout` function can facilitate this by setting `initmode = 4`. For example, if,

```
[x,xdot,y]=simout("model",
    {x0=x0_init, xd0=xd0_init, u0=u0_init, initmode=4})
```

then the operating point computation is bypassed and the `xdot` vector of the output argument of `simout` contains *not the derivatives*, but the *residuals* $\delta = f(x_0, \dot{x}_0, u_0)$ in its corresponding entries.

You can use this feature to write algorithms that compute the correct initial conditions x_0 , \dot{x}_0 , or u_0 iteratively.

14.2.5 Input Direct Terms

As previously mentioned, UCBs are systems that solve the following linked pair of equations,

$$\begin{aligned} \dot{x} &= f(x, u) \\ y &= g(x, u) \end{aligned}$$

where all of these (x , \dot{x} , u , and y) are vectors of user-defined size. Since a UCB generally has multiple inputs, it is possible (in many case, likely) that one or more of the inputs is not used in the computation of any of the outputs, but instead is used only to compute the state derivatives (next state in the discrete case).

This makes constructs such as the one shown in [Figure 14-2 on page 14-11](#) not only possible, but solvable without resorting to implicit solvers.

If none of the outputs are dependent on input 1, then the simulator may compute the other inputs to the UCB, execute the UCB, compute inputs to the Summer, and finally compute the Summer. Later when the simulator computes the state derivatives, input 1 will be valid. Since the simulator cannot automatically determine which UCB inputs are used to create outputs, the user must tell the simulator which inputs are direct terms. Only direct terms are included in the output pass that computes the UCB outputs. If direct terms are not identified, the simulator will assume that this construct forms an algebraic loop, and will either require the use of an implicit System Solver (if the SuperBlock is continuous) or will insert a time delay between the UCB and the Summer.

To make an input a direct term, set the direct terms vector on the UCB's Parameters tab. This vector is of the same size as the number of inputs to the UCB block. A value of 1 identifies the input as a direct term. A value of 0 means that the corresponding input is not used in the computation of any of the outputs of the UCB.

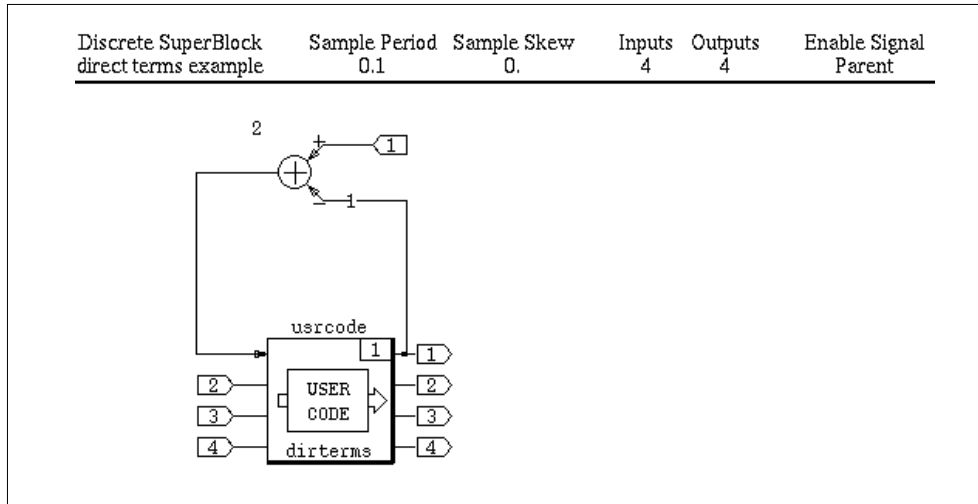


FIGURE 14-2 UCB with Direct Terms

Properly identifying direct terms will assist the simulator and AutoCode in the detection of algebraic loops in the model (more correctly, this capability allows the simulator to note that some loops that are seemingly algebraic are actually not). Note that any UCB input that is not marked as a direct term is undefined during the output pass of the UCB, but is valid for the other passes (including state, monit, and event). Use caution. Incorrectly identifying the direct terms can invalidate the numeric results of the simulation, since certain outputs would be computed using undefined inputs.

EXAMPLE 14-2: Sample Source File Demonstrating Direct Terms

```
#include "matsrc.h"

#if (FC_)
#define dirterms dirterms_
#endif
#if (PC || VAX)
#define DIRTERMS DIRTERMS
#endif

void dirterms (iinfo,rinfo, u,nu, x,f,nx, y,ny, rpar,ipar)
int iinfo[], ipar[], *nu, *nx, *ny;
double rinfo[], rpar[], u[], x[], f[], y[];
```

```

{
  int i;

  int init      = iinfo[1]==1;
  int last     = iinfo[1]==2;
  int state    = iinfo[2]==1;
  int output   = iinfo[3]==1;
  int lin      = iinfo[4]==1;
  int monit    = iinfo[5]==1;
  int event    = iinfo[6];
  int nse      = iinfo[7];
  int nip      = iinfo[8];
  int nrp      = iinfo[9];
  double time  = rinfo[0];
  double tsamp = rinfo[1];
  double tskev = rinfo[2];

  double *uop, *xop;

  /*-----
  Initialization code. It is recommended that you insert code here
  to check the assumptions required for this function to operate.
  The code below is an example of some checks that can be done.
  -----*/
  if (init) {
    if ( *ny!=*nu ) {
      stdwrt("ERROR : Number of inputs and outputs must be equal.\n");
      iinfo[0] = -2;
    }
    if ( *nx!=1 ) {
      stdwrt("ERROR : Number of states for this example must be 1\n");
      iinfo[0] = -2;
    }
  }
}

/*-----
State function update, compute state derivatives xdot in f vector
-----*/
if (state) {
  f[0] = u[0];
}

/*-----
Output function update, compute outputs in the y vector
-----*/
if (output) {
  y[0] = x[0];
  for (i=1; i<*ny; i++)
    y[i] = u[i];
}

if (lin) {
}

```

```

if (monit) {
}

if (event) {
}

/*-----
  Terminate simulation. If any action needs to
  be done when the simulation ends, do it here.
  -----*/
if (last) {
}

return;
}

```

14.2.6 State Events

In continuous systems, both explicit and implicit UCBs support integration of piecewise-continuous models; i.e., equations that have discontinuities in x , or, for implicit systems, in some of the derivatives. Moreover, the structure of a system may change completely by switching among different differential equations as a function of time or the state values.

The time points at which discontinuities occur are described by state events, which are implicitly defined by the zeros of a user-supplied monitor function. For explicit systems, the monitor function is of the form:

$$f_m(x, u, t) \quad \text{Eq. 14-24}$$

and, for implicit systems:

$$f_m(\dot{x}, x, u, t) \quad \text{Eq. 14-25}$$

where f_m is of order n_{se} , the number of state events specified in the UCB block parameter dialog.

Discontinuities are treated by stopping the integration at each discontinuity point and restarting the integration afterwards. Because the integration is restarted, the discontinuity does not affect the integration method. If many discontinuities appear during a simulation interval (e.g., at every integration step), this technique becomes very inefficient.

If State Events are selected in the UCB parameter dialog, the UserCode function is called from the integration executive to compute the monitor function.

If one or more of the monitor functions passes through zero, the integration executive will locate this point within a tolerance (`ztol`) with respect to the time axis, and halt the integration. At that point, the UserCode function is called and given the opportunity to modify the state vector for explicit systems, or the state and derivative vectors for implicit systems. This must be performed in the event section of the UCB code. Note that the implicit DAE should be consistent (i.e., $f(x, \dot{x}, u) = 0$) with the given state and derivative values.

If multiple state events are specified in one UCB, the event flag (`IINFO(7)`) may be used to determine which state event experienced a zero crossing. The range of the event flag is from 1 to NSE. If there are multiple state events at a given instant, the simulator will make individual calls for each event.

14.2.7 The Simulation API

The Simulation Application Programming Interface (SIMAPI) is a collection of library routines that are accessible from UserCode Blocks (UCBs) during simulation to provide access to internal simulation capabilities. There are four classes of capabilities:

- Gather information about the UCB block reference
- Control the currently running simulation
- Access the Runtime Variable Editing (RVE) capability
- Gather internal information about the state of the continuous integration being executed.

CAUTION: UCBs that take advantage of the SIMAPI capabilities will not be transferable to the AutoCode environment.

For C language UserCode Blocks, adding a reference to the header file `simAPI.h` will provide access to the SIMAPI functionality.

This capability is not recommended for FORTRAN UCBs. If SIMAPI access is attempted, it is the user's responsibility to make sure that the FORTRAN code calls SIMAPI functions with a C language calling interface.

This document describes the SIMAPI at the time of publication. The most current description can be found in the files `simapi_ucbinfo.h`, `simapi_rve.h` and `simapi_debug.h`, all of which are located in the `$SYSBLD/src` directory (`%SYSBLD%\src` on Windows systems).

14.2.8 Using the SIMAPI to Gather UCB Reference Information

The SIMAPI provides several functions to gather information about the current UCB block reference, and the environment in which it resides. These functions are:

```
int SIMAPI_GetUCBBlockInfo(int *iinfo, SB **SBptr, BLK **BLKptr)
```

Provides access to data that will allow the caller to find out information about the instance of the UCB being called, as well as the SuperBlock in which it exists.

```
int SIMAPI_GetBlockInputType(int *iinfo, int channel)
```

Returns the type of a specified block input.

```
char *SIMAPI_GetBlockInputLabel(BLK *block, int channel, SB *parent)
```

Returns the label associated with a specified block input.

```
int SIMAPI_GetBlockOutputType(BLK *block, int channel)
```

Returns the type of a specified block Output.

```
char *SIMAPI_GetBlockOutputLabel(BLK *block, int channel)
```

Returns the label associated with a specified block Output.

```
char *SIMAPI_GetDefaultOutputLabel(BLK *block, int channel)
```

Returns the default label associated with a specified block Output. This default is used when a user enters neither an output name nor an output label for the specified channel.

```
char *SIMAPI_GetBlockName(BLK *block)
```

Returns the name of the block reference

```
int SIMAPI_GetBlockId(BLK *block)
```

Returns the block ID of the block reference

```
char *SIMAPI_GetSBName(SB *Superblock)
```

Returns the name of the SuperBlock reference

To use these functions to access block information, first make a call to `SIMAPI_GetUCBBlockInfo`. This will initialize the SB and BLK structures that will be used in subsequent calls to SIMAPI functions. Once this call has been made, simply pass these structures, as required, to other SIMAPI functions. Any SIMAPI call that returns a character string allocates the memory for that string as part of the SIMAPI call. You can free that memory (using the standard 'C' `free()` call) if desired.

- The following example uses the SIMAPI to get the name of the currently executing block reference:

```
BLK   *blk;
SB    *sb;
char  *name;

SIMAPI_GetUCBBlockInfo(iinfo, &sb, &blk);
name = SIMAPI_GetBlockName(blk);
```

Note that BLK and SB are types defined in `simAPI.h`.

- The following example uses the SIMAPI to get the label associated with output number three of the current UCB reference.

```
BLK   *blk;
SB    *sb;
char  *name;

SIMAPI_GetUCBBlockInfo(iinfo, &sb, &blk);
name = SIMAPI_GetBlockOutputLabel(blk, 3);
```

14.2.9 Using the SIMAPI to Access and Modify Variables

The SIMAPI also provides users the ability to access variable information (%var and Variable block), as well as to modify their values. This feature provides the same capability as the Runtime Variable Editing (RVE) feature used in ISIM.

All of the variables in the model are ordered in a particular numeric sequence from the first variable to the last (to find the ID of the last variable, call the `SIMAPI_GetNumVars` function). To keep compatibility with the graphical RVE interface, the IDs for the variables range from 1 to the number of variables, instead of the standard C language scheme of numbering items from 0.

The functions provided by this capability are:

```
long  SIMAPI_GetNumVars(void);
```

Returns the number of variables (var block and %vars) used in the model, regardless of their "editability."

```
char  *SIMAPI_GetVarName(long varnum);
```

Returns a character string representing the name of the variable #varnum from the variable list.

```
char *SIMAPI_GetVarPartition(long varnum);
```

Returns a character string representing the Xmath partition in which variable #varnum was initialized.

```
long SIMAPI_IsVarEditable(long varnum);
```

Returns the editable status of variable #varnum.

```
char *SIMAPI_GetVarDatatypeName(long varnum);
```

Returns a string representing the name of the data type of variable #varnum.

```
char *SIMAPI_GetVarUsertypeName(long varnum);
```

Returns a string representing the name of the user datatype of variable #varnum. Returns NULL if no user type is assigned to that variable.

```
void SIMAPI_GetVarDimension(long varnum, long *dim);
```

Returns the dimensions of a selected variable. Any dimension not used by the variable will be set to 0 (i.e. if the variable is a 3x4 array, this array will be set to [3,4,0,0,0,0]). Scalars are treated as 1x1 arrays.

```
long SIMAPI_GetVarStorageSize(long varnum);
```

Returns the size (in bytes) needed to store a variable's value.

```
SIMAPI_StorageType SIMAPI_GetVarStorageType(long varnum);
```

Returns the method by which multi-dimensional data is stored, either BY_ROWS or BY_COLUMNS (which are defined in simapi_rve.h)

```
long SIMAPI_GetVarIndexByName(char *varname);
```

Returns the index of the variable whose name is varname.

```
long SIMAPI_GetVarData(long varnum, void *data, char *error_string);
```

Returns the value of the selected variable. The data pointer will be allocated by this routine, and will be of size SIMAPI_GetVarStorageSize(). The data will be of whatever type is specified by SIMAPI_GetVarDatatypeNameblock.

```
long SIMAPI_PutVarData(long varnum, void *data);
```

Edits the selected variable. Note that variable edits are not completed until the edit is flushed.

```
long SIMAPI_FlushVars(long *varlist);
```

Flushes selected variables. This completes the editing process for these variables. This routine expects a list of size `SIMAPI_GetNumVars()`. For each entry in this list with a non-zero value, the corresponding variable will be flushed (the edited value transferred to the simulation), assuming it had previously been edited. After this operation is complete, the edit can no longer be undone.

```
void SIMAPI_ResetVar(long varnum);
```

Cancels an edit on the selected variable. Undoes the effects of `SIMAPI_PutVarData`. Note that once a variable edit has been flushed (by `SIMAPI_FlushVars`), it cannot be canceled.

The following example uses the SIMAPI to retrieve the name of the third variable:

```
char *varname;
varname = SIMAPI_GetVarName(3);
```

The following example uses the SIMAPI to change the variable of a known scalar variable named `myvar` to 20:

```
longvarnum;
double myval_value = 20.0;
longretval;
long*varlist;

varnum = SIMAPI_GetVarIndexByName("myvar");
if (varname != 0) {
    retval = SIMAPI_PutVarData(varnum, &myval_value);
    if (retval != 0) {
        varlist = calloc(SIMAPI_GetNumVars(), sizeof(long));
        varlist[varnum-1] = 1;
        retval = SIMAPI_FlushVars(varlist);
    }
}
```

14.2.10 Using the SIMAPI to Access Simulation Debugging Information

The SIMAPI provides access to simulation debugging information, so that you can monitor certain internal computations in a SystemBuild model during the simulation.

The set of functions that provide access to the SystemBuild simulation signals are prototyped in a header file `simapi_debug.h`. This file is located in the directory

`$/SYSBLD/src`. In order to access the debugging features, the header file `simAPI.h` must be referenced in the C language UserCode Blocks, not `simapi_debug.h`.

The following is a list of the functions available through the debugging interface:

Functions to initialize and terminate debug data

```
SIMAPI_InitializeUserDebug(void);
SIMAPI_TerminateUserDebug(void);
```

Functions to return the dimensions of the SystemBuild model

```
SIMAPI_GetStateDimension      (int *n_states);
SIMAPI_GetImplicitOutputDimension (int *n_imp_outputs);
SIMAPI_GetExternalInputDimension (int *n_ext_inputs);
SIMAPI_GetExternalOutputDimension (int *n_ext_outputs);
```

Functions to return signal names of the SystemBuild model

```
SIMAPI_GetStateName          (int state_index, char **state_name);
SIMAPI_GetImplicitOutputName(int impout_index, char **impout_name);
SIMAPI_GetExternalInputName (int extinp_index, char **extinp_name);
SIMAPI_GetExternalOutputName(int extout_index, char **extout_name);
```

Functions to return signal values of the SystemBuild model

```
SIMAPI_GetStateValue          (int state_index, double *state_value);
SIMAPI_GetStateDerivativeValue(int state_index, double *deriv_value);
SIMAPI_GetImplicitOutputValue (int impout_index, double *impout_value);
SIMAPI_GetExternalInputValue  (int extinp_index, double *extinp_value);
SIMAPI_GetExternalOutputValue (int extout_index, double *extout_value);
```

Functions to return the Jacobians of the SystemBuild model

```
SIMAPI_GetOperatingPointJacobian(int *iinfo, int *is_available,
    double **OP_Jacobian);

SIMAPI_GetImplicitSolverJacobian(int *iinfo, int *is_available,
    double **IS_Jacobian);
```

The dimensions of the Jacobian matrices are as follows:

OP_Jacobian: (n_imp_outputs)x(n_imp_outputs)

IS_Jacobian: (n_states + n_imp_outputs)x(n_states + n_imp_outputs)

The IS_Jacobian is computed with respect to the variable ordering:

```
[states; imp_outputs]
```

All matrices are stored in FORTRAN storage style, that is, columnwise vector storage.

Function to return the simulation status of simexe

```
SIMAPI_GetSimStatus(int *iinfo, SimStatus *status);
```

Table 14-2 indicates the variables in the above function arguments that are inputs or outputs to functions:

TABLE 14-2

OUTPUTS:		INPUTS:
deriv_value	n_ext_inputs	extinp_index
extinp_name	n_ext_outputs	extout_index
extout_name	n_imp_outputs	impout_index
extinp_value	n_states	impout_index
extout_value	state_name	is_available
iinfo	state_value	IS_Jacobian
impout_name	n_ext_inputs	state_index
impout_value		status

Note that all functions described in this section return an integer status value:

```
SIMAPI_OK = 0,          /* Call was successful */
SIMAPI_Unsuccessful = 1, /* Call failed          */
```

14.2.11 Using and Managing the Debugging Access Functions

The following is a description on the usage of these functions. The function prefix SIMAPI_ has been omitted from the descriptions for brevity.

Initialization and Termination Functions

- The function `InitializeUserDebug` must be called in the `INIT` section of the UserCode block before any other debugging SIMAPI calls. This call performs internal initialization of the debugging capability.
- The function `TerminateUserDebug` must be called in the `FINAL` section of the UserCode block to allow the simulator to terminate and re-initialize the debug-

ging capabilities to prepare the debugging environment for subsequent debug simulations.

Dimension Functions

The user first calls the `Dimension` access functions to get the dimensions of signals in the model. The `Dimension` functions can be called anywhere, preferably in the `INIT` section for efficiency. This is because the `STATE` and `OUTPUT` sections of the `UserCode` block will be called many times throughout the simulation.

Name Functions

The `Name` functions are called to access the name of each signal in the system. Each function is called with the integer index of the associated signal such that $0 \leq i < n$ where n is the dimension of the signal. Note that this addressing scheme is done in C style. The memory allocation management for the string variables is done by the simulator. The `Name` functions can be called anywhere, preferably in the `INIT` section for efficiency. Note that the memory needed for the strings is allocated by the simulator, which cleans it up at the end of the simulation, during the call to `TerminateUserDebug`. You don't need to free this memory.

State Functions

The `GetStateValue` function can be called in any section of the `UserCode` block. To obtain state values during a converged integration algorithm pass, it should be called in the `OUTPUT` section when the `CONVERGED` flag is true.

State Derivative Function

The `GetStateDerivativeValue` function should be called in the `STATE` section of the `UserCode` block to access the state derivatives during intermediate integration algorithm evaluations.

Input Function

The `GetExternalInputValue` function can be called in the `STATE` or `OUTPUT` section of the `UserCode` block.

Output Functions

The `GetImplicitOutputValue`, `GetExternalOutputValue` functions are called in the `OUTPUT` section of the `UserCode` block.

Jacobian Functions

The functions `GetOperatingPointJacobian` and `GetImplicitSolverJacobian` are used to access the Jacobian of the model during these computations. In

the case of the Operating Point Jacobian, the matrix has dimensions $(n_{y_{imp}}) \times (n_{y_{imp}})$.

The dimension of the Implicit Solver Jacobian is $(n_x + n_{y_{imp}}) \times (n_x + n_{y_{imp}})$, where n_x is the number of states in the model, and $n_{y_{imp}}$ is the number of implicit (algebraic loop) variables in the model. The number of algebraic loops in the model is determined by the Simulation analyzer. See the `analyze` function for how to obtain this value.

Computational Status Function

`GetSimStatus` is called to determine what the simulation executable is doing at any given instant. This function provides information about the various computational modes of the simulator. The status flags are maintained in the following data structure (see `simapi_debug.h`):

```
typedef struct _SimStatus {
    SimulationMode simulationmode;
    StateUpdate    stateupdate;
} SimStatus;
```

The enumerated data structure `SimulationMode` can assume the following values:

```
integrate, /* (= 0) Integration           Update maybe converged states */
continuous, /* (= 1) Continuous Subsystem Update */
discrete, /* (= 2) Discrete Subsystem Update */
trigger, /* (= 3) Triggered Subsystem Update */
jacobian, /* (= 4) Jacobian Update */
opoint, /* (= 5) Operating Point Update for algebraic loops */
linearize, /* (= 6) Lin Update */
reset, /* (= 7) Reset Update */
converge, /* (= 8) Integration Update with converged states */
monitor, /* (= 9) Monit Update */
event /* (=10) Event Handle Update */
```

The enumerated data structure `StateUpdate` can assume the following values:

```
typedef enum
{
    skip_xdot,
    compute_xdot,
} StateUpdate;
```

Thus the `SimulationMode` data structure provides a snapshot of what the simulator is doing at any given instant (i.e. function call to the UserCode block). Based on this information, the user can decide which simulation data to access.

14.2.12 Integration Algorithm Updates

Since the User debugging feature is only accessible through UserCode blocks, it is very important to understand how SystemBuild executes UserCode blocks as well as other blocks in the model.

There are several factors that influence how often a UserCode block will be executed in a simulation:

Execution of State vs. Output Sections in the UserCode

1. Usually when a block is executed, it is called in two passes. In the first pass, the outputs are updated. In the second pass, the state derivatives are updated. The two updates cannot be done simultaneously because the outputs must be propagated throughout the system before the state derivatives can be computed. It is possible to have only an output update in order, for example, to post user output values. Also, during numerical integration some output updates may be skipped if they do not affect the dynamic part of the model.

One important issue in debugging a system with the UserCode block debugging feature is that if the UserCode block does not have any states (number of states is defined in the block form) then *it will not be called during STATE updates*. Thus, in order to debug the integration algorithm internal steps for states and derivatives, (i.e. derivative evaluations at non-converged time points) the UserCode block must be created with at least one (dummy) state. This will ensure that the block is in the appropriate chain of blocks to be executed during state updates.

A second issue for debugging integration algorithm internal computations is as follows: When the derivatives of the system are accessed, it is very important that these values belong to the most recent computational pass. In other words, the UserCode block should be the last block to be visited by the simulator during the computation of the derivatives. To ensure this order of computation, the UserCode block can be placed in the rightmost pane using a Sequencer block. This will make sure that when the analyzer sorts the blocks, the UserCode block is the last one to be executed.

2. Timing attributes.

The timing attribute of the parent SuperBlock is determined by its type: continuous, discrete free-running, discrete enabled, triggered, or procedure.

Continuous blocks are executed:

- At initialization (see next section)
- During numerical integration
- When state events occur
- When posting user output values

All other blocks are executed:

- At initialization (see next section)
- At the next sample time based on sample period, trigger values, etc.

3. Initialization

The initialization type is set by the "initmode" flag, and it can take the values [0-4]. The default value is initmode=3.

a. Type 0 initialization

Only the continuous subsystems are initialized. The UserCode block is executed once to initialize all system outputs. This is called the INIT call. The UserCode block can also be executed several more times with other calls (STATE, OUTPUT) by a Newton-Raphson solver in case the system includes ImplicitUserCode block or algebraic loops, so that a steady-state operating point can be found. Discrete subsystems outputs are left at $-\sqrt{\text{eps}}$.

b. Type 1 initialization

Continuous and discrete subsystems are executed once. Continuous subsystems (with output updates only), may be executed more than once if the system is implicit.

c. Type 2 initialization

Continuous, discrete, enabled, and triggered subsystems are executed once. Continuous subsystems (with output updates only), maybe executed more than once if the system is implicit.

d. Type 3 initialization

This is the same initialization type as for Type 2 except that outputs are propagated instantaneously between subsystems.

e. Type 4 initialization

This is the same initialization type as for Type 3 except that the Newton solver is disabled. This guarantees that the continuous subsystem is executed only once. However, after this initialization, the algebraic loops and ImplicitUserCode block derivatives (or states) remain uninitialized, with their values left at $-\sqrt{\text{eps}}$.

4. Numerical Integration Algorithm

The integration algorithm can take the values [1-9].

Below is a detailed explanation of all model updates done for each integrator. Converged updates are indicated in UserCode blocks by the flag `iinfo[13] = 1`. Otherwise it is 0. Note also that some blocks may be skipped for output updates during integration if they do not affect the dynamic part of the model. The notation is:

OUTPUT: Output Update.

STATE: State Update.

t: Time.

h: Current step size taken by the integration algorithm.

ialg = 1, Fixed-step Euler

```
OUTPUT: t = tk,    converged state values
STATE : t = tk,    converged state values
OUTPUT: t = tk+h, user output posting
```

ialg = 2, Fixed-step RK2

```
OUTPUT: t = tk,    converged state values
STATE : t = tk,    converged state values
OUTPUT: t = tk+h,  inside integration
STATE : t = tk+h,  inside integration
OUTPUT: t = tk+h,  user output posting
```

ialg = 3, Fixed-step RK4

```
OUTPUT: t = tk,    converged state values
STATE : t = tk,    converged state values
OUTPUT: t = tk+h/2, inside integration
STATE : t = tk+h/2, inside integration
OUTPUT: t = tk+h/2, inside integration
STATE : t = tk+h/2, inside integration
```

```

OUTPUT: t = tk+h,   inside integration
STATE  : t = tk+h,   inside integration
OUTPUT: t = tk+h,   user output posting

```

ialg = 4 or 5, Fixed and variable Kutta-Merson

```

OUTPUT: t = tk, converged state values
STATE  : t = tk, converged state values
OUTPUT: t = tk+h/3, inside integration
STATE  : t = tk+h/3, inside integration
OUTPUT: t = tk+h/3, inside integration
STATE  : t = tk+h/3, inside integration
OUTPUT: t = tk+h/2, inside integration
STATE  : t = tk+h/2, inside integration
OUTPUT: t = tk+h, inside integration
STATE  : t = tk+h, inside integration
OUTPUT: t = tk+h, user output posting

```

ialg = 6, DASSL

repeated sequence:

```

OUTPUT: t = tk+h, inside predictor iterations
STATE  : t = tk+h, inside predictor iterations
OUTPUT: t = tk+h, inside jacobian update
STATE  : t = tk+h, inside jacobian update

```

corrector update:

```

OUTPUT: t = tk+h, corrector update if not converged
STATE  : t = tk+h, corrector update if not converged

```

user outputs:

```

OUTPUT: t = tk+h, user output posting

```

ialg = 7, Adams-Bashforth-Moulton

```

OUTPUT: t = tk+h, predictor update
STATE  : t = tk+h, predictor update
OUTPUT: t = tk+h, corrector update
STATE  : t = tk+h, corrector update
OUTPUT: t = tk+h, interpolate outputs
STATE  : t = tk+h, interpolate outputs
OUTPUT: t = tk+h, user output posting

```

ialg = 8, Quicksim

```

OUTPUT: t = tk,    inside integration
STATE : t = tk,    inside integration
OUTPUT: t = tk+h, user output posting

```

ialg = 9, ODASSL and ialg=10, GEARS

Identical to DASSL

The timestep t_k is chosen at every point by computing the minimum of:

- difference between the current time and the next discrete, triggered, or enabled event
- difference between the current time and the next external output time determined by the user time vector.
- dtmax
- dtout

Note that the variable-step integrators may use a smaller internal timestep.

Other Issues and Notes

- The time steps taken by the variable-step numerical integration algorithms are limited by the user output points. The algorithms may take smaller steps whenever necessary (i.e. in order to satisfy the local error tolerance criterion). However, these intermediate converged values are not necessarily posted in the user's output vector during simulation from Xmath or the Editor. The debugging feature provides access to these values.
- Note the following for the computation of the operating point Jacobian matrix:
 - The Jacobian is computed only for algebraic loops associated with continuous subsystems,
 - The Jacobian computation is skipped when the sim option initmode=4.

If the function `SIMAPI_GetOperatingPointJacobian` is used for cases when the Jacobian is not computed, or not available due to one of the above cases, a program crash, or garbage output may result.

- Note the following for the computation of the Implicit Solver Jacobian matrix:

The Jacobian is only computed for the two Implicit Stiff Solver integration algorithms, ialg = 6 (DASSL) and ialg = 9 (ODASSL).

The symbolic form of the Jacobian computed by DASSL and ODASSL is:

$$J = \frac{\partial f}{\partial x} + c \frac{\partial f}{\partial \dot{x}} \quad \text{Eq. 14-26}$$

where $f = f(x, \dot{x}, u)$, and c is an iteration constant determined by the Implicit Solver.

14.2.13 SIMAPI Debug UserCode Block Example

An example UserCode block that illustrates the usage of the SIMAPI debugging features is provided with the MATRIX_X installation. In order to run the example, perform the following steps:

1. Copy the file `usrdebug.c` to your local directory.

```
copyfile "$SYSBLD/src/usrdebug.c"
```

2. Load and edit a SystemBuild model you would like to debug. In the editor, drag and drop a Sequencer block from the Software Construct palette to the model. Place the Sequencer to the right of all the blocks in the SuperBlock.
3. Drag and drop a UserCode block from the User Programmed palette, to the **right** of the Sequencer. This will ensure that the UserCode block is the last block to execute in the block update sequence.
4. In the UserCode block dialog, set the number of inputs, outputs and states equal to one. Even though the UserCode block does not have any states, setting the number of states equal to one will ensure that the block is executed during the state update pass of the simulator.
 - In the **File Name** field type `usrdebug.c`. In the **Function Name** field type `usrdebug`.
 - Set the number of Integer Parameters to 8. The dialog will now display an array of eight zeros as the default values of the integer parameters.
 - Set the values of the integer parameters as shown in [Table 14-3 on page 14-29](#) where the integer parameters array is referred as `ipar[1:8]`. The table also shows the files that will be created for the purpose of writing the rele-

vant data. Because the IPAR values are all 1, any easy way to enter them is to type **ones(1,8)** in the Integer Parameters field. Click **OK**.

TABLE 14-3 IPAR Values

Parameter	Purpose	File
ipar[1] = 1	Debugging feature for the UserCode block	sysinfo.dat
ipar[2] = 1	Enables state data	states.dat
ipar[3] = 1	Enables state derivative data	derivs.dat
ipar[4] = 1	Enables implicit output data	impouts.dat
ipar[5] = 1	Enables external input data	extinps.dat
ipar[6] = 1	Enables external output data	extouts.dat
ipar[7] = 1	Enables Operating Point Jacobian data	opjac.dat
ipar[8] = 1	Enables Implicit Solver Jacobian data	isjac.dat

5. Simulate your model.
6. Examine the data in the files listed in [Table 14-3](#).

SIMAPI Debugging Notes:

- To turn off the debugging of a particular type of variable, set the appropriate `ipar[]` element to zero.
- For the contents of each file and the data format used for writing the results to the files, please see the relevant `fprintf` statements in the source code `usrdebug.c` source code.
- For some models with a large number of states and many steps of simulations, some of the data files can become very large. Implicit Solver Jacobians can especially become very large due to repeated evaluations of the Jacobian inside the Implicit Solvers (`ialg= 6` and `9`).
- The example provided in `usrdebug.c` is an application adequate for users who would like to write the simulation data to a file and inspect the results. For more advanced applications, users are encouraged to read and freely re-use the source code of `usrdebug.c` for writing their own UserCode block debugging applications.

14.2.14 USR01 and IUSR01 Template

UserCode function templates are furnished in the `$SYSBLD/src` subdirectory. Copy the appropriate file to your working directory and insert your custom simulation algorithms in the example function.

The template consists of a single function prototype with sections designed to contain the computational algorithms associated with the seven distinct modes of operation: INIT, STATE, OUTPUT, LIN, MONIT, EVENT, and LAST.

Execution of each section is controlled by flags set by the simulator in the `IINFO` vector, as described in [Table 14-4 on page 14-33](#). The sequence and the modes of operation the UserCode function is required to compute will depend on the information specified in the block Parameters tab.

INIT Mode

INIT mode is performed once at the start of a simulation. During the INIT mode call, the value of the `IINFO(2)` flag is set to 1. If the UCB reference has states, the UCB will be called in INIT mode twice; once while executing OUTPUT mode, and again while executing STATE mode. If the UCB reference does not have states, the UCB will be called in INIT mode only once (while executing OUTPUT state).

Typically no action is required in the INIT section because the initial conditions defined in the UCB parameter dialog are automatically copied into `X`, the state vector. However, the state vector can be modified to override these automatically loaded values during this initialization.

If the LIN mode is used in your UserCode function, set the value of the `IINFO(5)` flag to 1 during the INIT call.

Other tasks can be performed at this time, such as opening files and allocating memory. Note that if the UCB performing these tasks has states, it is your responsibility to make sure that these operations occur only once (even though the UCB will be called in INIT mode twice). To ensure a single operation only, call these tasks only when both INIT and OUTPUT mode are active.

STATE Mode

STATE mode is performed to compute `xdot`, the state derivatives for the continuous case, `x_next` for the discrete case, or the local residual error for implicit equations. The result is returned in the `F` argument. During the State mode call, the value of the `IINFO(3)` flag is set to 1.

This call is only made when the block is specified with states.

OUTPUT Mode

OUTPUT mode is required, because UCBs must have one or more output signals. During the OUTPUT mode call, the value of the IINFO(4) flag is set to 1.

In OUTPUT mode, your code should compute the output vector in the Y argument. Be careful to assert every output of the block at every cycle, not just when the value changes.

MONIT Mode

MONIT mode is required, *only* for blocks that have state events. Compute the monitor function in F where the zeros define the locations of possible state event transitions. During the MONIT mode call, the value of the IINFO(6) flag is set to 1.

EVENT Mode

EVENT mode is required *only* for blocks that have state events when a zero crossing is found by the integrator. At this point the integration is halted and you may reinitialize the state values (for explicit) or state and derivative values (for implicit blocks). During the EVENT mode call, the value of the IINFO(7) flag is set to the i^{th} zero crossing detected during the MONIT mode calls.

LIN Mode

This optional mode allows you to explicitly calculate the Jacobian linearization of state and output equations. During the LIN mode call, the value of the IINFO(4) flag is set to 1.

By default, a double-sided finite difference linearization is always performed by the simulator linearization. However, you can use an explicit linearization calculation and insert a custom algorithm.

To enable the UCB to call your function with the LIN flag, set the lin flag, IINFO(5)=1 during the INIT mode. This will instruct the simulator to call your UserCode function to evaluate the following two expressions in the F and Y arguments, whenever a linearization is needed:

$$\delta \dot{x} = \left. \frac{\partial f}{\partial u} \right|_{x_{op}, u_{op}} \delta x + \left. \frac{\partial f}{\partial u} \right|_{x_{op}, u_{op}} \delta u \quad \text{EQ. 14-27}$$

$$\delta y = \left. \frac{\partial g}{\partial x} \right|_{x_{op}, u_{op}} \delta x + \left. \frac{\partial g}{\partial u} \right|_{x_{op}, u_{op}} \delta u \quad \text{EQ. 14-28}$$

LAST Mode

This optional mode is called once, at the completion of simulation. During the LAST mode call, the value of the IINFO(1) flag is set to 2.

This mode allows you to “close your books” on the simulation; e.g., by closing files, deallocating memory, and other housekeeping. No parameters are passed with this mode.

14.2.15 Explicit UserCode Function Calling Arguments

Refer to the listing of the `usr01` or `iusr01` templates for more information. The arguments for the explicit UserCode function are:

```
USR(IINFO, RINFO, U, NU, X, F, NX, Y, NY, RPAR, IPAR)
```

and for the implicit UserCode function, they are:

```
IUSR(IINFO, RINFO, U, NU, X, XD, NX, F, FC, Y, NY, RPAR, IPAR)
```

Where NU, NX, and NY are the dimensions of the respective values. It is important that during each call mode, only specific arguments that have write access are modified.

The meaning of U, Y, X, and F varies with the type of call and can be determined by values in IINFO.

NU	=	Number of inputs
NX	=	Number of states
NY	=	Number of outputs
RPAR	=	general vector of floating point parameters initialized by the simulator with the values entered from the parameter dialog and dimensioned NRP, which is found in IINFO(10)
IPAR	=	general vector of integer parameters that the simulator initializes with the values entered from the parameter dialog and dimensioned NIP, which is found in IINFO(9)

The parameter vectors RPAR (real) and IPAR (integer) let you pass parameters to your model from the block dialog. The values in RPAR and IPAR, as well as any initial state values, x_0 , can be changed in the block parameter dialog, so that a given UCB can be reused in a model, with different parameters. The RPAR, IPAR, and Initial Conditions parameters can be entered as %Variables. This lets you

modify the routine logic and equation coefficients between simulations without having to re-edit the model.

The tables in the following pages define the types of variables you will encounter using UCBs. [Table 14-4 on page 14-33](#) defines the IINFO vector. [Table 14-5 on page 14-34](#) defines the RINFO vector, and [Table 14-6 on page 14-35](#) defines all the Mode parameters associated by all the modes of the UCBs.

14.2.16 UserCode Function Arguments

IINFO is an integer array that contains information flags used for communication with the simulation engine. Except as noted in the discussion of INIT mode on [page 14-30](#), the UserCode function may only modify the first element of IINFO which is a status flag; modifying other values is illegal and may produce unpredictable results. As in the rest of this chapter, in the table, FORTRAN conventions are used in indexing into IINFO. For C code, the index starts from zero.

TABLE 14-4 IINFO Vector

IINFO(1)=0, -1, -2	Error Flag {0=Normal, -1=Warning, -2=Error}.
IINFO(2)=1, 2	INIT or LAST mode. Initialize (1 = First call, 2 = Last call).
IINFO(3)=1	STATE mode. Compute state derivatives in F.
IINFO(4)=1	OUTPUT mode. Compute outputs in Y.
IINFO(5)=1	LIN mode. Compute linearization in F and Y.
IINFO(6)=1	MONITOR mode. Compute monitor function for state event detection in F.
IINFO(7)=I	EVENT mode. Handle i th state event transition.
IINFO(8)=NSE	Number of state events.
IINFO(9)=NIP	Number of Integer Parameter Values.
IINFO(10)=NRP	Number of Real Parameter Values.
IINFO(11)=NC	Number of constraint equations (implicit UCB only).
IINFO(12)=1	Inside Linearization Process (Jacobian).
IINFO(13)=1	Inside SIM Initialization Process (TIME=0).

TABLE 14-4 IINFO Vector (Continued)

IINFO(14)=1	Update with converged state values.
IINFO(15)=1	Integration Algorithm (IALG).

RINFO is a real array that contains timing and related information for the called routine. UserCode may not modify any values in RINFO.

TABLE 14-5 RINFO Vector

	Continuous	Discrete	Triggered
RINFO(1)	Current Time	Current Time	Current Time
RINFO(2)	0.0	Sample Interval	1.0
RINFO(3)	0.0	Initial Time Skew	0.0
RINFO(4)	0.0	0.0	Timing Requirement
RINFO(5)	Sim Start Time	Sim Start Time	Sim Start Time
RINFO(6)	Sim End Time	Sim End Time	Sim End Time
RINFO(7)	reltol	reltol	reltol

TABLE 14-6 Mode Parameters

Mode	Name	Dimen	Access	Explicit UCB	Implicit UCB
INIT	U	NU	RO	Input vector	Input vector
	Y	NY	--	Output vector	Output vector
	X	NX	RW	State vector	State vector
	XD	NX	RW	--	Derivatives
	F	--	--	--	--
	FC	--	--	--	--
STATE	U	NU	RO	Input vector	Input vector
	Y	--	--	--	--
	X	NX	RO	State vector	State vector
	XD	NX	RO	--	Derivatives
	F	NX	WO	State derivatives	Residual error
	FC	NC	WO	--	Constraint eqn
OUTPUT	U	NU	RO	Input vector	Input vector
	Y	NY	WO	Output vector	Output vector
	X	NX	RO	State vector	State vector
	XD	NX	RO	--	Derivatives
	F	--	--	--	--
	FC	--	--	--	--
MONIT	U	NU	RO	Input vector	
	Y	--	--	--	
	X	NX	RO	State vector	State vector
	XD	NX	RO	--	Derivatives
	F	NSE	WO	Monitor function	Monitor function
	FC	--	--	--	--

TABLE 14-6 Mode Parameters (Continued)

Mode	Name	Dimen	Access	Explicit UCB	Implicit UCB
EVENT	U	NU	RO	Input vector	Input vector
	Y	--	--	--	--
	X	NX	RW	State vector	State vector
	XD	NX	RW	--	Derivatives
	F	--	--	--	--
	FC	--	--	--	--
LIN	U	2*NU	RO	[Uop;dU]	[Uop;dU]
	Y	NY	WO	dG	dG
	X	2*NX	RO	[Xop;dX]	[Xop;dX]
	XD	NX	RO	--	[XDop; dXD]
	F	NX	WO	dF	dF
	Y	NY	WO	dG	dG
LAST	No provision is made for passing parameters with this mode.				

Definitions:

$$df = \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial u} du + \frac{\partial f}{\partial \dot{x}} d\dot{x} \quad \text{EQ. 14-29}$$

$$dg = \frac{\partial g}{\partial x} dx + \frac{\partial g}{\partial u} du + \frac{\partial g}{\partial \dot{x}} d\dot{x} \quad \text{EQ. 14-30}$$

$$df_c = \frac{\partial f_c}{\partial x} dx + \frac{\partial f_c}{\partial u} du + \frac{\partial f_c}{\partial \dot{x}} d\dot{x} \quad \text{EQ. 14-31}$$

where $f = f(x_{op}, \dot{x}_{op}, u_{op})$, $g = g(x_{op}, \dot{x}_{op}, u_{op})$, $f_c = f_c(x_{op}, \dot{x}_{op}, u_{op})$, and x_{op} , \dot{x}_{op} , u_{op} correspond to the current operating point.

Access refers to what values the UCB is allowed to read or modify.

```
RO = Read only, do not modify
WO = Write only, must be calculated
RW = Read or write, may optionally modify
```

14.3 Variable Interface UserCode Blocks

This section discusses the variable interface UserCode block capability. It should only be used if your ultimate goal is generated code, because its sole benefit is enhanced code efficiency.

14.3.1 Overview

Although SystemBuild UCB code ([Section 14.2.14 on page 14-30](#)) cannot be used in AutoCode, handwritten UCB code intended to be linked with AutoCode can be linked into a SystemBuild simulation if the user code conforms to a standard interface. A standard UCB interface can be either **Fixed** or **Variable**.

A **Fixed** interface UCB conforms to a required set of function arguments specified in the *AutoCode Reference*. In the **Fixed** interface code generation paradigm for UCBs, all signals passed into the UserCode block are converted to float; at the completion of the user code task, floats are cast to the output datatype specified in the block diagram.

The **Variable** interface accommodates mixed datatypes and optional arguments. It passes the input signal datatypes and output datatypes specified in the UserCode block dialog. In addition, signal labels and names from the dialog will influence UCB inputs and outputs (to produce scalars or arrays) just as they do for other blocks. Finally, although the variable interface makes a strictly ordered call to the UCB, it allows arguments to be optional, that is, it does not call or generate code for parameters that are not used.

With these capabilities it is possible for the same handwritten code to be used in both SystemBuild and AutoCode simulations. Because the code is written for AutoCode simulation, you must write a C wrapper that will allow the SystemBuild simulator to interface with the fixed calling sequence that AutoCode expects. The UCB

will call the wrapper, which will in turn call the handwritten code. [Figure 14-3](#) illustrates the SystemBuild and AutoCode calling sequence for user code.

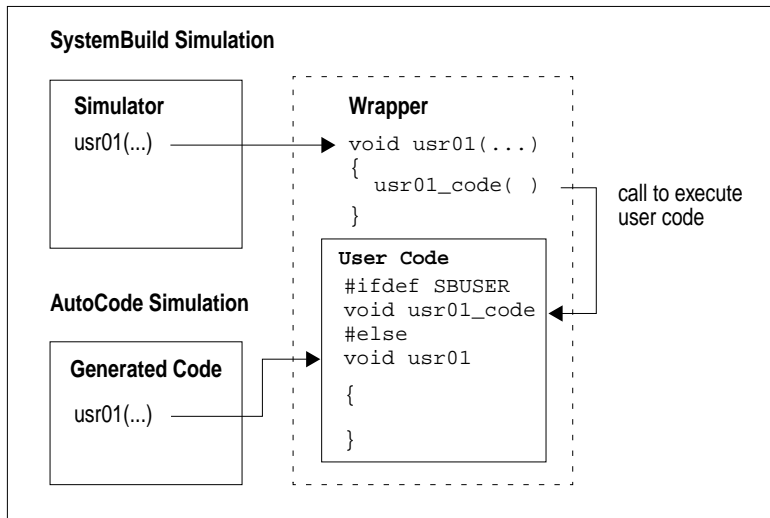


FIGURE 14-3 SystemBuild and AutoCode Share User Code

To use a variable interface UCB for simulation, you must:

- Specify the variable interface in the UCB dialog ([Section 14.3.4](#)).
- Create user code that conforms to the specification within the UserCode block, including function name, inputs, outputs, etcetera.
- Write a C wrapper that calls the user code.

14.3.2 Using a Wrapper so that SystemBuild Can Simulate Code Written for AutoCode

As depicted in [Figure 14-3](#), the UCB must call a wrapper that then calls the user code. A simple example of a single file that includes a wrapper and the user code is available in `$SYSBLD/examples/variable_ucb/sample/samp_vucb.c`. This code can be used for both SystemBuild and AutoCode. [Figure 14-4 on page 14-39](#) highlights portions that are of particular interest.



FIGURE 14-4 Excerpts from samp_vucb.c

14.3.3 Writing a Wrapper

The purpose of the wrapper is to interface the `sim` UserCode interface to your variable interface user code. The wrapper must convert the input and output data from the `sim` Usercode interface to match the variable interface user code. Please, examine the sample files `test_vucb.c` and `samp_vucb.c`; each contains a wrapper for UserCode functions. To copy the files to your current working directory, issue the following commands from the Xmath command area:

```
copyfile "$SYSBLD/examples/variable_ucb/test/test_vucb.c"
copyfile "$SYSBLD/examples/variable_ucb/sample/samp_vucb.c"
```

The wrapper must perform the following tasks:

1. Create local variables that match the datatype and shape of variable interface UCB inputs and outputs.
2. Convert the data from the SIM interface to the local variables (see [Converting Data From the SIM Interface](#)).
3. Initialize the INFO status record (if needed) to indicate the appropriate phase (INIT, OUTPUT or STATE).
4. Call the variable interface UserCode using the local variables as arguments to the function.
5. Convert the local variables representing the output data back into the `sim` interface.
6. Reflect any error code (if using the INFO status record) back into the `sim` interface.

Converting Data From the SIM Interface

The `sim` interface presents all input and output data as floating-point data. Since the variable interface UserCode will most likely have datatypes other than float, the `sim` data must be converted to the appropriate datatype.

To perform datatype conversion, use the AutoCode SA Library header files and some of the macros defined within them. Refer to the *AutoCode Reference* manual and the

SA Library source files for additional discussion. Several of these macros are useful for conversion, in particular:

`I_Fpr()` Conversion from floating-point to Integer (using protection and rounding).
`SB04_Fpr()` Conversion from floating-point to SignedByte, radix 04.
`ULn02_Fpr()` Conversion from floating-point to UnsignedLong radix -02.

Converting Data Back to the SIM Interface

After executing the variable interface UserCode, you must convert the output data back into the SIM interface. This implies converting the output data and copying it into the SIM interface. It is acceptable to typecast the integer and logical values into the SIM interface. However, for fixed-point data, you must use a conversion routine. Refer to the SA Library files for various conversion macros. The following conversion macros can be useful for this purpose:

`F_SS03()` Conversion from SignedShort radix 3 to floating-point.
`F_UB06()` Conversion from UnsignedByte radix 6 to floating-point.

14.3.4 Specifying the Variable Interface

The UserCode block parameters are fully documented in the online help. To see this information, type `help usercode` from the Xmath command line, or press the **Help** button from the UserCode block dialog.

Setting Variable Interface Parameters

To use the Variable Interface, go to the Parameters tab and set the parameter Interface Type to Variable. The Time Argument and Info Argument fields are enabled when the Interface Type is set to Variable; choose the appropriate action (Yes or No).

Specifying Datatypes

Input datatypes are inherited from source blocks. Output datatypes are specified on the UCB Outputs tab.

Specifying Input Shapes

Input labels and names determine the structure of input data to the user code. See [Creating Sequential Names for Vectors or Matrices](#). The precedence for creating the shape (scalar or vector) is determined by:

1. Input names (specified on the UCB Inputs tab). These override names inherited from the input signal.
2. Output names (inherited from the signal source block).
3. Output labels (inherited from the signal source block).
4. UCB block name.

Specifying Output Shapes

Output labels and names determine the structure of the output data from the user code, as explained in [Creating Sequential Names for Vectors or Matrices](#). The precedence for creating the shape (scalar or vector) is determined by:

1. Output names (specified on the UCB Outputs tab).
2. Output labels (specified on the UCB Outputs tab).
3. UCB block name

14.3.5 Running a Variable Interface Example

Use the following commands to move the example to your current working directory.

```
copyfile "$SYSBLD/examples/variable_ucb/test/test_vucb.cat"  
copyfile "$SYSBLD/examples/variable_ucb/test/test_vucb.c"
```

Simulating the Variable Interface UCB in SystemBuild

1. Load the file `test_vucb.cat`. This file contains both model data and Xmath variables.
2. Open the SuperBlock `test_vucb`. On the Parameters tab, make the Time Vector `t` and the Input Variable `u`. Give the output vector the name `tv_sb`.
3. Click Plot Outputs and Typecheck, then hit **OK**. The simulation results will be shown in a strip plot, and saved to the variable `tv_sb`.

Generating Code for a Variable Interface UCB in SystemBuild

In the SystemBuild Catalog Browser, select the SuperBlock test_vucb. To generate code, select Tools→AutoCode, then click OK.

After generating the code, you can compare the generated code outputs with the simulation outputs. For instructions on compiling and linking the user code, see the section titled "Standalone Simulation" in the *AutoCode User's Guide*.

14.4 UCB Programming Considerations

Several details must be considered when programming UCB subroutines:

- Take care not to overwrite areas of memory used by the simulator executable. The names of any subroutines you add, as well as any COMMON blocks, must be different from those already used in the product. To ensure uniqueness, prefix all your global variables and internal function names with ZZ, which is never used inside the simulation engine.

Use function names that begin with `usr` or `iusr` to avoid conflict with other symbols defined in the simulation engine.

- If using dynamic memory allocation/deallocation, make sure that your code does not write beyond the allocated buffer; write into a buffer that has been deallocated, etc.
- If a UCB is included in a system more than once, it must use static memory very carefully, for each instance of an identically-named UCB consists of another call to the user-supplied code, with potentially different parameters, but the same static variables.
- If you have existing code that evaluates derivative and output equations for other integration packages, it is probably best to insert a call to that routine in a UCB, rather than writing the equations directly into the UCB. In that way your equations will be usable under both SystemBuild and your other packages.

14.5 Building, Linking and Debugging UCBs

The simulator is responsible for reading information about UserCode from the model and other sources, then taking that information and collecting, compiling and executing UserCode. This section explains this process. As much as possible, the discussion will be platform and language independent, but differences between platforms and languages are noted.

After the simulator has completed the UCB compilation process, all of the user code will reside in a shared library (on Windows, a Dynamic Link Library) with the name `simucb.ext`, where the extension is platform specific. This is referred to as the UCB shared library.

CAUTION: Before MATRIX_x Version 6.0, new simulator executables (`sim.exe.lnx` on UNIX systems or `simexe.exe` on PC systems) were created as the end result of this process.

You must remove old versions of a rebuilt simulator executable before using any of the following guidelines. Failure to do this will cause unexpected runtime failures.

14.5.1 Collecting UserCode Files

UserCode files can be specified from the UserCode block parameter tab, with the keywords `csource` and `fsource`, in the makefile itself, or with the `ucbcode loc` simulation option. This section discusses each method.

NOTE: If you have both C and FORTRAN UCB source files, make sure that they do not have the same root name (ignoring the extension). For example, it is an error to have UCB source files `myucb.c` and `myucb.f` in the same simulation.

UCB Block Parameter Form

The primary location for entering a file name is the UCB block parameter form. This filename may be a simple file name (i.e. `usrcode.c`), or, it may contain a relative pathname (for example, `usrcodedir/usrcode.c` on UNIX systems), a complete path name (including a drive letter on Windows systems), or environmental variables (for example, `%ucbcode loc%\usrcode.c` on a Windows system). Only environmental variables that exist before SystemBuild is invoked will be recognized by the simulator.

CSOURCE and FSOURCE

Additional source code files may be specified using `csource` (for C language files) and `fsource` (for FORTRAN language files). `csource` and `fsource` files may be specified with the `linksim` function or the `SETSBDEFAULT` command. For example,

```
linksim ("top", {csource="ucb1.c ucb2.c", fsource="ucb3.f"});
SETSBDEFAULT {csource="ucb14.c ucb15.c", fsource="ucb11.f"}
```

If `csource` or `fsource` is set using `SETSBDEFAULT`, the file names will be combined with those on the `linksim` command line to create the file list.

Specifying Sources in the makefile

When the simulator creates the UCB shared library, it uses a platform-specific makefile. This makefile resides at `$SYSBLD/src/makefile` for UNIX systems, and `%SYSBLD%\src\makeucb.mk` for PC systems. It may be copied into the local project directory, and if it exists there, the simulator will use the local makefile instead of the default in the standard location.

All makefiles have a section at the top that is safe to modify. Modifications made outside this area can corrupt the UCB shared library creation process.

UNIX

For UNIX systems, simply list any additional C language source code files at the end of the line that reads:

```
CSOURCES =
```

Add any FORTRAN language source code files at the end of the line that reads:

```
FSOURCES =
```

Windows

For PC systems, add all source code files (C or FORTRAN) to the line that reads:

```
SOURCES =
```

Reusing Sources from the Previous Simulation

If a UCB shared library exists in the project directory before this process starts, the simulator will attempt to include all of the objects in the previously existing shared library in the new one. This makes it possible to quickly switch between different models that contain different UCBs.

Specifying Another Location for UCB Code

The `sim` command line option `ucbcodeLoc` may be used to specify a location for user code different than the local directory. If this option is supplied by the user, the simulator will assume that all UCB source files, as well as the UCB shared library, will be located in the directory specified with the `ucbcodeLoc` option (except any source code file that has a complete path name as its specification, in which case,

the simulator will look for the file there). This location may be set using SETSBDEFAULT. `ucbcode loc` is designed to support multiple engineers accessing the same infrequently changed shared UCB library so that team members don't need to keep local copies of the entire team's UCB source.

In this case, all team members should have the following line in their `startup.ms` file.

```
setsbdefault {ucbcode loc="/path_to_team_UCB_code" }
```

14.5.2 Compiling and Linking User Code

The simulator compiles each of the identified source code files (if necessary) using `make` on all platforms (with a different, platform dependent, `make` file on each platform). Once the `make` facility has determined that each source code file has been successfully compiled, the source code objects are linked into the UCB shared library. Any compile or link errors will be reported by the simulator, which will then terminate.

Windows

On Windows platforms, a Dynamic Link Library (DLL) is created. Because a DLL is a name-resolved stand-alone entity, all symbols referred to in the user's code must exist in the DLL before it can link. If a UCB calls a user-supplied utility function that function has not been made available to the simulator in one of the ways described above, the DLL will fail to link.

UNIX

On UNIX platforms, a shared library is created; shared libraries are not name resolved. Although the shared library is not expected to link completely, it is your responsibility to make sure that all user-supplied code is contained in the shared library. The simulator will verify that all UCB routines are included in this library, but it cannot verify that user-supplied utility routines are included in the shared library. If, during simulation, UserCode attempts to call a non-existent function, the simulator will immediately stop executing with a dynamic linker error.

Supported Compilers

ISI tests the UCB capabilities of the simulator using the compilers (and compiler versions) that are officially supported by ISI. If an unsupported compiler is used to compile and link UCB shared libraries, ISI cannot assist you with problems encountered while compiling code, creating the UCB shared library, or any run-time problems. The list of supported compilers is available in the MATRIX_X System Administrator's Guide, or on our website at <http://www.isi.com>.

14.5.3 Debugging User Code

Follow these guidelines when debugging user code:

1. To debug user code in the simulation environment, the code must first be compiled and linked with debug information.

UNIX

On UNIX platforms, the code is compiled and linked with the debug information automatically.

Windows

On Windows platforms, you will need to copy the make file, `makeucb.mk` into the local project directory (even if using the `ucbcodeLoc` option). You can do this with the following Xmath call:

```
copyfile "$SYSBLD/bin/makeucb.mk"
```

Edit `makeucb.mk` with a text editor, and make sure the top portion of the make-file contains the line:

```
DBG = Yes
```

Next, create the UCB shared library using the `linksim` command. This will guarantee that when you eventually enter the debugger, that the UCB shared library will already be created and loaded into the simulator. For more information on the `linksim` command, consult the online help.

2. From the Xmath command area, type the following command:

```
debug simexe
```

3. Call the `sim` function normally, either from the Xmath command line, or the SystemBuild editor Tools menu pulldown. Once the simulator is invoked, a GUI dialog will appear.

CAUTION: Do not hit OK in this dialog until the debugging session is ready for it (step 5).

Windows

On PC systems, an OS dialog will also come up, notifying you that a debug exception has occurred, and asking if you would like to invoke the debugging environment. Do so.

UNIX

On UNIX systems, you will have to bring up the debugger and attach it to the running process. Return to the Xmath GUI dialog, and take note of the command listed in the window. Execute that command at the OS command line. This will invoke the debugger, and attach it to the simulation process.

4. Once in the debugging environment, do any initialization you like (setting break points, data watch points, etc.).

NOTE: On SunOS and Solaris systems, it is important that you also execute the command:

```
ignore USR1
```

Remember that the names of the functions directly called by the simulator will generally have a trailing underscore (UNIX systems) or will be in all capital letters (Windows systems) due to the mixed C/FORTRAN language support in the simulator.

5. After the debugging session is initialized, make sure that the simulator is running (by using the **cont** command on most UNIX debuggers, or pressing the **F5** key in the Windows environment).
6. Once the simulator is running, you may hit **OK** in the Xmath GUI dialog. At this point the simulator will resume, and the debugging session may continue.

Because the simulator does not terminate at the end of each simulation, you can keep the debugging session open through multiple subsequent simulations, provided the simulator does not crash, the command **undefine simexe** is not issued from the Xmath command area, and Xmath remains running.

14.6 Posting Error Indications

Use the ISI-supplied function `stdwrt` in your C or FORTRAN UCB to write messages to the Xmath window. `stdwrt` is a void function that takes one string argument containing the message to be displayed in Xmath.

```
void stdwrt(char * message)
```

The prototype for this function is in the file `$ISIHOMe/sysbld/src/matsrc/h`. Be sure to include this file in your C language file. Here's a simple example of its use:

```
#include "matsrc.h"
int a = 1;
```

```
int b = 2;
  if (a != b)
    stdwrt("The Two values are not equal\n");
```

Notice that a newline character ends the string argument. `stdwrt` does not automatically add newline characters.

If you must write messages from your own FORTRAN subroutine, use logical I/O units numbered 99 or higher. Here's a simple example:

```
CHARACTER*80 BUFFER
WRITE(BUFFER,10) RPAR(4),IPAR(3)
10 FORMAT('RPAR(4)=' ,1PE12.4,' and IPAR(3)=' ,I2)
CALL STDWRT(BUFFER)
```

At simulation time, extensive error checking is performed, and a set of standard error messages is provided, which the UCB can use as well; see the list below. The numbers provided are message numbers for error reporting by UserCode Blocks. The status word `IINFO(1)` is provided to let a UCB return error information to the simulator program. If the UCB returns the value -2 in this location, a generic error message is generated. But a UCB can also use the following formula to have the simulator program post a specified message:

$$\text{Error Indication} = -(10 * M\# + 2)$$

where `M#` is the message number. These error indications are useful to SystemBuild and its UCB technology; they are not compatible with AutoCode. For example, to post the message "Square Root of negative number," the UCB simply returns -62 in the `IINFO(1)` status word.

Simulation Errors

1. `SIM_ERROR`: Division by 0.0 produces infinity

If the second input vector to a divide block, `u(NO + 1: 2: NO)`, contains a zero value, then this simulation error occurs.

2. `SIM_ERROR`: Raise 0.0 to a non-positive power.

A simulation error occurs when the input to an exponential block and the constant power is less than or equal to zero.

3. `SIM_ERROR`: Both arguments to `ATAN2` are zero.

The output of the `arctangent2` function is undefined when both inputs are zero.

4. SIM_ERROR: ASIN or ACOS argument out of range.

The input to the arcsine or arccosine function block must be in the range from -1 to 1. The output of this function is in the range 0 to π .

5. SIM_ERROR: Natural log of zero or negative number.

A simulation error occurs if any input to the log block is less than or equal to zero.

6. SIM_ERROR; Square root of negative number.

A simulation error occurs if any input to the square root block is negative.

7. SIM_ERROR: Incoming data not in range of table.

For the two-input system, no extrapolation is performed on evaluation to extend the ranges of the inputs to the LinearInterp block. If either input range is exceeded in the analysis of this block, an error occurs.

8. SIM_ERROR: raise negative number to non-integer.

A simulation error occurs when the input to an exponential block represents a floating point power and the constant is less than zero.

9. SIM_ERROR: Overflow in $y = \text{EXP}(u)$ function

Quantity out of range of hardware.

15

Fixed-point Arithmetic

This chapter describes SystemBuild Fixed-point arithmetic. This feature emulates two's complement binary arithmetic, allowing systems running AutoCode generated code (for example, in embedded processors) or RealSim to interface directly with inexpensive processors that do not support floating-point math.

The following SystemBuild blocks support fixed-point arithmetic.

AbsoluteValue	BilinearInterp	Constant	ConstantInterp
Gain	CrossProduct	DataStore	DataPathSwitch
DeadBand	DotProduct	ElementDivide	ElementProduct
Limiter	LinearInterp	LogicalOperator	MatrixTranspose
Preload	ReadVariable	RelationalOperator	Saturation
ScalarGain	ShiftRegister	Summer	TimeDelay
TypeConversion	WriteVariable		

Fixed-point blocks are compatible with other SystemBuild blocks, the simulator, and other features of SystemBuild. Thus, models may contain floating-point, integer, logical, and fixed-point components in any mixture.

Fixed-point arithmetic only operates in discrete SystemBuild models; that is, the SuperBlocks in the part of the model using fixed-point must be discrete free-running, enabled, triggered, or procedure — never continuous.

Fixed-point arithmetic differs from floating-point arithmetic in several ways:

- Addition and multiplication are not associative.
- Overflow can occur if the output number is too big for the datatype.
- Questions of precision and significance arise because of the fixed wordsizes and ranges of the datatypes.

The SystemBuild user interface provides several access points to fixed-point information:

- The SystemBuild connection editor reports on the datatypes of fixed-point signals.
- The Inputs tab allows you to specify fixed-point input datatypes.
- The Outputs tab allows you to specify fixed-point output datatypes.
- The Gain and ScalarGain block dialogs allow you to specify a radix position for the gain parameter.
- User-defined datatypes (also called usertypes) provide an aliasing capability for SystemBuild datatypes. It is possible to assign to a name that is relevant to your problem to a datatype. Although usertype support is a general SystemBuild feature, it is particularly convenient for fixed-point arithmetic, where you may need to switch datatypes after each simulation. For more information on usertypes, see [Section 15.5 on page 15-40](#), and for a full treatment of datatypes, see [Section 4.5 on page 4-19](#).
- For simulation or code generation, the SystemBuild Analyzer performs all necessary datatype checking, based on datatype checking rules tabularized in [Section 15.3 on page 15-23](#). For operation of the Analyzer, see [Section 7.5 on page 7-10](#).
- The block set for use with fixed-point numbers includes arithmetic, logical, and piecewise linear blocks. For the complete list see [Table 15-1 on page 15-23](#).

AutoCode provides full support for fixed-point arithmetic in generated code. See "Fixed-point Arithmetic" in the *AutoCode User's Guide*.

15.1 Introduction to Fixed-point Arithmetic

For computation with real numbers, floating-point representation and arithmetic is the usual approach; however, floating-point calculation is slow compared to integer calculations. Coprocessors can speed up floating-point calculations, but integer arithmetic is still faster in most cases. Libraries that emulate coprocessors are also notoriously slow. Fixed-point representation and arithmetic is an alternative approach that takes advantage of processor instructions. This computational method (and corresponding notation) uses scaled integers to represent floating-point numbers, thereby avoiding the overhead of floating-point calculations.

The rest of this chapter discusses the issues involved in simulating models and generating fixed-point C code from a SystemBuild model, with integer datatypes and scaling, and the options available to produce fixed-point code.

15.1.1 Fixed-point Number Representation

Fixed-point arithmetic uses integer datatypes, in which a fixed number of the bits (fractional part) are used to represent the fractional component of a number and the rest (integer part) are used to hold the integer component. This allows real numbers (or an approximation of them) to be stored in integers. Fixed-point numbers are restricted to 8-bit, 16-bit and 32-bit representations. This limitation is referred to as *wordsize*; thus the possible wordsizes for fixed-point numbers are 8, 16, and 32 bits.

Throughout this chapter, m refers to the number of bits in the fractional part (also referred to as the *radix position*), and n denotes the number of bits in the integer part. Thus, a fixed-point number's wordsize is $n + m + 1$ if it is signed or $n + m$ if unsigned. The precision of a fixed-point number is 2^{-m} , its maximum significance is $2^n - 1$, and the range of the fractional part is $[0, 1 - 2^{-m}]$. The range of an unsigned fixed-point number is $[0, 2^n - 2^{-m}]$. The range of a signed fixed-point number is $[-2^n, 2^n - 2^{-m}]$.

[Figure 15-1](#) shows an eight-bit unsigned fixed-point number with radix position 4, meaning that four bits are for the fractional part and leaving four bits for the integer part. The precision is $1/16$, the range of the fractional part is $[0, 15/16]$. The maximum significance of the number is 15, and the range of the number is $[0, 15 \frac{15}{16}]$.

[Figure 15-2](#) on page 15-4 shows an eight bit signed fixed-point number with radix position 6. The precision is $1/64$, the range of the fractional part is $[0, 63/64]$, the maximum significance is 1, and the range of the number is $[-2, 1 \frac{63}{64}]$.

Note that when the radix position is zero, the [Figure 15-1](#) example is an unsigned integer with range $[0, 255]$, and the [Figure 15-2](#) example is a signed integer with range $[-128, 127]$.

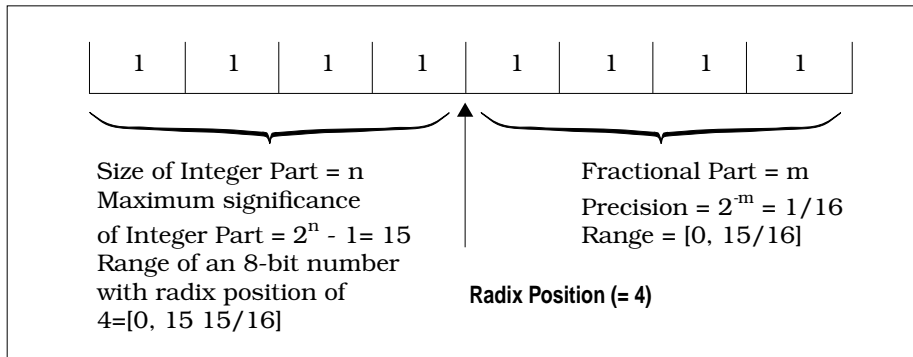


FIGURE 15-1 An Unsigned Fixed-point Number with 8 Bits and Radix Position 4

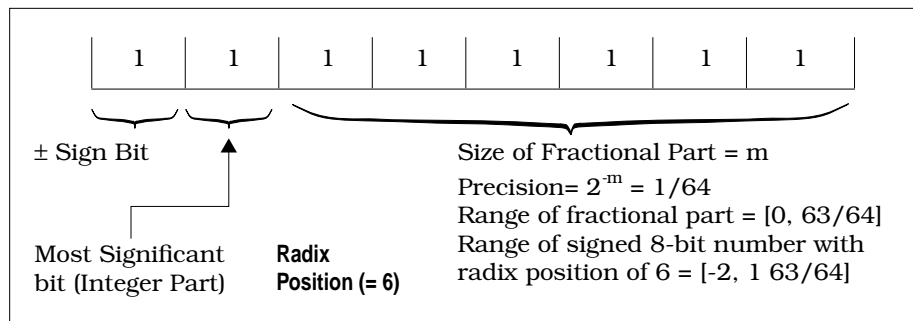


FIGURE 15-2 A Signed Fixed-point Number with 8 Bits and Radix Position 6

The *scaling* referred to in fixed-point arithmetic is directly related to the radix position, m , of the fixed-point datatypes. The scale factor is 2^m . As an example, the number 1.25 is stored as 00000101 in an unsigned 8-bit fixed-point datatype with ($n=6, m=2$). In regular two's complement integer format this binary number is equivalent to the decimal number 5, which is 1.25 times the scale factor 4. Since the scaling is implicit in the radix position, we will henceforth refer only to the latter.

Obviously, two different fixed-point numbers cannot be operated on without knowing their radix positions. Were the wordsize limitations not a constraint, the following rules would provide the result radix position, wordlength, sign (that is, the

result datatype), and required operand alignment strategy for maximum precision in the basic algebraic operations, when only two operands are involved.

- Addition and Subtraction** The operand with the smaller radix position must be aligned with the larger radix position (this involves a left shift of the bit pattern), and the result radix position is the larger of the operand radix positions.
- Multiplication** The result radix position is the sum of the radix positions of the operands. No alignment is required for the operands.
- Division** The result radix position is the radix position of the numerator minus the radix position of the denominator. No alignment is required for the operands.

Note that these rules are the equivalent of decimal point alignment in decimal arithmetic.

In practice, the wordsize is constrained to values such as 8, 16, or 32, which may cause serious problems for the rules above. For example, the potential left shift involved in addition and subtraction may result in an overflow[†] if it causes a significant bit to “fall off” the left (most significant) side of the word boundary. A similar difficulty may arise in multiplication if the sum of operand radix positions plus the number of bits necessary for the integer part of the result is greater than the allowable product wordsize.

Moreover, the user has frequently determined a desired datatype for the result of the basic fixed-point operations. Such a determination might be a result of test data, experience, or previous simulations (fixed or float). Almost always they would impact the radix position and alignment rules discussed above.

For maximum flexibility, every block that supports fixed-point arithmetic allows the user to specify an output datatype. In most cases, such output datatype stipulation resolves operation datatype issues appropriately. Nevertheless, there are cases where more datatype information is required before the fixed-point operation is fully defined. These cases are explained in the following section.

We now turn our attention to basic fixed-point operations. In the examples below an unsigned fixed-point number will be represented as an ordered pair (i, rn) , where i is the integer and n is the radix position. Example: $(37, r4)$ is a fixed-point number with integer 37 and radix position 4 ($=2.3125$). For characterizing signed numbers we employ the triplet $(i, rn, sign)$.

[†] Overflow is defined as loss of significance; i.e., losing bits in the integer part of the number. The term underflow is used to mean overflow on a negative number.

15.1.2 Conversion Between Fixed-point Numbers

Users can convert one fixed-point datatype to another using a Signal Type Conversion Block. Algorithmically, if the wordsize is not changed, this operation amounts to a change in the radix position.

In AutoCode generated software, to change the radix position of a number, we use multiplication or division by bitwise shifting as explained below.

To increase the radix position of a fixed-point value by one, the integer is multiplied by two, by shifting the integer one position to the left. Note that this operation will overflow if the result cannot fit in the number of bits available. When this happens a significant bit “falls off” the left-hand side of the number. (A significant bit is 1 for unsigned or signed positive; 0 for a signed negative number.)

Similarly, to reduce the radix position of a fixed-point value by one the integer is divided by two, using a bitwise right shift. In this instance, loss of precision will occur if the least significant bit of the stored integer is a significant bit before the shift. On many computers right shifting a negative integer n positions will yield a different result from dividing it by 2^n . In SystemBuild and AutoCode the division standard is implemented, although AutoCode can be configured to use the shifting standard instead. The bit shifted into the most significant bit location of the integer is zero for unsigned numbers and equal to the sign bit for signed numbers.

[Example 15-1](#) shows conversion between fixed point numbers of different radix position.

EXAMPLE 15-1: Conversion of fixed-point numbers

0010.0101 (n1 = (37, r4), decimal value: 2.3125)

(\wedge indicates the imaginary position of the radix position within the binary data)

Align to radix position of 6 (i.e. shift left n1 by 2):

10.010100 (n2 =(148,r6), decimal value: 2.3125)

15.1.3 Addition and Subtraction

For fixed-point addition and subtraction, the radix positions of the two operands are aligned and then the numbers are added or subtracted, respectively. The result's radix position stays in the same position. [Example 15-2 on page 15-7](#) adds fixed-point

[Example 15-3](#) illustrates multiplication of fixed-point number $n1 = (37, r4)$ and fixed-point number $n2 = (65, r1)$ to produce fixed-point result number $n3$ with radix position 2, using 8-bit unsigned variables and 16-bit intermediate result.

EXAMPLE 15-3: Multiplication of Fixed Point Numbers of Different Radix Positions:

0010.0101 ($n1 = (37, r4)$, decimal value: 2.3125)

*

0100000.1 ($n2 = (65, r1)$, decimal value: 32.5)

00001001011.00101 ($n3' = (2405, r5)$, decimal value: 75.15625)

Align the radix position of $n3'$ to the radix position of the result (i.e. shift $n3'$ right by 3 bits). Place aligned results in $n3$:

1001011.00 ($n3 = (300, r2)$, decimal value: 75.0)

15.1.5 Division

Division of fixed-point numbers does not require pre-alignment of radix positions. The radix position of the quotient is the radix position of the dividend minus the radix position of the divisor. The stored value of the result is the integer division of the stored values of the operands. Depending on the processor, narrowing division may be performed internally in the division operation, meaning that a dividend is twice the length of the divisor, and the result has the same wordlength as the divisor. The benefit of narrowing division is that the dividend can be left-shifted to increase its radix position before the division to give a result with the maximum possible precision. [Example 15-4](#) illustrates division of fixed-point number $n1=(128, r4)$ by fixed-point number $n2=(224, r5)$ to produce fixed-point result number with radix position 7, using 8-bit unsigned variables and a 16-bit internal variable.

EXAMPLE 15-4: Division of Fixed Point Numbers of Different Radix Positions:

1000.0000 (n1 = (128, r4), decimal value: 8.0)

Left Shift dividend to increase its radix position to:

1000.000000000000 (n1' = (32768, r12), decimal value: 8.0)

÷

111.00000 (n2 = (224, r5), decimal value: 7.0)

000000001.0010010 (n3'=(146, r7), decimal value: 1.140625)

Store result in 8-bit unsigned variable n3:

1^0010010 (n3 = (146, r7), decimal value: 1.140625)

15.1.6 Relational Operations

For comparing two fixed-point numbers of the same wordsize the number with the smaller radix position (i.e., the number with the lower precision) is aligned with the number with the larger radix position, then the numbers are subtracted. The result is a logical value, with one indicating the comparison is true, and zero indicating false. [Example 15-5](#) shows greater-than comparison of fixed-point number n1 = (25, r3) and fixed-point number n2 = (17, r1), using 8-bit unsigned variables.

EXAMPLE 15-5: Relational Comparisons

In binary representation:

00011.001 (n1 =(25, r3), decimal value: 3.125)

>

0001000.1 (n2 = (17, r1), decimal value: 8.5)

Align the radix position of n2 to the radix position of n1 before comparing (i.e. shift n2 left by 2 bits). Place aligned results in n2':

00011.001 (n1 =(25, r3), decimal value: 3.125)

>

10000.100 (n2' = (136, r3), decimal value: 8.5)

FALSE

15.1.7 Overflow

An overflow occurs when the result of the operation is too large to fit in the number of bits available. Overflow protection is the ability to detect and correct overflows and underflows within fixed-point calculations. If overflow protection is enabled, the numeric results will exhibit saturation. If overflow protection is disabled, the numeric results show that a wrap occurs.

If a fixed-point compatible block performs numeric computations, you have the option of enabling fixed-point protection for that block alone. Go to the Output tab and enable the **Overflow Protection** checkbox. The following blocks have this option: BilinearInterp, ConstantInterp, Gain, CrossProduct, DotProduct, ElementDivide, ElementProduct, LinearInterp, LogicalOperator, MatrixTranspose, Preload, ScalarGain, Summer, and TypeConversion.

Overflow can be detected efficiently in assembly code by examining the processor status flags, but in C these flags are not available, and we must test results for consistency. [Example 15-6](#) shows overflow in the context of conversion of fixed-point number (32, r4) to a fixed-point number with radix position 7.

EXAMPLE 15-6: Conversion of Fixed-point Number with Overflow Protection.

```
0010.0000 (n1 = (32, r4), decimal value: 2.0)
Align to a radix position of 7 (i.e. shift left n1 by 3):

0.0000000 (n2 = [0, r7], decimal value: 0.0)*
Correct Overflow (i.e. use maximum number possible):

1.1111111 (n2 = (255, r7), decimal value: 1.9921875)

* Overflow has occurred while left-shifting n1.
```

[Example 15-7 on page 15-11](#) shows addition of fixed-point number n1=(249, r4) and fixed-point number n2=(7, r3) to produce a fixed-point result number with radix position 4, using 8-bit unsigned variables. Overflow protection is provided.

EXAMPLE 15-7: Addition with Overflow Protection

1111.1001 (n1 =(249, r4), decimal value: 15.5625)

+

0000.111 (n2 = (7, r3), decimal value: 0.875)

Align the radix position of n2 to the radix position of the result before adding (i.e. shift n2 left by 1 bit). Place aligned results in n2':

1111.1001 (n1' =n1= (249, r4), decimal value: 15.5625)

+

0000.1110 (n2' = (12, r4), decimal value: 0.875)

10000.0111 (n3' = (7, r4), decimal value: 0.4375)*

Correct Overflow (i.e. use maximum number possible):

1111.1111 (n3 = (255, r4), decimal value: 15.9375)

* Overflow has occurred while adding n1' and n2'.

15.2 SystemBuild Fixed-point

15.2.1 User Interface

The user of fixed-point math proceeds much as one does in the traditional SystemBuild paradigm, selecting, placing, parameterizing, and connecting blocks. Particular differences are the emphasis placed on assigning datatypes, and its influence on the ranges and precisions of data values. Two places are provided for specifying fixed-point data:

SuperBlock Input dialog box — The input tab view is accessed from the SuperBlock Attributes dialog. The Attributes dialog is accessed through the SuperBlock menu in the SuperBlock ID bar at the top of the SystemBuild display. Click on the Inputs Tab, then on the **Input Data Type**, and some of the datatype selections are offered. To obtain the complete list of choices, press the left mouse button or the **Input Data Type** field label until a vertical list appears, then move the mouse down to select the appropriate datatype, then release the mouse button. You can add the **Input Radix** as required.

Block Output dialog box — The Block Output tab view is accessed as one of the tabs in the block parameters dialog. Click on the Outputs Tab, then on the **Output Data Type**, and some of the datatype selections are offered. To obtain the complete list of choices, press the left mouse button or the **Output Data Type** field label until a vertical list appears, then move the mouse down to select the appropriate datatype, then release the mouse button. Then you can add the **Output Radix** as required.

In both these dialog boxes, you can also type a user defined datatype or usertype. This method is described in [Section 15.5 on page 15-40](#).

Note that Float (real floating-point numbers) is the default setting for inputs and outputs of the blocks discussed in this chapter.

15.2.2 Simulator

For simulating from the Xmath Command area, use the `fixpt` keyword to invoke fixed-point arithmetic. For simulating from the Simulation Parameters dialog, click on the **FixedPointSim** field to change it from no to yes. The `fixpt` keyword or **FixedPointSim** field are useful for comparisons and for studying the effect of quantization. Simply run a simulation with `fixpt` off, then with it on, and compare the results.

If `fixpt` is set, you can also use the `fixpt_round` keyword. If true, results of fixed point calculations are quantized by rounding to the nearest fixed point number and rounding away from zero if at a mid-point. If `fixpt_round = 0`, results of fixed point calculations are quantized by truncation. Default = 0. If `fixpt` is not set, `fixpt_round` has no effect.

Note that whenever `fixpt` is set:

- Datatype checking is always on.
- External inputs are always rounded
- Parameters are rounded, and their datatypes are not directly specified, but are derived from the block type.
- Saturation arithmetic is used: values are clipped with no wrapping.
- Calculation quantization is controlled by the `fixpt_round` keyword.

15.2.3 Building a Model and Demonstrating Overflow

When the size of a value becomes too large to fit in the datatype, overflow has occurred (thus, in the example in [Figure 15-1 on page 15-4](#), trying to place a value ≥ 16 in the register that holds the number causes overflow; in [Figure 15-2 on page 15-4](#), attempting to place a number $\geq +2$ or < -2 in the register similarly overflows).

Rather than showing garbage when a value has overflowed, the simulator returns the extremal number for the given output datatype (positive or negative depending on whether the number that overflowed was positive or negative). This is known as Saturation Arithmetic. In SystemBuild simulation, you can clearly see the effects of overflow by building a simple one-block model. [Example 15-8](#) shows this.

EXAMPLE 15-8: Demonstrating Fixed-point Arithmetic and Overflow

1. First create a discrete free-running SuperBlock; name it `Test`. The exact **Sampling Period** does not matter; accept the default value of 0.1.
2. Inside the SuperBlock place an `ElementProduct` block. Connect two external inputs to the inputs of the block, and connect the output to an external output. See [Figure 15-3 on page 15-13](#) for this model.

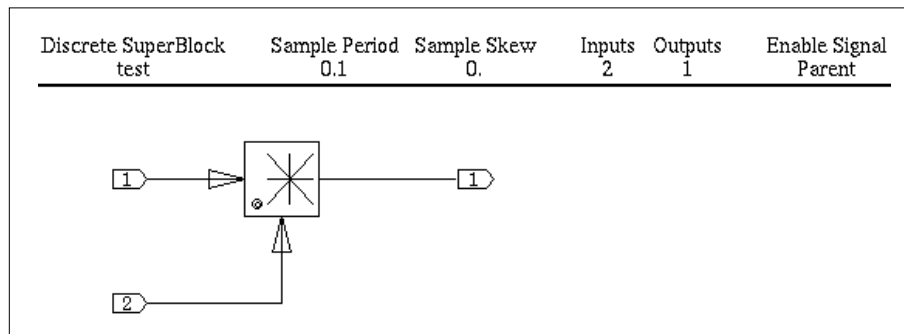


FIGURE 15-3 Model to Demonstrate Overflow

3. Click the left mouse button in the SuperBlock ID area to invoke the SuperBlock Properties dialog. On the Inputs tab, change the datatype of both inputs to **Signed Byte**.

See [Figure 15-4](#). This step is required to change the default input datatype (float) to the needed fixed datatype. Observe the bottom line of the dialog, where a Mnemonic (SB0 for Signed Byte, 0 radix) is shown, followed by Minimum,

Maximum, Resolution, and Scale Factor. Note that the range [minimum, maximum] is [-128, 127].

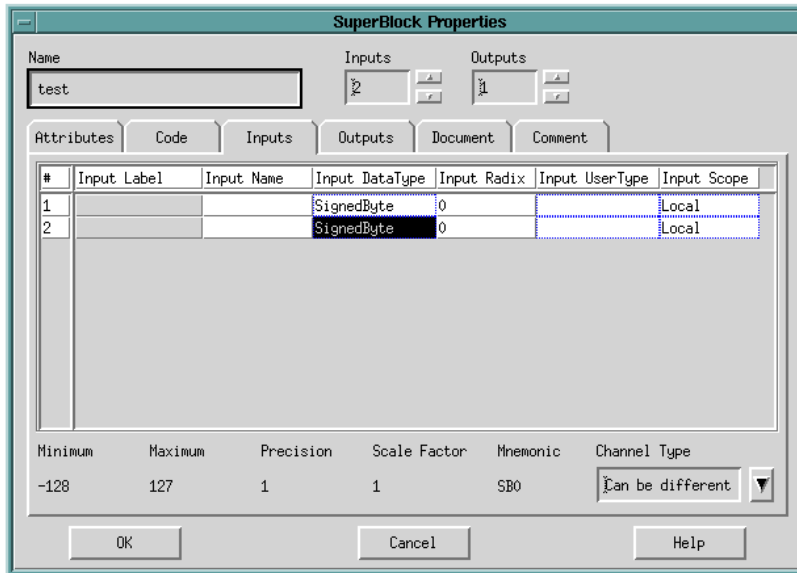


FIGURE 15-4 SuperBlock Dialog with Input Types Shown

4. Select the block and press return to raise its block dialog. On the Outputs tab, select Signed Byte from the **Output Data Type** combo box on the right. See 15-5.
5. Try a simulation without `fixpt` asserted. From the Xmath command area, type:

```
t = [0:.1:10]';
u = [2*t,t];
y = sim("test",t,u,{graph});
```

See Figure 15-6 on page 15-15 for the way the plot should look. The range of the `y` output is [0, 200], and, as we saw in the Output block dialog (Figure 15-5), that is outside the range of the output datatype, SB0. However, the `fixpt` keyword is not activated, and therefore floating-point arithmetic is performed, without the absolute fixed-point limitations on the output.

6. Try another simulation, this time with `fixpt` asserted:

```
y = sim("test",t,u,{graph,fixpt});
```

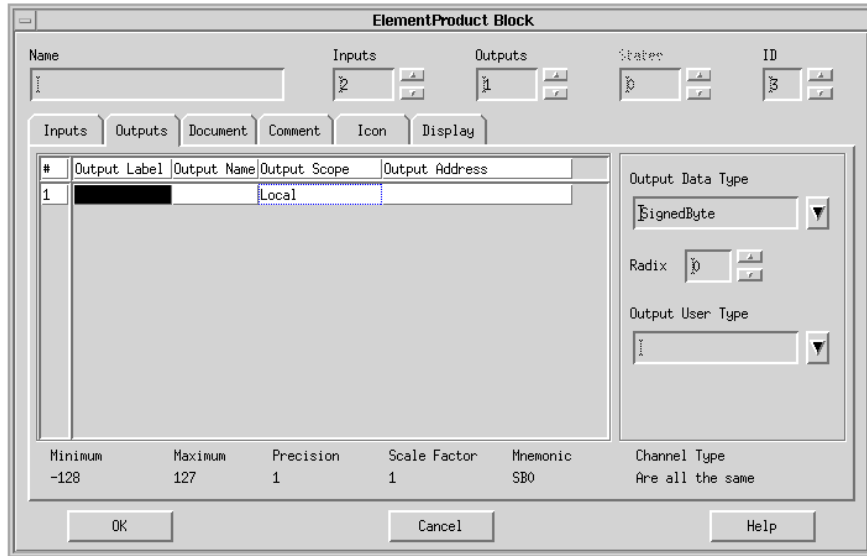


FIGURE 15-5 Outputs tab with Signed Byte Chosen

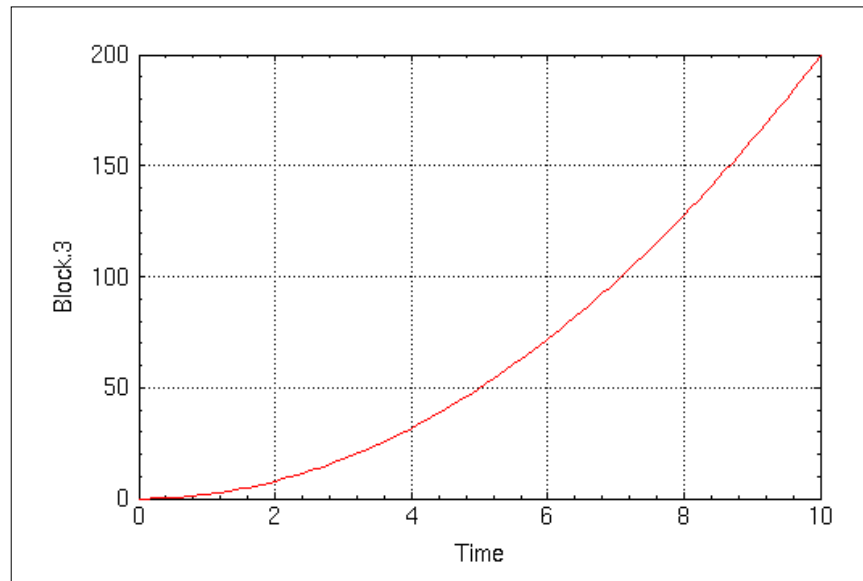


FIGURE 15-6 Fixed-point Plot without Overflow

The plot should look like [Figure 15-7](#).

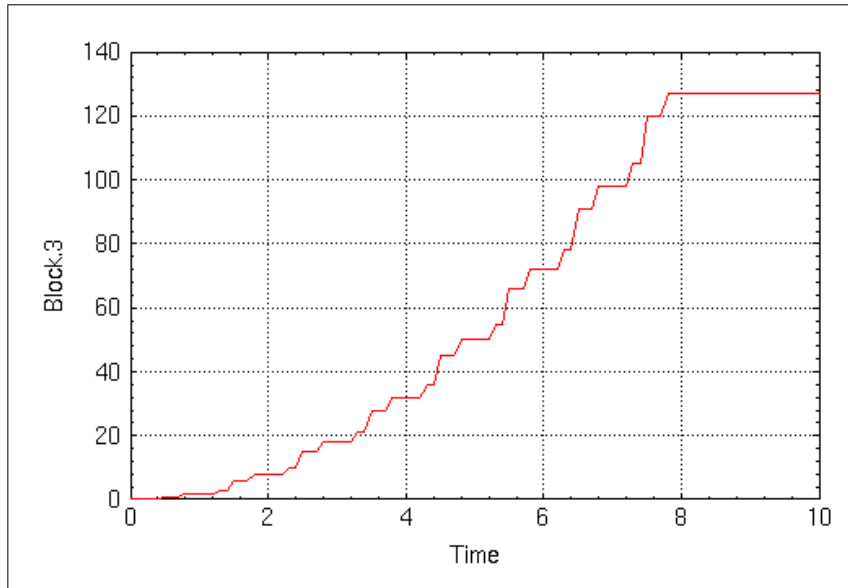


FIGURE 15-7 Fixed-point Plot with Overflow

Observe what is happening here; the value increases until 127, the maximum number for the SB0 datatype, is reached, then it flattens out. The range of output values for the simulation run would be [0, 200], which is outside the range of the SB0 output datatype as seen in [Figure 15-5 on page 15-15](#). SystemBuild responds to a data out-of-range (overflow), by truncating the output value at the extremal range point. If the output value stops overflowing the output datatype, the simulation stops returning maximal values.

15.2.4 Comparing Fixed- and Floating-Point Numbers

[Example 15-9 on page 15-17](#) shows a way to compare fixed- and floating-point numbers. One way to perform a quantitative comparison is to convert the fixed-point number to floating-point, then subtract it from a floating point input of the same value.

EXAMPLE 15-9: Comparing Fixed- and Floating-Point Numbers

Proceed as follows:

1. Start a new **SuperBlock**. Name it `test1`. Make it discrete and specify two inputs and one output.

Create the model shown in [Figure 15-8](#).

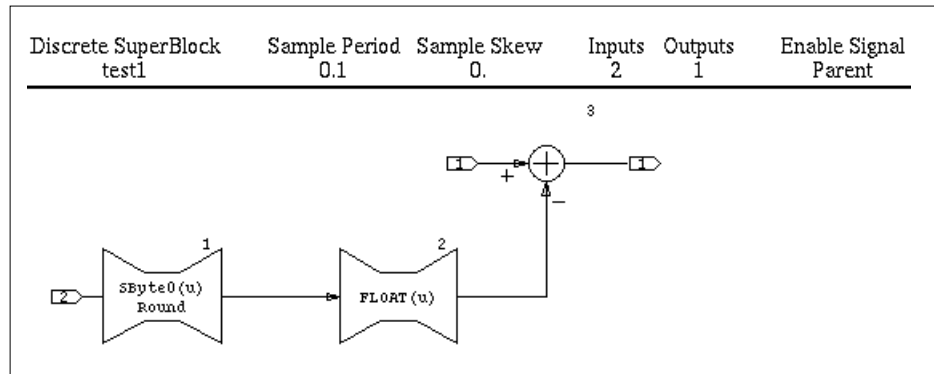


FIGURE 15-8 Example Showing how to Compare Fixed and Floating Types; SBO Case Shown

2. The parameterization of the TypeConversion block is as follows: block 1 accepts a floating-point signal and produces a fixed-point output. Change the **Output Type** from Integer to Signed Byte, and give it a **Radix Position** of 6; this is critical to the precision of the output. See [Figure 15-9 on page 15-18](#) for the parameters tab with these changes made.
3. Modify block 2 so that the **Input Type** is Signed Byte, the **Input Radix** is 6, and the **Output Type** is Float.
4. The way this model works is to accept the same sine-wave input on two different pins. One of the inputs (topmost) is kept in floating point format and run to the positive side of a summing junction. The other input is changed to SB6 (signed byte, Radix Position 6) and then changed directly back to floating point. The second input is then fed to the minus side of the summer. Thus, when the out-

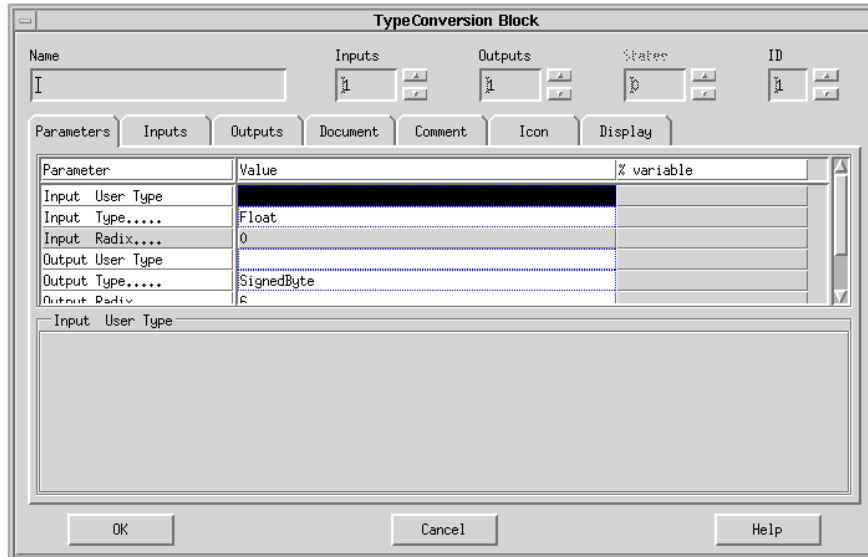


FIGURE 15-9 Block 1 Parameters Tab

put is plotted, it shows the difference between the floating-point input and its fixed-point form. In the Xmath command area, type:

```
t = [0:.1:10]';
u = [sin(t),sin(t)];
```

5. Select Tools→Simulate. In the dialog, specify t and u for the time vector and the input data variable, respectively. Make the Sim Type Fixed (Round) and enable **Plot Outputs** (i.e., plot). The Simulation Parameters dialog should look like [Figure 15-10 on page 15-19](#). Click **OK**.
6. See [Figure 15-11 on page 15-19](#) for the way the plot should look. From the plot, it appears that, at this Radix Position, the fixed and floating values of the input sine wave will never vary by more than about ± 0.008 . This agrees with the predicted range of differences, $\pm 1/2$ the resolution, which is equal to $2^{-6} = 0.015625$.
7. To observe the impact of the Radix Position on the precision of a vector of values, change the output radix position of block 1 to 0, and change the input radix position of block 2 to 0. Select Tools→Simulate and click OK to rerun the simulation (the dialog remembers your last settings). The output plot should look like [Figure 15-12 on page 15-20](#).

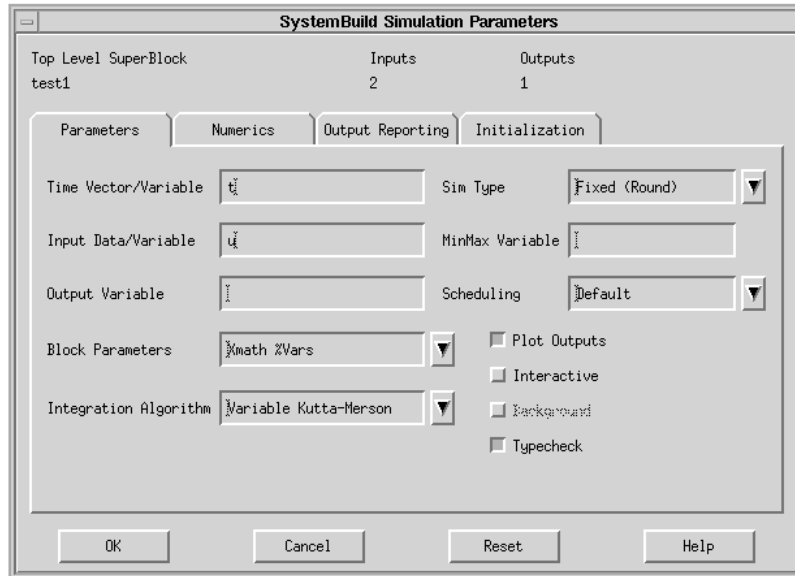


FIGURE 15-10 Sim Dialog, Ready for Simulating Comparison

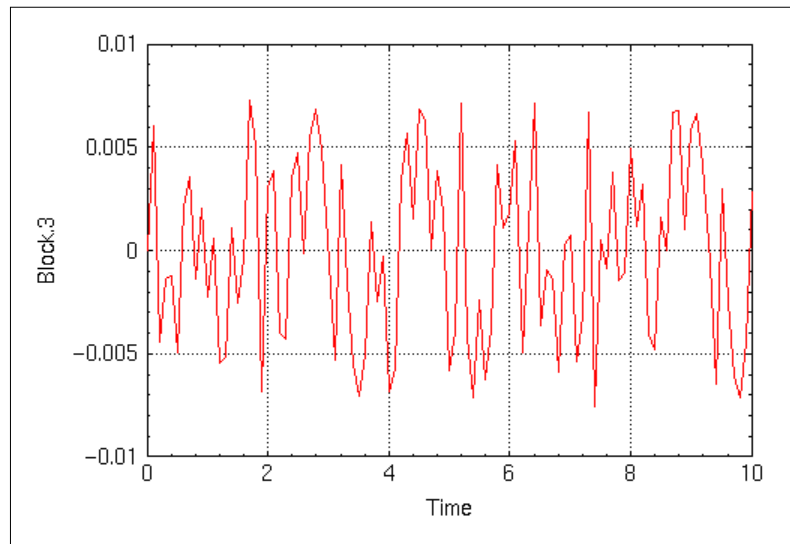


FIGURE 15-11 Plot of Comparison with Radix Position 6

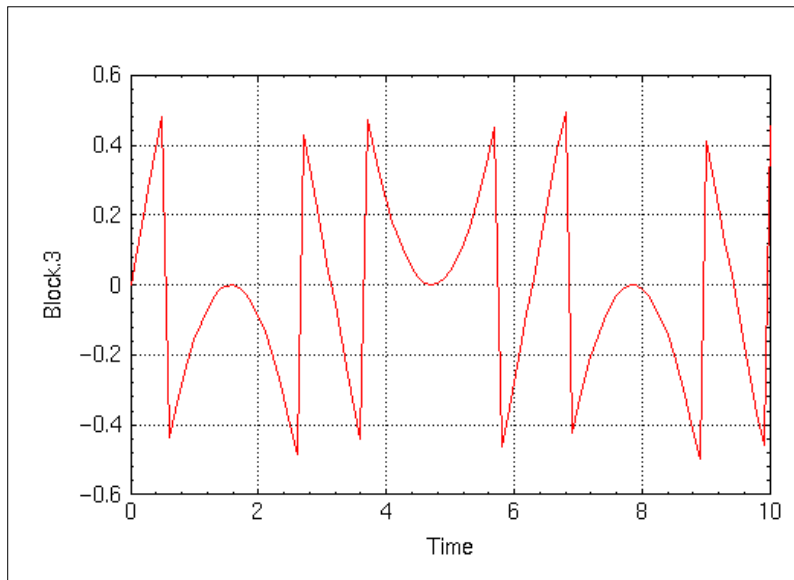


FIGURE 15-12 Plot of Comparison with Radix Position 0

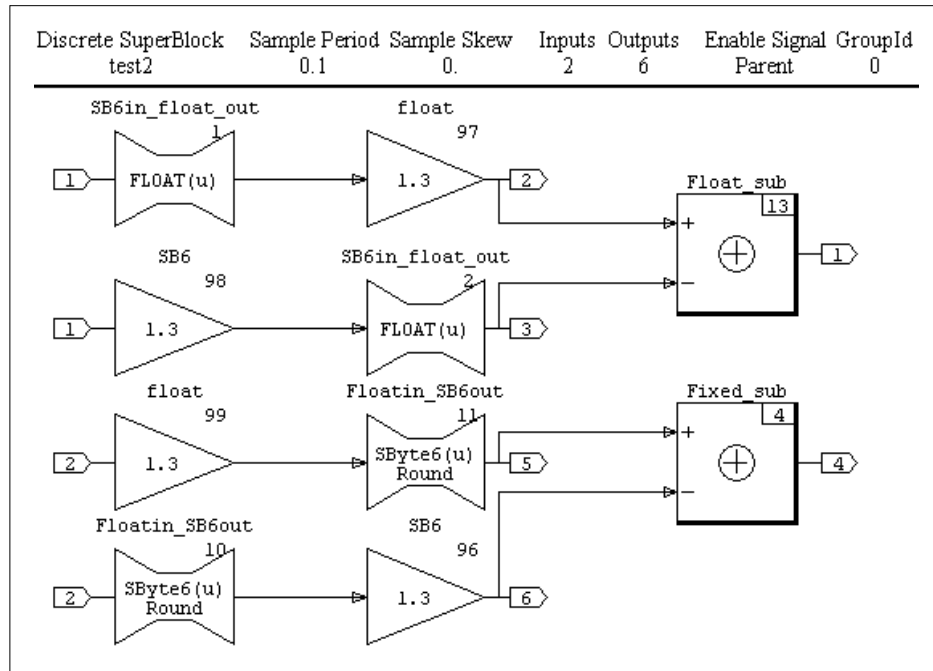
Examination of this plot shows that with a radix position of 0, the fixed-point and floating-point representations can differ by as much as ± 0.5 ; again this agrees with the predicted range of differences, which would be $[-1/2, +1/2]$.

15.2.5 Comparing the Effects of Different Conversion Sequences

The effect of converting between floating and fixed datatypes for different arithmetic operations varies according to the sequence of operations. This is illustrated in [Example 15-10](#), where fixed- and floating-point inputs are brought into a network of TypeConversion blocks and Gain blocks. Each input is changed to the other datatype and multiplied by a constant; it is the sequencing of these operations that is significant.

EXAMPLE 15-10: Effect of Datatype Conversion before and after Multiplication

1. Build up the model shown in [Figure 15-13](#). Give the model a name, `test2`, and make the SuperBlock discrete. Observe common points of the blocks:

**FIGURE 15-13** Model for Datatype Conversion before and after Multiplication

2. All the Gain blocks share the same **Gain** value, 1.3. The output datatype for the Gain blocks named `float` is `float`; the output datatype for those named `SB6` is `SB6`.
3. The two Conversion blocks named `SB6in_float_out` have input datatypes of `SB6` and output datatypes of `float`; the other two `TypeConversion` blocks have input datatypes of `float` and output datatypes of `SB6`.
 - The upper summing junction, `Float_sub`, has an output datatype of `float`; the other summing junction, `Fixed_sub`, has an output datatype of `SB6`.
 - The datatype of Input 1 is `SB6`; for Input 2 it is `float`.

- When the blocks are connected correctly, create a t-vector and u-matrix, then proceed to simulate the model:

```
t = [0:.1:10]';
u = [sin(t), sin(t)];
y = sim("test2",t,u,{fixpt, graph})
```

The plot should look like [Figure 15-14](#). Observe the many differences between the sequences of operation. You can design many other modes of operation to compare.

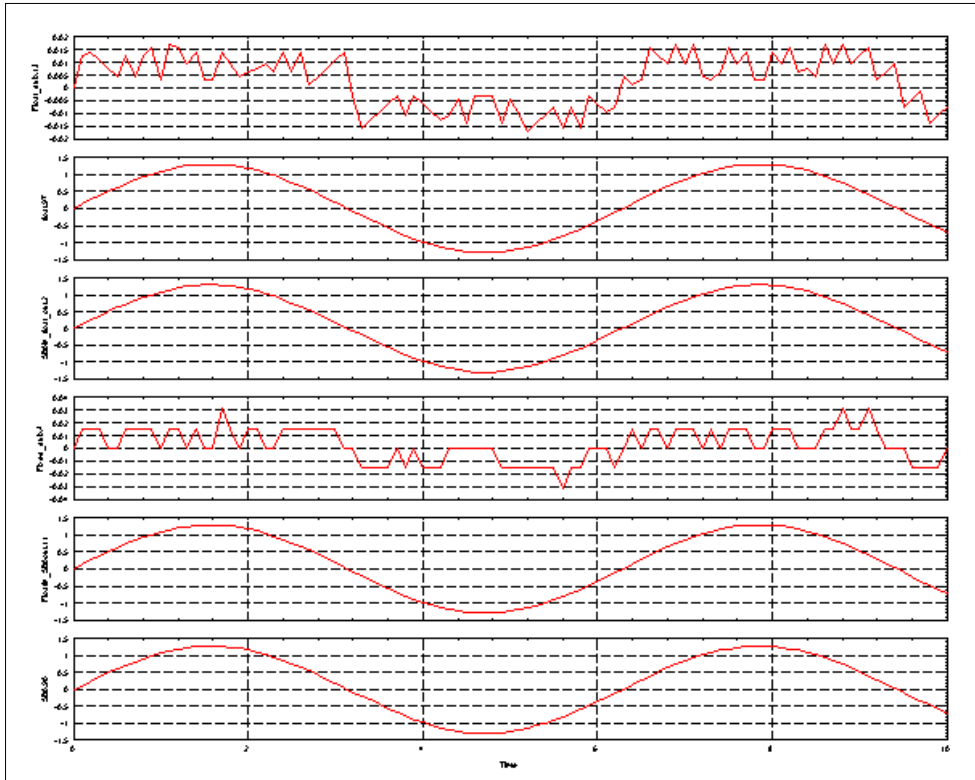


FIGURE 15-14 Plot of the Datatype Conversion before and after Multiplication Example

15.3 Fixed-point Blocks and I/O Datatype Rules

A selected group of SystemBuild blocks have been adapted for use with fixed-point arithmetic. [Table 15-1](#) lists the changed blocks, plus data typing I/O and block parameter rules for using the blocks with fixed-point arithmetic. Integrated Systems software supports four datatypes: floating-point, integer, logical, and fixed. Under fixed, we support 390 “sub-types”, referred to as fixed-point datatypes, which constitute the subject matter of this chapter.

Unless otherwise noted, in [Table 15-1](#) the terms “type” or “fixed type” refer to fixed-point datatypes. Fixed-point types are said to be the same only if the length, sign status (signed or unsigned), and radix position are identical.

TABLE 15-1 Blocks Compatible with Fixed Point, with Datatype Rules

Block Names	Input/Output Datatype Rules
SuperBlocks	
Datastore	<ul style="list-style-type: none"> ■ Fixed type of input to block must be the same as the register type.
ReadVariable WriteVariable	<ul style="list-style-type: none"> ■ Type of input to block must be the same as input parameter type. ■ All inputs must be the same type. ■ All outputs must be the same type.
Algebraic Blocks	
Gain	<ul style="list-style-type: none"> ■ All inputs must be the same fixed datatype. ■ All outputs must be the same fixed datatype. ■ Output wordsize must be \geq input wordsize. ■ See Section on page 15-32.
Summer ElementProduct DotProduct CrossProduct	<ul style="list-style-type: none"> ■ Each element in an input vector must be the same fixed type. ■ All outputs must be the same fixed type. ■ Output wordsize must be \geq input wordsize.
ElementDivide	<ul style="list-style-type: none"> ■ Each element in an input vector must be the same fixed type. ■ All outputs must be the same fixed type. ■ Numerator wordsize may exceed denominator wordsize

TABLE 15-1 Blocks Compatible with Fixed Point, with Datatype Rules (Continued)

TypeConversion	<ul style="list-style-type: none"> ■ Fixed type of input to block must be the same as input parameter fixed type. ■ All inputs must be the same fixed type. ■ All outputs must be the same fixed type.
Piece-wise Linear	
DeadBand	<ul style="list-style-type: none"> ■ All inputs must be the same fixed type. ■ All outputs must be the same fixed type. ■ Parameter type is the same as the fixed input type
Saturation Limiter	<ul style="list-style-type: none"> ■ All inputs must be the same fixed type. ■ All outputs must be the same fixed type. ■ Parameter fixed type must be the same as the output fixed type.
AbsoluteValue	<ul style="list-style-type: none"> ■ All inputs must be the same fixed type. ■ All outputs must be the same fixed type.
Preload	<ul style="list-style-type: none"> ■ All inputs must be the same fixed type. ■ All outputs must be the same fixed type. ■ Mag: Exact same as output type. ■ Slope: shrinkwrapped output wordlength.
Dynamic Blocks	
TimeDelay	<ul style="list-style-type: none"> ■ All fixed datatypes must be the same. ■ Initial output must be the same fixed datatype as the output.
Logical Blocks	
LogicalOperator RelationalOperator	<ul style="list-style-type: none"> ■ All datatypes are accepted, in any combination.
ShiftRegister	<ul style="list-style-type: none"> ■ All datatypes must be the same.
DataPathSwitch	<ul style="list-style-type: none"> ■ The first input can be any type. ■ All other input fixed types and the output fixed type must be the same; can be different from the first.

TABLE 15-1 Blocks Compatible with Fixed Point, with Datatype Rules (Continued)

Interpolation Blocks	
ConstantInterp LinearInterp	<ul style="list-style-type: none"> ■ All inputs must be the same fixed type. ■ Invals: same fixed type as inputs; ■ Outvals: same fixed type as outputs.
BilinearInterp	<ul style="list-style-type: none"> ■ Input 1 and Input 2 can be different fixed types. ■ Inval 1 must be the same fixed type as Input 1; Inval 2 must be same fixed type as Input 2; Outvals must be the same fixed type as outputs.
Matrix Equations	
ScalarGain	<ul style="list-style-type: none"> ■ All inputs must be the same fixed datatype. ■ All outputs must be the same fixed datatype. ■ Output wordsize must be \geq input wordsize. ■ See Section on page 15-32.
MatrixTranpose	<ul style="list-style-type: none"> ■ All inputs must be the same fixed datatype. ■ All outputs must be the same fixed datatype. ■ Output wordsize must be \geq input wordsize.
Constant	<ul style="list-style-type: none"> ■ All inputs must be the same fixed datatype. ■ All outputs must be the same fixed datatype. ■ Output wordsize must be \geq input wordsize.

15.3.1 Advanced Simulation Topics

This section provides information for advanced users on topics of Intermediate Datatypes, various simulation topics, 32-bit operations, and the Gain block.

Intermediate Types

The basic fixed-point algebraic operations are uniquely defined when all three of the following conditions are met:

1. The operation is binary (requires two operands) or unary (requires only one operand).
2. The operand types are defined.
3. The result datatype is defined.

Since the datatypes of all inputs and outputs are required for any block in a Super-Block, it follows that all fixed-point binary operations are well-defined in SystemBuild. Complications arise, however, when models are created that involve basic operations with more than two operands. An example of such a situation would be a summation block with four inputs, formally described by:

$$y = a + b + c + d \quad \text{EQ. 15-1}$$

For this discussion assume that the computation order in the above expression is equivalent to:

$$y = (((a + b) + c) + d) \quad \text{EQ. 15-2}$$

This ordering indicates that the computation would proceed by calculating the sum of a and b, which in turn will be added to c, etc. In other words, the following intermediate steps are involved in the calculation of y:

$$s1 = a + b \quad \text{EQ. 15-3}$$

$$s2 = s1 + c \quad \text{EQ. 15-4}$$

$$y = s2 + d \quad \text{EQ. 15-5}$$

The question arises as to what the datatype of s1 and s2 should be as the user has not specified them in SystemBuild. Variables s1 and s2 are *intermediate* variables and the types associated with them are called *Intermediate Types* or ITypes. Although these variables are transparent to the user, a consistent and predictable determination of their types is crucial to the final result. Generally speaking, for any block that combines n operators, n-1 intermediate types can be defined.

As another example consider the following case, which raises a subtle issue:

$$y = -a + b \quad \text{EQ. 15-6}$$

At first sight, it appears that [Equation 15-6](#) is the same as [Equation 15-7](#):

$$y = b - a \quad \text{EQ. 15-7}$$

and therefore requires no intermediate variables and hence no ITypes. But closer inspection reveals that [Equation 15-6](#) may be written as:

$$s1 = -a; \quad \text{EQ. 15-8}$$

$$y = s1 + b; \quad \text{EQ. 15-9}$$

Thus, [Equation 15-6](#) is not a simple subtraction but represents a negation operation followed by an addition. With two operands, negation and addition, it is not

surprising that [Equation 15-6](#) involves an intermediate datatype. This implies that [Equation 15-6](#) and [Equation 15-7](#), depicted in [Figure 15-15](#), may lead to results that are different (although numerically close). Using [Equation 15-7](#) is more efficient because it contains no ITypes and also it maps directly to one operation in both the simulation engine and the generated code. It is therefore the recommended usage.

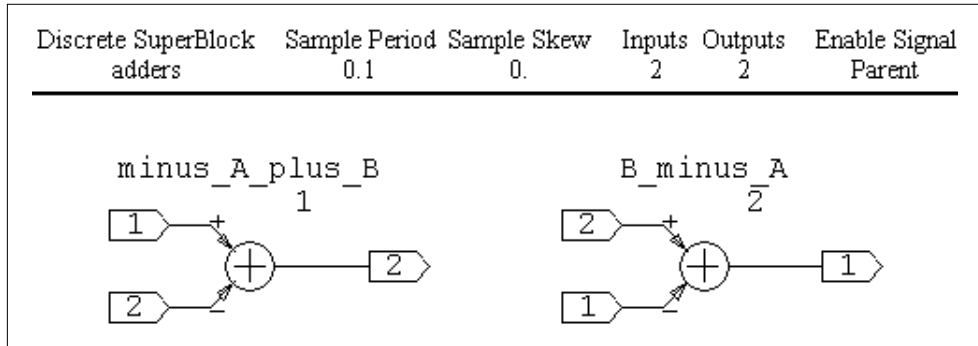


FIGURE 15-15 Add and Subtract Sequencing

Integrated Systems' built-in IType rules are based on two goals. Given the operand types and the nature of operation, the ITypes are derived such that:

1. The likelihood of overflow is minimized.

This makes sure that, possibly at the expense of precision in the least significant bits, the most significant bits in the operation are protected. (There is one minor exception to this rule; see Item (4) part (b) below.)

2. Wordlength promotions are minimized at the expense of precision in the digits to the right of the radix point.

The idea behind this goal is to strike a balance between precision and computational expense in the eventual code that is generated from the SystemBuild model. (Word promotions refer to increase in the size of the datatype wordlength; e.g., from 8 bits to 16 bits.) [Example 15-11](#) on [page 15-28](#) illustrates this phenomenon.

EXAMPLE 15-11: Need for Intermediate Types

1. Suppose that $z = ((x1 + x2) + x3)$ with:

Type(x1):= Signed Byte (8 bits) with Radix 3 (SB3)

Type(x2):= Signed Byte with Radix 4 (SB4)

Then we can use rules 1 and 2 to determine an IType for the sum of x1 and x2. To apply rule 1 we evaluate the extremal number that can possibly be produced by the addition. In the worst case, the largest possible sum (in an absolute value sense) occurs when $x1 = -16$ and $x2 = -8$, in which case $x1 + x2 = -24$. Thus the IType that would guard against overflow would be SB2 (signed byte radix 2).

2. In 1, if the datatype of x2 were changed to:

Type(x2):= Signed Byte with Radix 1 (SB1)

Then the IType based on rules 1 and 2 would be SB0.

3. In 1, if the datatype of x2 were changed to:

Type(x2):= Signed Byte with Radix 0 (SB0)

Then in the worst case, no 8 bit signed datatype could guard against overflow. In this situation, the IType would be a 16 bit signed number with radix 7 (denoted as SS7).

4. Now let us consider the earlier expression $y = -a + b$.

We need to determine an IType for the negation operation.

- a. First assume that:

Type(a):= Unsigned Byte with Radix 4 (UB4).

Then, since the negated value will be negative, the IType must be signed. Since, in the worst case, datatype UB4 may be as large as 15.9375 the appropriate IType is SB3.

- b. Now consider the case where:

Type(a):= Signed Byte with Radix 4 (SB4)

The only difference here is that a is now a signed quantity. Ignoring the possibility of a being equal to the extremal negative number accommodated by this datatype (-16), SB4 would accommodate all the other values that fit in

Type(a). Since the price of accommodating this last value is too much (losing one complete bit of information), an exception is made to IType selection rules above to keep SB4 as the IType here. Thus the negation of signed types, in general, is a minor exception to the first rule of ITypes.

The fixed-point enhancements to SystemBuild include a generalized version of the IType presented above. This generalization is implemented for all the datatypes and basic algebraic operations (negation, addition, subtraction, multiplication, and division, as well as all the supported blocks that require them). Note that, for 8 and 16 bit datatypes the IType rules guarantee that overflow would not take place (case 4b above is the only exception). For 32 bit types, this guarantee is not possible because no word promotion beyond 32 bits is provided.

NOTE: Consider a computation with a large number of additions, such as:

$$y = a1 + a2 + a3 + a4 + a5 + a6 + a7 + \dots \quad \text{Eq. 15-10}$$

To keep the nesting of intermediate datatypes from becoming excessive, we have placed a limit on the number of intermediate datatype computations that will be performed for any summation or multiplication block. The limit is placed at six; the seventh intermediate datatype is set to the result datatype and the cycle continues.

It should be noted that the user can avoid the use of intermediate type rules by restricting block diagrams to contain only well-defined binary or unary operations.

Simulation Issues

As you construct SystemBuild diagrams that perform fixed-point operations, keep certain issues in mind.

Fixed-point addition and multiplication (with or without intermediate types) forms an algebraic system that, although commutative, is not associative. For example, as illustrated in Figures 15-15 and 15-16 this means that:

$$(a + b) + c = c + (a + b), \text{ (commutative)} \quad \text{Eq. 15-11}$$

whereas

$$(a + b) + c \neq a + (b + c) \text{ (not associative)} \quad \text{Eq. 15-12}$$

Therefore, you must take care in forming block diagrams that perform such operations.

The non-associativity of the elementary algebraic operations implies that the computation order must be defined for these operations. For example, does a summation block that adds up three variables a , b , and c perform:

$$(a + b) + c \quad \text{EQ. 15-13}$$

$$a + (b + c) \quad \text{EQ. 15-14}$$

$$\text{or } (a + c) + b? \quad \text{EQ. 15-15}$$

To provide a consistent answer to this question, SystemBuild algebraic blocks are organized such that the operation order is the same as the order that the connection editor assigns to the input pins. Thus in the Summing Junction example the signal connected to input pin (1) is added to the signal connected to pin (2) first, and then the result is summed with the signal that is connected to pin (3) (see [Figure 15-16](#)). This can have a profound impact on the final operation result as it is possible to enhance the precision or even avoid dealing with overflow as in (1) under [Example 15-12](#) below,

EXAMPLE 15-12: Possible Implementations of the Expression $a+b+c$

[Figure 15-15](#) shows two of the three possible implementations of the expression $a+b+c$. This figure assumes that $\text{Type}(a) := \text{SB3}$, $\text{Type}(b) := \text{SB4}$, $\text{Type}(c) := \text{UB3}$, and that any summation block output datatype is SB3.

In this set-up, it is easy to see how the different implementations may produce different results: assume $a = -14.75$, $b = 6.75$, and $c = 14.75$. Then, using the summation rules for fixed-point datatypes, the implementation of $a+b+c$ as $(a+b)+c$ would yield:

$$a + b = -8, (a + b) + c = 6.75$$

Whereas the implementation of $a+b+c$ as $a+(b+c)$ results in:

$$b + c = 15.875, (b + c) + a = 1.125 \text{ (but SB3 saturates at 15.875)}$$

$$a + (b + c) = 1.125$$

1. Now assume that only one summation block is used to realize the sum. Then, the equivalent implementation to that of (1) would be as shown in [Figure 15-16 on page 15-31](#).

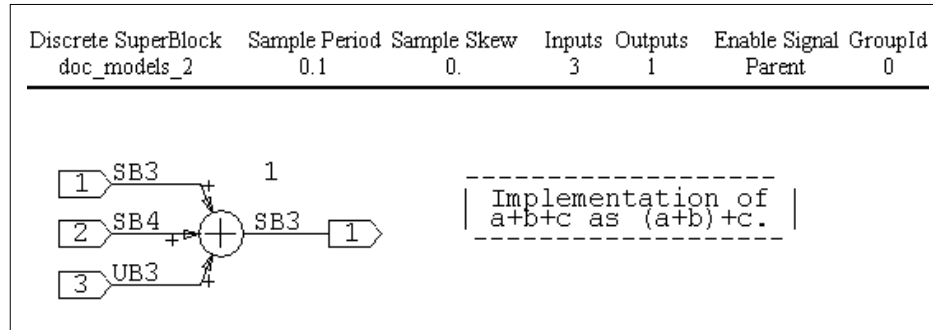


FIGURE 15-16 Adding Three Operands

Now we have:

$(a + b)$ has the IType SB2

$a + b = -8,$

$(a + b) + c = 6.75$

which shows no loss of precision.

2. The SystemBuild implementation of the fixed-point operations is based on the saturation arithmetic approach. This means that when results of operations overflow the limits of prescribed output datatypes, the result is a value that is clipped at the appropriate limit of that datatype.
3. 32 bit multiplication and division are dealt with differently in fixed-point arithmetic. This is explained in [32-bit Operation Issues](#), below.
4. Some of the blocks that are supported in fixed-point have parameters. For the block operation to be defined fully these parameters require types. In all blocks, with the exception of the Gain block, the parameter datatypes are derived from input and/or output types as explained in [Table 15-1 on page 15-23](#). The Gain block exception is explained in [Gain Block: A Special Case](#) on page 15-32.

32-bit Operation Issues

The operations of 32 bit multiplication and division are different from their 8 bit and 16 bit counterparts, because the maximum wordsize available is only 32 bits and therefore operands cannot be promoted to a higher wordsize before performing multiplication or division.

In 32-bit multiplication, if the sum of the radix positions of the operands is greater than that of the destination radix position, then the operands are shifted right or left so that when the two operands are multiplied they produce a result that conforms to the radix position of the destination (the radix position of the destination need not be the same as the radix position of the result). This is done in order to lessen the chance of overflowing during multiplication (i.e., shifting significant bits out of the register to the left), although it cannot always prevent overflow from occurring. Also, while shifting right, the operands can lose precision (i.e., shift significant out of the register to the right), and this can reduce the accuracy of results. If the sum of the radix positions of the operands is less than that of the destination, then the operands are multiplied and the result is aligned with the destination radix position. The result value gets clipped if overflow occurs when it is aligned with the destination. This does not always prevent overflow from occurring, because the result of the multiplication itself could overflow.

In 32-bit division, if the radix position of the dividend is greater than that of the divisor, the operands are divided and the result is aligned to the radix position of the destination. If the radix position of the divisor is greater than that of the dividend then the divisor is shifted right so that it gets aligned with the dividend. The result of the division is aligned to the radix position of the destination. Shifting right might result in zeroing of the divisor. If this happens, depending on the sign of the dividend, the extremal value that can be represented in 32 bits is returned. Without this adjustment, when shifting right, the divisor might lose precision and impact the accuracy of the result.

Gain Block: A Special Case

The Gain block is among the most commonly used, and most frequently parameterized, blocks in the Integrated Systems block library. Partly for this reason, for all its apparent simplicity, the Gain block represents a special case in fixed-point arithmetic. The exception regarding this block is that a user can optionally specify the Radix Position for the gain parameter, if the inputs and outputs of the block are fixed point. This feature is useful for defining datatypes with headroom for possible calibration using Run-time Variable editing. See [Example 15-13 on page 15-33](#).

The fixed-point datatypes are defined by three variables: signed or unsigned, length, and Radix Position. In setting the datatype of the Gain block, the variables are governed by the following considerations:

- Whether the gain parameter is signed or unsigned is defined by the sign of the parameter.
- The wordsize of the parameter is defined by the input datatype.
- By default, the Radix Position of the gain parameter is derived by “shrink wrapping” the user-supplied value of the gain parameter. This means that the Radix Position is chosen to give the minimum loss of precision (truncation of less significant bits) with no loss of significance (truncation of more significant bits).

15.3.2 Radix Calculations

If you specify an unsigned short as the output type of a Gain block, and place the number 3.1 inside it, the assigned radix will be = 14.

If you specify the Output datatype of the Gain block as fixed-point, the Parameters tab in the block dialog offers a **Define Radix** binary field with a default value of no. If you click to change this value to yes, a **Gain Radix** field appears, allowing you to specify a value for the Radix Position of the gain parameter. The only restriction on this value is that it must be consistent with the wordsize and Sign status of the gain parameter, which are the same as the wordsize and the sign of the parameter.

Define Radix Position used in conjunction with RVE helps you interactively establish an optimal fixed datatype for a given parameterized Gain block. The operation of the Define Radix Position feature is illustrated in Examples 15-13 below and 15-14 on page 15-36.

EXAMPLE 15-13: Building and Exploring Gain Blocks with Define Radix Position

In this example we build up a block diagram with %Variable Gain blocks that have different Gain Radix Positions, and compare the outputs of the blocks with gain values chosen to illustrate the effects of appropriate and inappropriate choices of Gain Radix Positions.

1. Create a new SuperBlock and name it test3. Make it discrete and accept the default Sampling Interval of 0.1. We will set the datatypes later. Start building the model shown in [Figure 15-17](#).

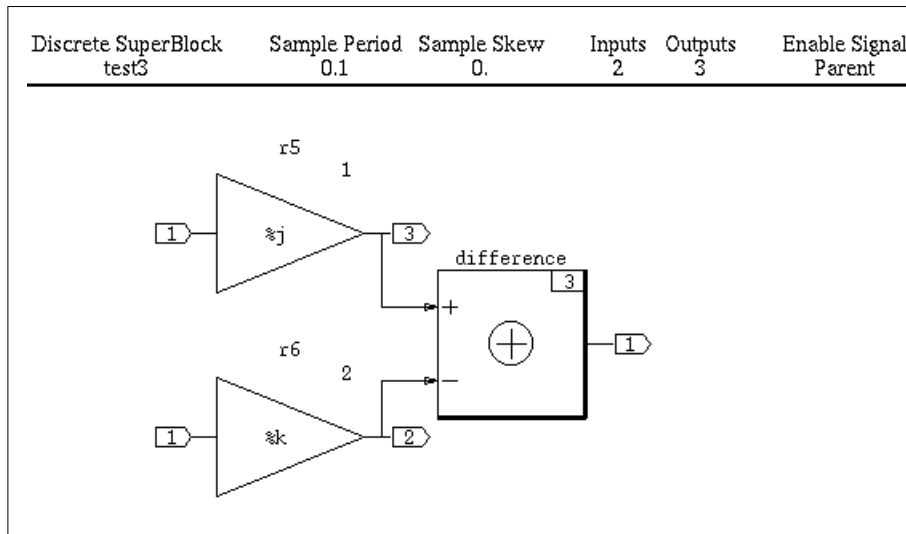


FIGURE 15-17 Gain Block Radix Example

2. Name the first block r5. Make the **Gain** 1.3, and the parameter j. Type ctrl-p to send the variable to Xmath. Click the **Outputs** tab and change the **Output Type** to Signed Byte and the **Output Radix** to 6. Return to the **Parameters** tab and observe that **Define Radix: No** appears. Click no to Yes. The **Gain Radix** field appears; make the gain radix 5. Click **OK**.
3. For the second block duplicate the steps for the first, except name the block r6 and specify a gain radix of 6. Although the value of the gain is also 1.3, we cannot use the same variable because the radix is different. Name the gain variable k, and type ctrl-p to save the value to Xmath.
4. Set the **SuperBlock Input Data Types**. Press the left mouse button on the SuperBlock ID bar, and hold it until the SuperBlock Attributes dialog appears. Click **Inputs** and change the **Input Type** to SignedByte. Press **Return** and change the Input Radix to 6. Click **Input Number** to change the 1 to 2, then change the **Input Type** to Signed-Byte, and change the Input Radix to 6. Make all data types must be the same.
5. Make the **Output Data Type** of the **Summing Junction** Signed Byte with an Output Radix of 6. Name the summing junction “difference”.

6. Prepare for simulation. In the Xmath command area, type:

```
t = [0:.1:10]';
u = [sin(t), sin(t)];
```

7. Simulate model with the following command:

```
sim("test3",{time=t,input=u,ialg="ALG",fixpt=1,fixpt_round=1,
minmax="minv",graph=1,typecheck=1,simtimer=1,initmode=0})
```

8. The plot should look like [Figure 15-18](#).

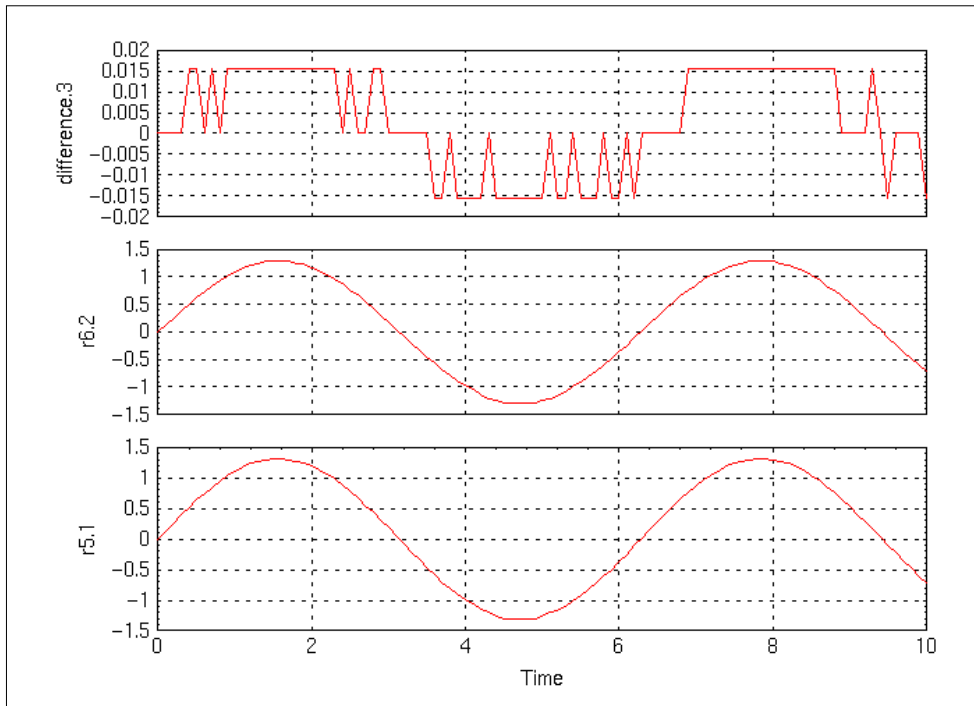


FIGURE 15-18 Plot Comparing Different Gain Radix 5 vs. 6

The spikes in the top strip of [Figure 15-18 on page 15-35](#) reflect the differences between the quantized sine waves in the two other strips. The distribution and heights of the spikes are caused by a combination of quantization factors:

- The sine waves were quantized on input (rounded), being converted to different datatype fixed-point numbers.
 - Both input sine waves were multiplied by 1.3, which was variously expressed as two fixed-point numbers of different datatype: 1.296875 (Radix 6), and 1.3125 (Radix 5).
 - The two sine waves were subtracted, with the result quantized to radix 6.
 - Finally, the difference was converted to floating-point for output and display.
-

EXAMPLE 15-14: Overflow Caused by Gain Values

For this example, we will use the same model, but use a different gain value that will force overflow on one of the channels of comparison.

1. Display the model from [Figure 15-17 on page 15-34](#).
2. Change the values of the gains:

```
j = 2.6;  
k = 2.6;
```

3. Run the simulation again with the t and u values from [step 6 on page 15-35](#) and the sim command from [step 7](#).
4. The plot should look like [Figure 15-19 on page 15-37](#).

Observe what is happening. The new gain value, which is a fixed-point number approximately 2.6 (actually 2.59375), when multiplied by the numbers from the u-vector (sine wave) at its extremal values, produces an overflow in the block whose datatype is SB6 and therefore whose range is [-2: 1.984375]. The other block, whose datatype is SB5, has a range close to [-4: +4], and therefore does not overflow at these values. You can see the effect of overflow clearly enough by looking at the bottom strip in [Figure 15-19](#), and you can see the approximate amount of the overflow in the top strip.

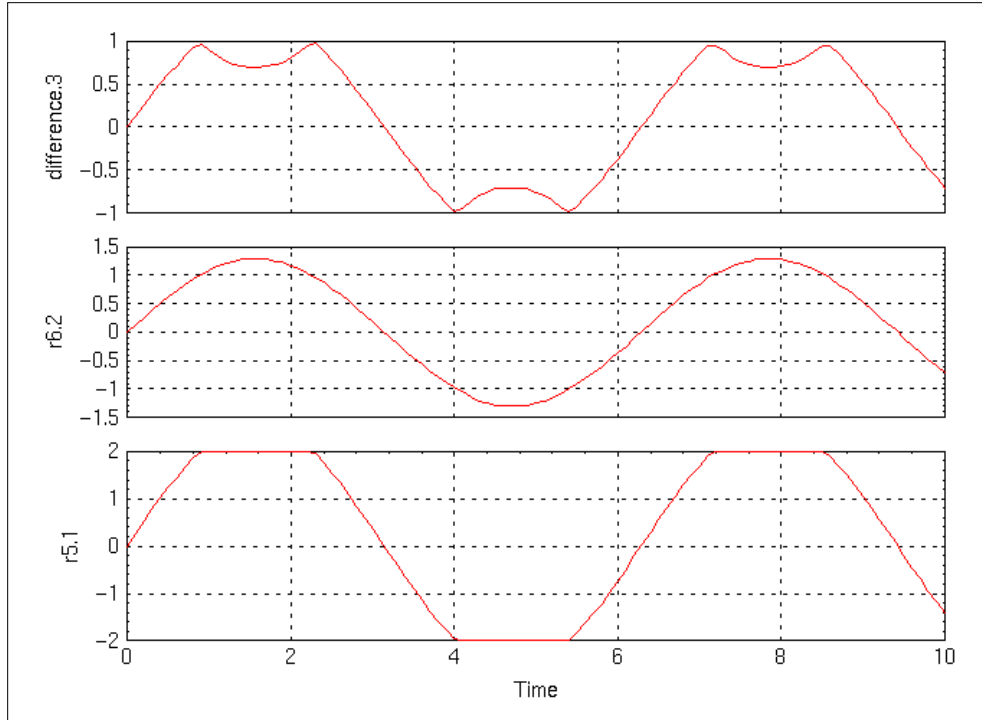


FIGURE 15-19 Plot of Gain Radix Positions with Overflow.

15.4 MinMax Data Logging

The MinMax data logging tool keeps track of the range of values output by each block. When the MinMax feature is on, the simulator stores the first occurrence of the maximum and minimum values for each block. During fixed-point simulations, the tool also records the first occurrence of an underflow or overflow in the block calculations, and the overflow protection status. After simulation, the results are transferred over to Xmath and stored in a special MinMax dataset. The name and partition location of the dataset are user-defined. The data in the list is available for direct manipulation, for example, it can be used for post-simulation processing and analysis.

Once the dataset is created it can be viewed with the `minmax_display` GUI interface.

Restrictions:

- No continuous SuperBlocks.
- Will not work with the `resume` keyword in simulation.

15.4.1 Activating MinMax Logging

In the simulation parameters dialog, enter a `variablename` (or `partition.variablename`) into the **MinMax Variable** field. When you click **OK**, the simulation occurs and the MinMax information is copied into an Xmath List Object (referred to as a Dataset), with your specified name.

Simulating with the minmax Keyword

the `sim` keyword `minmax` takes a string specifying the name of the Xmath dataset used for storing the MinMax data. For example,

```
y = sim("Top", t, u, {minmax = "test1"});
```

or, to include a partition

```
y = sim("Top", t, u, {minmax = "partitionname.test1"});
```

Note, MinMax display is not supported with standalone simulation.

Saving MinMax Datasets to a File

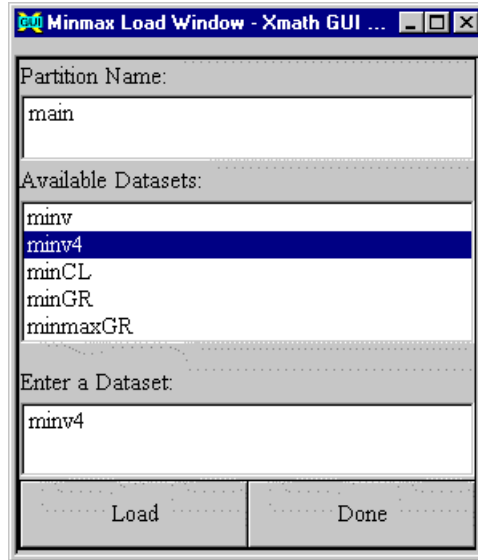
Datasets are stored as Xmath List variables. Like any other Xmath variable, they can be save to a file with the `save` command. If you are saving a catalog object from the Catalog Browser, be sure to include Xmath data if you want to preserve your dataset.

15.4.2 MinMax Display Tool

The MinMax Display tool allows you to display minimal and maximal values of signals from a simulation run. Timing and overflow information are also displayed. To launch MinMax, type the following in the Xmath command area.

```
minmax_display
```

To load a dataset for display, select Special→Load. The Load window displays the name of the current partition, and all datasets available within it. The partition can be changed. Select a dataset then press Load. Press Done when you are finished.



The MinMax Tool is shown in [Figure 15-20](#).

Output Name	Data Type	Max Value	Max Time	Min Value	Min Time
r5.1	SByte6	1.98438	0.9	-2	4.1

SuperBlocks	Blocks	Overflow	Underflow	Protection
1 test3	r5.1	0.9	4.1	No
	r6.2			No
	difference.3			No

FIGURE 15-20 MinMax Display, Simple Overflow ([Figure 15-19](#) Example Shown)

SuperBlocks that have MinMax information loaded and available to be displayed are listed in the **SuperBlocks** area. Child SuperBlocks are indented directly below their

parent SuperBlocks. Each block number is part of the name for unique identification, because two SuperBlock instances can have the same name.

The **Blocks** field displays the blocks in the selected SuperBlock. Two numbers (or blanks) may be displayed for each block: the time of the first overflow and underflow of the block. The Protection field indicates whether the Overflow Protection feature (on the Output Tab) was enabled at simulation time; No indicates it was disabled, and an empty column indicates it was enabled. The output information for the block you select will be displayed in the **Dataset** field.

For each output, the **Dataset** field displays the selected block's name, datatype, the initial minimum and maximum values, and the times they occurred.

Display Options

Choose Select→Output Display to change the display options. After making any changes, click **UPDATE** to see the new display. Display options stay in effect until they are changed.

If **Display SuperBlock** is checked, the window containing the selected SuperBlock will be raised whenever it is selected in the MinMax dialog.

15.5 User-Defined Data Types (Usertypes)

As a convenience in situations where multiple datatypes are required, you can assign your own meaningful names to datatypes, called User-defined datatypes or Usertypes. You can use these names in all the block datatype information dialogs. A special editor is provided to let you change the meaning of a custom datatype.

15.5.1 Usertype Editor

The Usertype editor provides an interactive interface for usertype creation, modification and deletion. This tool is launched from the Xmath command window. Before launching, make sure that the numerical display format is set to compact. To view the current format, type **SHOW FORMAT**. If the format is not compact, type: **SET FORMAT COMPACT**.

To launch the Usertype editor, type:

```
usertype
```

The Windows implementation of the Usertype editor is shown in [Figure 15-21](#); the UNIX version has the same fields, organized in the same manner, but its appearance is slightly different.

On the left side, a scrolled list displays all currently defined usertypes. The buttons on the lower right can update or delete the selected variable.

mytype voltage	DataType: FIXED	
	Un/Signed: Signed	Max Value: 7.99976
	Wordsize: Short	Min Value: -8
	Radix: 12	Resolution: 0.0002441406
	Name: mytype	Radix: 12
	Data Type:	
	<input type="radio"/> Logical	<input type="radio"/> Integer
	<input type="radio"/> Float	<input checked="" type="radio"/> Fixed
	Wordsize:	
	<input type="radio"/> Byte	<input checked="" type="radio"/> Short
<input type="radio"/> Long		
Signed/Unsigned:		
<input checked="" type="radio"/> Signed	<input type="radio"/> Unsigned	
Min/Res/Max		
-8 <- 0.000244140625 -> 7.99976		
Update	Delete	Quit

FIGURE 15-21 Usertype Editor

To create a new usertype — First type the usertype name in the **Name** field and press **Return**. Then choose the datatype for the new usertype. When you are satisfied with the usertype name and datatype, click the **Update** button. The new usertype appears in the scrolled list on the left side of the usertype editor window.

To modify a usertype — select the usertype name on the scrolled list with a single mouse click. You should see the name of the usertype filled in the **Name** field. Select the new datatype, and click **Update**.

To delete a usertype — select the usertype name on the scrolled list with a single mouse click. Click the **Delete** button at the lower left of the window.

15.5.2 Usertype MathScript Commands

A set of Xmath commands provide the same functionality as the Usertype editor. See the online help for more information on each command. Try typing the commands in [Example 15-15](#).

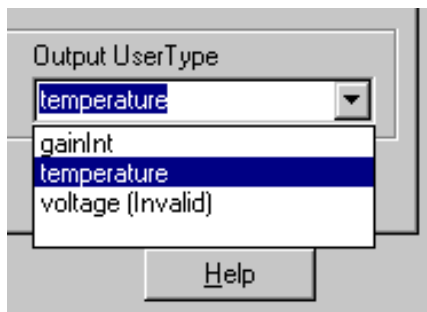
```
createusertype    Creates a new usertype definition.
modifyusertype   Changes the datatype of a usertype.
deleteusertype   Deletes a usertype.
listusertype     Lists out all defined usertypes in the Xmath Window.
```

EXAMPLE 15-15: Using Usertype MathScript Commands

```
createusertype "test1", {float}
createusertype "test2", {integer}
createusertype "test3", {wordsize = 16, radix = 4, signed = 1}
listusertype
modifyusertype "test2", {logical}
listusertype
modifyusertype "test2", {wordsize = 8, radix = 3}
listusertype
```

15.5.3 Using Usertypes in SystemBuild

Usertype information is located wherever datatype information resides. Typically you have an opportunity to enter an output usertype on a block dialog's Outputs tab, as shown below.



To enter a usertype, you can type its name in the field, or make a selection from the drop-down menu.

- Any usertypes available in the current catalog will be listed on the usertype menu. If a usertype is illegal in the current context it will be marked (Invalid), as shown above).
- Once a usertype is entered, other datatype items become read only.

EXAMPLE 15-16: Creating and Using a Usertype

1. Create a usertype named `test1`.
2. Create a Gain block model. Go to the output tab. Enter the usertype name `test1` in the field named Output UserType.
3. Click **DONE** to close the block.
4. Go the usertype editor. Change the datatype of `test1`.
5. Open the Gain block again. Inspect the output datatype. It should be updated to the new usertype definition.

If a usertype is deleted, but still referenced in a block, the last value of the usertype will be used.

15.5.4 Storing Usertypes

Usertypes are optionally stored in the SystemBuild model file.

The SystemBuild SaveAs and Load dialogs allow you to choose whether to save or load all usertypes, or none at all.

An Xmath keyword, `usertype`, is provided to let you control the loading and saving of usertypes. The `usertype` keyword indicates that only usertype data should be saved. Specifying `{!usertype}` indicates that only Xmath data and SystemBuild catalog data should be saved.

15.6 SystemBuild Functions in Fixed-point

15.6.1 Linearization Function

When a SystemBuild model is linearized in the fixed-point mode, the following steps are performed by the program:

1. The system parameters (i.e. parameters defined in the block forms of each block in the model) are quantized according to their fixed-point datatypes.
2. The operating point inputs and state (i.e., system initial conditions) are quantized according to the fixed-point rules.
3. A finite-difference linearization is performed by perturbing the states and inputs in the quantized model. The perturbation calculations are done in floating-point arithmetic.

By linearizing a model first in floating-point (i.e. floating-point parameters) and comparing the results to the linearization in fixed-point as above, one can observe the quantization effects on the model. Especially important is whether the system eigenvalues *after* the quantization are still stable (on or inside the unit circle for discrete models).

The linearization of a multirate discrete model in the fixed-point mode is also done in a similar way:

1. The model parameters and the operating point are quantized.
2. Model is simulated in floating-point with state and input perturbations.
3. Linearization is calculated from the simulation data.

For more details on how multirate linearization is done, see [Section 9.6 on page 9-7](#).

15.6.2 Simout Function

The `simout` function performs the following:

```
[x, xdot, y] = simout("model", {fixpt, other options})
```

The calculations for the output `y` are fixed-point computations. The states `x` are extracted from the model *after* they are quantized according to their datatypes. How-

ever, for the calculation of \dot{x} , the computation is done in floating-point. \dot{x} is the “pseudo-rate”, which is computed from

$$\frac{x[k + 1] - x(k)}{\Delta T} \quad \text{Eq. 15-16}$$

In this computation, $x[k+1]$ is calculated from $f(x[k], u[k])$ as a fixed-point calculation. On the other hand, ΔT is the sampling time, which is a floating-point number. The \dot{x} calculation is done in floating-point (that is, no roundoffs) to allow a user to extract $x[k+1]$ from it.

15.7 Scaling Aid Blocks

For your convenience in scaling your model, a special set of scaling aid icons has been added to the palette of ISIM icons. The scaling aid icons are found in the FI ISIM subpalette. See [Figure 15-22](#).

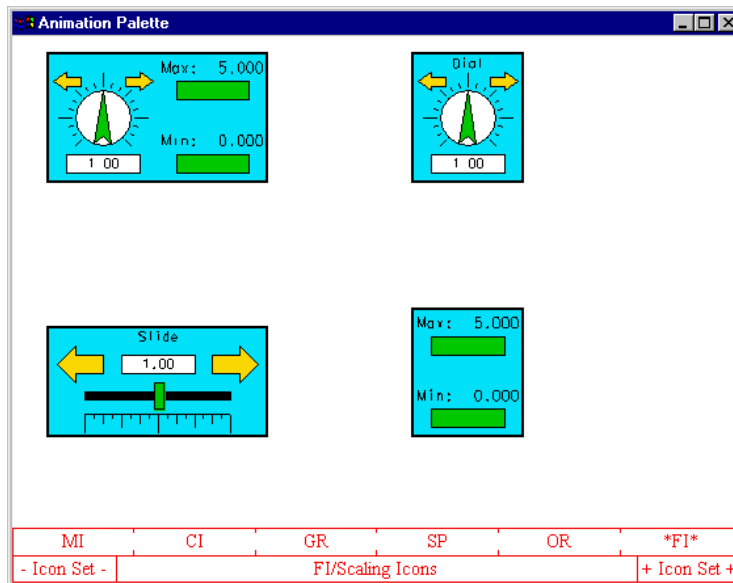


FIGURE 15-22 Scaling Aid Icons

The top left icon is a scaling aid. This icon uniquely allows you to generate a gain value that can be presented to your model *during the simulation run*, and to monitor an output from the model in the same time, recording its maximum and minimum values as you proceed.

16

Building Custom Icons

16.1 IA Basics

Interactive Animation (IA) is a graphics language for creating icons for display in the SuperBlock editor. For example, the icon displayed on a block in the editor (as opposed to blocks on the Palette Browser, which are bitmapped) are defined with the IA graphics language. Beyond this, IA icons have the ability of controlling inputs to and displaying outputs from a simulation while it is running.

- A simple use for the IA language is to create a static picture to display in the SystemBuild editor. Anyone with a SystemBuild license can use the IA language to create a custom block icon.
- The Interactive Animation module requires a separate license. It gives you the capability to create and compile custom icons using the IA Builder and provides additional tools to aid in creating icons and grouping them on new or existing palettes or combining them to produce control panels. Compiled IA source code takes the form of .sog files. A .sog file can be attached to a custom block via a reference on the Icon tab.

Note that you must have a license for the IA module to reference .sog files. For more information about IA, including use of the IA Compiler, see the *Interactive Animation User's Guide*.

All users can view IA palette(s) from the SystemBuild editor; click on the **IA** icon in the editor toolbar.

- If you do not have an IA license, a single palette of ISIM icons will be displayed.
- If you are licensed for the Interactive Animation module, clicking **IA** provides access to a set of five default palettes.

16.1.1 Adding a Custom Icon to a Block Diagram

There are several ways to add an icon to a block diagram:

All users can:

- Type icon source code into the Icon tab, then set the block icon type to **Custom** to see the new icon ([Section 16.2.2](#)).
- Reference or import an external bitmap ([Section 16.2.1](#)).

Users with an IA license can:

- Attach an icon to a block's custom view by specifying the name of a precompiled icon `.sog` file in the Icon tab. The Icon tab must contain the `ICON_SOGF:` command and the `ICON_NAME:` command, each pointing to an icon that exists on your system and is also specified in your `animation.cfg` file. ([Example 16-2 on page 16-6](#))
- "Drag and drop" an IA or ISIM icon from the Interactive Animation palette and place it, centered approximately, on the block icon to which it is to be attached ([Example 16-3 on page 16-7](#)).

16.1.2 Sample Icon Source

To view examples of icon source code, look at the source files that create the five main palettes delivered with IA. Each file contains all icon definitions for an existing IA palette. These files are typical of the resources found in `$SYSBLD/src`.

<code>moni.src</code>	Monitor Animation Icons
<code>coni.src</code>	Controller Animation Icons
<code>graf.src</code>	Graphic Shapes Icons
<code>spec.src</code>	Special Animation Icons
<code>orig.src</code>	Original Animation Icons

Precompiled `.sog` versions of these files are found in `$/SYSBLD/etc`. `Alarm.sog` becomes the sixth palette when alarm processing is turned on in the local `animation.cfg` file. Other files found in `$/SYSBLD/etc` are:

- `isim.sog`, which contains the limited icon set provided for SystemBuild ISIM. No `.src` file is furnished for this icon set because it is not designed to be user-modifiable.
- `control.sog`, which defines the IA Builder Control Panel. In `$/SYSBLD/src`, the file `control.src` is provided, to allow you to rearrange the control panel. Make copy of `control.src` and proceed with caution when rearranging the control panel.

16.2 Defining Custom SystemBuild Icons

You do not need to be licensed for IA to define your own icons. The Icon tab in the dialog box of any primitive block can be used to define your own icon design, or, reference or import an external bitmap.

16.2.1 Importing or Referencing an External Bitmap

You can assign a bitmap display to a block icon. The bitmaps may be BMP, XPM, GIF or JPEG format. Any size bitmap may be used. It's up to the user to size the block in a way that best displays the bitmap. A bitmap can be referenced as an external file or imported into the block diagram.

If you reference an external bitmap, the SystemBuild data file only stores the file path, and loads up the bitmap when needed. The Icon tab syntax is:

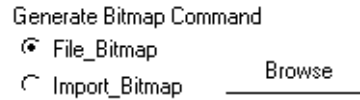
```
FILE_BITMAP 'path_to_file' [xloc yloc xsiz ysiz]
```

An imported bitmap becomes part of the diagram. To import an internal bitmap, the command is:

```
IMPORT_BITMAP 'path_to_file' [xloc yloc xsiz ysiz]
```

In both cases `[xloc yloc xsiz ysiz]` are specified in IA graphical units. An IA graphical unit is approximately the size of one pixel. `xloc` and `yloc` specify the bitmap location within the icon. `xsiz` and `ysiz` specify the dimension of the bitmap in graphical units. If the icon is zoomed or reduced in the editor, the size of a graphical unit scales accordingly.

Alternatively, you can specify an icon interactively. In the Icon tab, go to the Generate Bitmap command area.



Select either File or Import, then click the Browse button. Locate the desired file, then double-click on it. When the file is found, the proper IA source will be generated in the Custom Icon text field.

On the Display tab, change the Icon Type to Custom to view the new icon.

16.2.2 Creating or Attaching an IA Source Icon

When you click **OK** to release the dialog, the SystemBuild program compiles your IA Graphics language statements. If the icon code is syntactically correct, SystemBuild will display the image you have defined. This image will be displayed when the **Icon Type** is set to **Custom**. In the absence of any `BEGIN` or `END` section statements, your statements are assumed to be in the static graphics section by default. See [Section 16.3 on page 16-8](#) for the sections of the icon definition. Note that changing the icon display of the block has no effect on the block's operation or parameters. See [Example 16-1](#).

EXAMPLE 16-1: Making Your Own Custom Icon

In this example we describe a custom block, compile it, and display it in a block diagram model. Note, a custom icon is for display purposes only; it has no functionality.

1. Load the following file:

```
$SYSBLD/examples/auto/cruise_d.cat.
```

2. Open the SuperBlock continuous_automobile.
3. When the car model appears, select the 'engine' block (a Limited Integrator) and press **Return** to open the dialog; select the Icon tab.

- In the Icon Tab, type the following:

```
DRAW_RECT 100 100 800 800
SET_TEXT_FONT 14
DRAW_TEXT 500 500 22 '427 CID V8'
```

NOTE: By default these commands are in the static graphics section of the icon definition.

- Click **OK** to release the block and compile the icon. Place the cursor over the block and press the **S** key until the icon type changes to **Custom**. See [Figure 16-1](#) for the appearance of the new icon in the block diagram, along with the Icon tab contents.

When the icon appears, it may appear too small or may be hard to read. Grab the ID area in the upper right corner and drag in any direction necessary to remove the distortion.

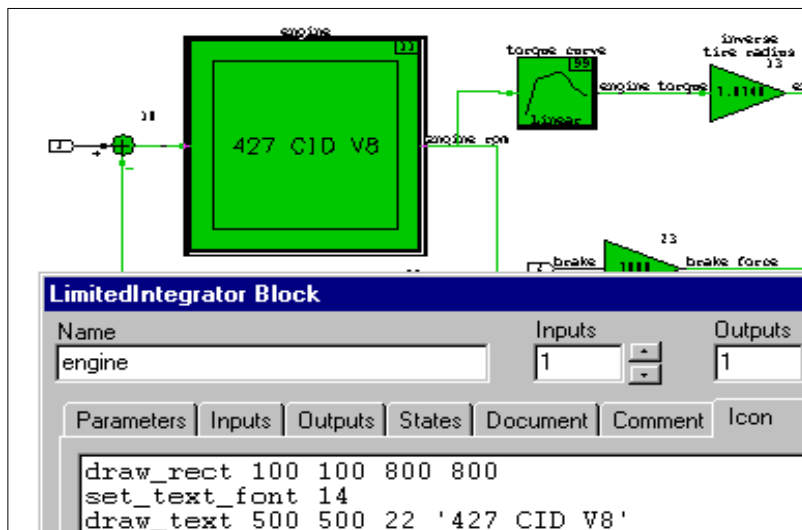


FIGURE 16-1 A User-Defined Icon and Its Description On the Icon Tab

Note that if you attach a custom icon to a block, the code defining the custom icon appears in the block dialog **Icon** tab from that time forward. Also, you can attach a custom icon to a block by placing the identification of the `.sog` file that holds the block definition and the name of the block into the **Icon** tab. See [Example 16-2](#) for attaching a custom icon.

EXAMPLE 16-2: Attaching a Precompiled .sog File Icon by Name and Filename

In this example we attach an ISI-supplied custom icon (strip chart) to a primitive block (Gain block) by placing its filename and icon name in the primitive block's Icon tab. When selected as a custom icon, the strip chart displays a history of the outputs of the Gain block.

1. In the continuous_automobile model, click to open the accelerometer block's dialog. This is the Gain block with ID 6.

2. In the **Icon** field, type:

```
ICON_SOGF: MI
ICON_NAME: SC1
```

3. Click **OK** to release the dialog. If the strip chart does not appear immediately, place the mouse cursor on the Gain block and press the **s** key a few times until the strip chart is seen. See [Figure 16-2](#) for a view of this icon and the dialog box with the .sog file and icon name.

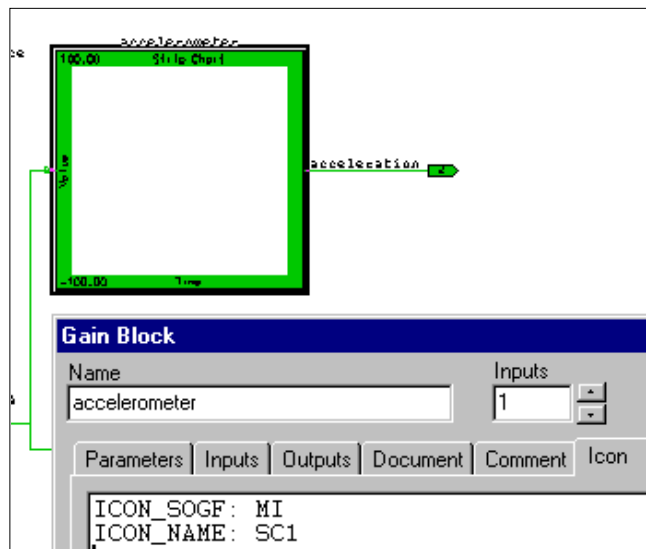


FIGURE 16-2 Strip Chart Icon Attached to a Primitive Block

A third method for attaching a custom icon to a primitive block is by drag-and-drop. [Example 16-3 on page 16-7](#) illustrates how this is done.

EXAMPLE 16-3: Attaching a Custom Icon Using Drag-and-Drop

This example assumes that you have access to the full IA palette, including the 1973 Corvette icon. If you do not have access to this palette, you can substitute any standard ISIM icon. Note, the custom icon has no functionality; it merely changes how the model looks.

1. With the continuous_automobile model displayed, click on the **IA** icon in the editor tool bar.
2. When the palettes become available, click **SP** to obtain the Special Icons palette.
3. When the Special Icons palettes appear, drag the automobile icon until it is just centered over the Car Inertia block. To perform the centering accurately, you may want to select the Car Inertia block first, so that you can see the exact block outlines. See [Figure 16-3](#) for this method.

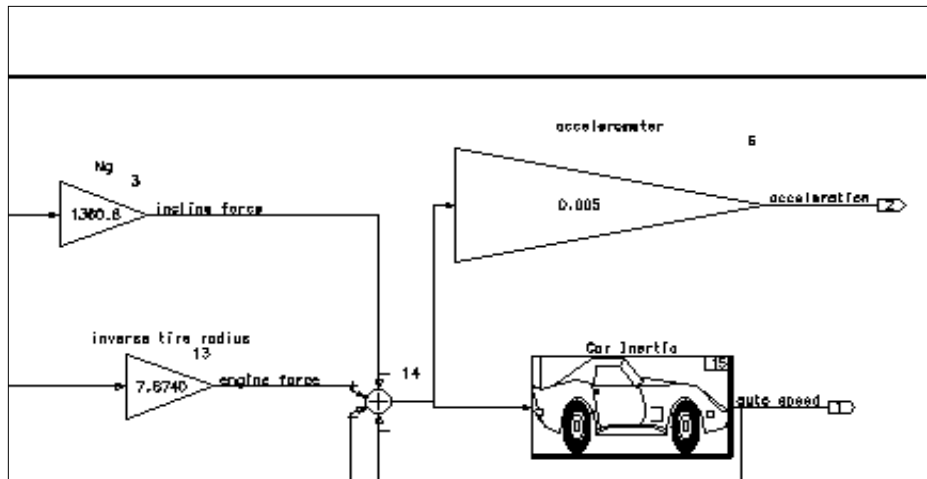


FIGURE 16-3 Attaching a Custom Icon by Drag-and-Drop

4. As can be seen from [Figure 16-3](#), the icon, once dropped in place, may appear distorted. Press and hold to grab the ID area in the upper right corner and pull in any direction necessary to correct the distortion.
5. Open the block dialog and click on the Icon tab observe the code produced by this action.

16.3 An Icon Source File

The format for a source file containing a single icon appears in [Example 16-4](#) below. Keywords are CAPITALIZED. Parameters follow keywords and are shown in lower case courier. For example, the keyword `DRAW_TEXT` has four parameters:

```
DRAW_TEXT x y mode strings
```

In source files, parameters are typically integers or reals; string parameters are enclosed in single quotes (' '). You must provide a value for a parameter, even if it's zero. Comments are shown in the normal text format.

The IDENTIFICATION shown in the third line of [Example 16-4](#) furnishes an abbreviation for the palette name ("MY" in the example), followed by a slash ("/") separator, followed by a name that is used to identify the palette in the icons data base ("MY animation icons" in the example).

The icon definition arguments (❶ through ❹ in [Example 16-4](#)) are discussed in [Section 16.3.1 on page 16-10](#). Sections are limited with `BEGIN` and `END`; you need not specify all sections. Section syntax and parameter explanations are discussed in [Section 16.3.6 on page 16-19](#).

EXAMPLE 16-4: Icon Source File Format

```
WS_DRAW ICON DEFINITION SOURCE FILE VERSION 4.00
C
IDENTIFICATION: 'MY/MY animation icons'

BEGIN_ICON
  ICON_TYPE:           number ❶
  ICON_NAME:           'Character string in single quotes'
  ICON_PRIVILEGE:     privilege ❷
  ICON_WIDTH:         width ❸
  ICON_HEIGHT:        height ❹
  INTEGER_VARIABLE:   number ❺ initial value ❻ 'Form Prompt'
  REAL_VARIABLE:      number   initial value 'Form Prompt'
  STRING_VARIABLE:    number   initial value 'Form Prompt'
  ANIMATION_POINTER:  number   'Form Prompt'
  OUTPUT_POINTER:     number   'Form Prompt'
  STATE_POINTER:      number   'Form Prompt'
```

```
BEGIN_INITIALIZATION_SECTION
C      Insert commands to be performed before static draw.
C      The following commands are allowed:
C      if/else/endif;
C      calculate;
C      sound_bell;
C      sound_key_click;
C      do/enddo;
C      math_function;
END_INITIALIZATION_SECTION

BEGIN_BACKGROUND_SECTION
END_BACKGROUND_SECTION

BEGIN_STATIC_GRAPHICS
C      Insert all static commands performed only at initial window
C      Insert non-graphic commands to be performed even if the
C      window is not displayed (the window must be loaded,
C      however). Typically this section is not used. Background
C      processing only works through the USRIA1 interface (USRIA1
C      UCB and RTMPG,not in ISIM.)
END_STATIC_GRAPHICS

BEGIN_ANIMATION_GRAPHICS
C      Insert commands updated at each data cycle.
END_ANIMATION_GRAPHICS

BEGIN_POINTER_ACTION
C      Insert commands performed when the user clicks on the icon.
END_POINTER_ACTION

BEGIN_FORM_DEFINITION
C      Form definition for the icon.
END_FORM_DEFINITION
END_ICON

BEGIN_PALETTE_DEFINITION
C      The palette definition occurs once
C      at the end of the source file.
END_PALETTE_DEFINITION
```

16.3.1 Icon Identification

This section lists the possible parameters for each keyword in the identification portion, which immediately follows the `BEGIN_ICON` keyword (see ❶ on [page 16-8](#)). When defining your own icon, *do not use tabs to indent any portion of your code. Use spaces only.*

- ❶ `Icon_Type` is the icon identification number for each icon. This number will be used in the `PALETTE_DEFINITION` section. If you have changed the `IDENTIFICATION` field ([page 16-8](#)), and thus changed the name of the palette, you may use sequential integers (1...n) for your icons.
- ❷ `ICON_PRIVILEGE` is normally set to 0.
- ❸ `ICON_WIDTH` gives the icon box width in IA graphical units (100 ~ 1 cm). Normally an IA graphical unit is approximately one pixel, however, if the icon is zoomed or reduced the size of a graphical unit varies accordingly.
- ❹ `ICON_HEIGHT` gives the icon box height in pixels (100 ~ 1 cm).
- ❺ A number you assign the declared variable. You refer to the variable using the format `Type[#]` where `Type` is usually a unique character specifying the type of integer, real, or strings. Types are discussed in [Section 16.3.2](#).
- ❻ Initial value can be '##' where ## is the number of characters.

16.3.2 Types

The following types are possible for integers, reals, and strings. Any variable can be replaced by an expression that reduces correctly to the appropriate type of variable.

Integer Types

Hardcoded Integer

I[#] Integer Variable

Real Types

Hardcoded Real

R[#] Real Variable

A[#] Animation Pointer (to input vector)

O[#] Output Pointer (to output vector)

T[#] Time & Hold Information (pointer to t-vector)

String Types

Hardcoded String

S[#] String Variable

Typically variables are declared at the top of the icon source file (see [Example 16-4 on page 16-8](#)); each separate type is numbered sequentially starting with 1. For example, to define a string variable numbered 17, specify S[17].

16.3.3 General Control and Calculation Statements

The following keywords and their parameters can be used in the ANIMATION_GRAPHICS and STATIC_GRAPHICS sections. Note the use of the RETURN commands in a few places in these examples. You may use RETURN at appropriate places in your code to force a return; executing the last keyword of a section has the same effect.

TABLE 16-1 Control and Calculation Keywords

Syntax	Example
CALCULATE v1ptr = fun v2ptr v3ptr CALCULATE v1ptr = v2ptr	CALCULATE I[3] = I[4] + I[5] CALCULATE I[3] = I[4]
MATH_FUNCTION v1ptr = fun v2ptr v3ptr	MATH_FUNCTION R[1] = ATAN2 [R[2] R[3]
IF value relation value THEN ELSE ENDIF	IF I[3] < 10 THEN your_statements ELSE your_statements ENDIF
IF value relation value THEN RETURN ENDIF	IF I[2] EQ 1[5] THEN RETURN ENDIF
DO variable start end inc RETURN ENDDO	DO I[2] 1 5 1 Your_Statement ENDDO

CALCULATE functions use the pointers v1ptr, v2ptr, and v3ptr. Pointers can be reals, integers, or strings. CALCULATE functions can be used in any section of a program.

+	Add
-	Subtract
*	Multiply
/	Divide
AND	Logical Operation
OR	Logical Operation
MAX	Maximum of the two args (strings OK)
MIN	Minimum of the two args (strings OK)

MATH_FUNCTION v1ptr = function v2ptr v3ptr

The ATAN2 function uses v3ptr.

MATH_FUNCTIONS are:

SIN	COS	TAN	SQRT	ABS
ASIN	ACOS	ATAN	ATAN2	

```
IF value relation value THEN
  ELSE
ENDIF
```

IF/THEN values can be real or integers; strings are also possible. Possible relations are:

EQ or =	GE or >=	GT or >
NE or <>	LE or <=	LT or <

```
DO variable start end inc
  RETURN
ENDDO
```

variable is an integer representing the counting variable. start and end are the beginning and ending values in the loop, and inc is the counter increment.

16.3.4 General Graphic Statements and Coordinate System

These keywords and their parameters can be used in the ANIMATION_GRAPHICS and STATIC_GRAPHICS sections. Properties of things drawn with general graphic statements are determined by the general graphic characteristics settings (see [Section 16.3.5](#)). Objects are located via a coordinate system that defines 0,0 as the lower left corner of the screen, and entities within an icon are located using a similar coordinate system that defines 0,0 as the lower left corner of the icon. See Examples [16-5](#) through [16-16](#).

DRAW_RECT	x1 y1 width height
DRAW_TEXT	x y justify strings
ERASE_TEXT	x y justify strings
GET_TEXT_SIZE	string R[width] R[height] Returns the width and height of the string in IA graphical units.
DISPLAY_VALUE	iptr x y width dplaces justify
ERASE_VALUE	x y width dplaces justify
DRAW_ARC	xc yc xr yr start end
DRAW_LINE	npoints x1 y1 x2 y2 ... xn yn
ROTATE_LINE	ptr xc yc npoints x1 y1 x2 y2...xn yn
RELATIVE_POSITION_LINE	xd yd npoints x1 y1 x2 y2 ...
GENERAL_LINE	npoints x1 y1 x2 y2 ... xn yn
VARIABLE_SIZE_BOX	dir iptr x1 y1 v1p x2 y2 v2p
VARIABLE_POSITION_LINE	dir iptr x1 y1 v1p x2 y2 v2p

EXAMPLE 16-5: Graphics Draw Rectangle Statement

```
DRAW_RECT x1 y1 width height
```

Draws a rectangle where `x1` and `y1` designate the lower left rectangle corner, and `width` and `height` are the rectangle dimensions, all expressed in terms of the icon coordinate system. By default, the icon dimensions are 1000 × 1000; these values can be reset using the `ICON_WIDTH` and `ICON_HEIGHT` keywords.

EXAMPLE 16-6: Graphics Draw Text Statement

```
DRAW_TEXT x y justify string
```

Draws the text `string` starting at the point specified with `x` and `y`. (The font is determined by `SET_TEXT_FONT`, described on page 478.n.) The text is aligned on that point according to the `justify` parameter. This parameter is a 2 digit value where the tens column aligns horizontally and the ones column vertically.

justify digit	horizontal (tens)	vertical (ones)
1	left	top
2	center	center
3	right	bottom

For example, if `justify` is 21 the string will be drawn centered on the starting point and top-aligned, because a 2 was in the tens column and a 1 was in the ones column.

EXAMPLE 16-7: Graphics Erase Text Statement

```
ERASE_TEXT x y justify strings
```

Erases a string created by `DRAW_TEXT` when given exactly the same parameters.

EXAMPLE 16-8: Graphics Display Statement

```
DISPLAY_VALUE iptr x y width dplaces justify
```

Displays a value `iptr` at the point specified by `x` and `y`. Placement is determined by the parameters `width` (the number of characters), `dplaces` (decimal places) and `justify` (you supply the same 2-digit code value used for `DRAW_TEXT justify`).

EXAMPLE 16-9: Graphics Erase Value Statement

```
ERASE_VALUE x y width dplaces justify
```

Erases a value created by DISPLAY_VALUE when given exactly the same parameters.

EXAMPLE 16-10: Graphics Draw Arc Statement

```
DRAW_ARC xc yc xr yr start end
```

Draws an arc centered on the point specified by `xc` and `yc`. `xr` and `yr` describe the radius of the arc and `start` and `end` give the angle in degrees at the start and end of the arc. The keyword `SET_ARC_TYPE` (see page 478.n) determines the fill pattern for the arc.

EXAMPLE 16-11: Graphics Draw Line Statement

```
DRAW_LINE npoints x1 y1 x2 y2 ... xn yn
```

Draws a fixed line based on coordinate pairs of `x,y` values, where `npoints` is the number of break points in the line and an `x,y` coordinate pair is specified for each point in the line. During simulation, if you wish to be able to change the angle, displacement, or scale of the line, use `GENERAL_LINE` instead.

EXAMPLE 16-12: Graphics Rotate Line Statement

```
ROTATE_LINE ptr xc yc npoints x1 y1 ... xn yn
```

Does what `DRAW_LINE` does but adds the ability to rotate the line centered on the point specified by `xc` and `yc`. `ptr` is the number of degrees of rotation counterclockwise from the horizon (i.e., the positive part of the x-axis).

EXAMPLE 16-13: Graphics Relative Position Line Statement

```
RELATIVE_POSITION_LINE xd yd npoints x1 y1 x2 y2 ...
```

Does what `DRAW_LINE` does but adds the ability to place the line relative to the point specified by `xd` and `yd` (the coordinate displacement in IA graphical units).

EXAMPLE 16-14: Graphics General Line Statement

```
GENERAL_LINE npoints x1 y1 x2 y2 ... xn yn
```

Draws a line based on coordinate pairs of `x,y` values, where `npoints` is the number of segments in the line and an `x,y` coordinate pair is specified for each point in the line. A general line can be manipulated with the `SET_ANGLE`, `SET_SCALE` and `SET_DISPLACEMENT` settings in [Section 16.3.5](#).

EXAMPLE 16-15: Graphics Box statement

```
VARIABLE_SIZE_BOX dir iptr x1 y1 v1p x2 y2 v2p
```

Defines a rectangular area whose lower left corner is specified by `x1,y1` and whose upper right corner is specified by `x2,y2`, then fills a portion of that area, creating a box. `iptr` points to a value you have calculated, indicating the percentage of the box defined by the `x` and `y` values. The box is created by filling a portion of the area delimited according to `v1p`, the minimum range, and `v2p`, the maximum range; these parameters are usually values you have calculated elsewhere. `dir` can be either `'HOR'` or `'VER'`, specifying that the filling will proceed horizontally or vertically from the lower left corner of the area until the portion of the area specified with `v1p`, `v2p` is filled.

EXAMPLE 16-16: Graphics Variable Position Line Statement

```
VARIABLE_POSITION_LINE dir iptr x1 y1 v1p x2 y2 v2p
```

Does what `VARIABLE_SIZE_BOX` does, only draws a line in the defined area (rather than filling a box).

16.3.5 General Graphic Characteristic Statements

These keywords and their parameters can be used in the ANIMATION_GRAPHICS and STATIC_GRAPHICS sections.

```

SET_COLOR          color
SET_LINE_TYPE      type
SET_LINE_WIDTH     width
SET_FILL_PATTERN   pattern
SOUND_BELL         loudness
SOUND_KEY_CLICK    loudness
SET_ARC_TYPE       type
SET_TEXT_FONT      font type
SET_TEXT_SLOPE     angle
SET_LINE_DISPLACEMENT xd yd
SET_LINE_ANGLE     angle x y
SET_LINE_SCALE     xsc ysc  xs ys

```

SET_COLOR color: Sets color to a number from the following list:

0=white	4=cyan	8=orange	12=lt blue
1=black	5=magenta	9=pink	13=purple
2=red	6=yellow	10=yellow-green	14=brown
3=green	7=blue	11=blue-green	15=gray

SET_LINE_TYPE type: Sets line type to a number from the following list:

1 = solid line	5 = dash-dot	8 = dashed
2 = dotted line	6 = wide-spaced dash	9 = dashed
3 = dashed line	7 = close dots	10 = dotted
4 = close dash		

SET_LINE_WIDTH width

Specifies line width in pixels; the width of a normal line is 1 pixel.

<code>SET_FILL_PATTERN pattern</code>	Specifies the pixel density of a filled area. If 0 is supplied, there is no fill. If 2 is supplied the fill will be solid. Density is also indicated with numbers between 48 and 62, which indicate the number of pixels (ranging from 1 to 15) in a 4 × 4 array. For example, if you specify <code>SET_FILL_PATTERN 55</code> , expect a density of 8 pixels.
<code>SOUND_BELL loudness</code>	Sets bell volume to an integer between 0 and 8, where 0 is silent and 8 is the loudest.
<code>SOUND_KEY_CLICK loudness</code>	Sets key click volume to a number between 0 and 8 where 0 is silent and 8 is the loudest
<code>SET_ARC_TYPE type</code>	Specifies fill type as 0, 1, or 2. 0 indicates empty, 1 indicates a filled arc, and 2 indicates pie fill.
<code>SET_TEXT_FONT font type</code>	Sets the font type with an integer between 1 and 14. To see what these fonts look like, bring up the IA palette GR/GRaphic shapes. The appearance of these fonts may vary among platforms.
<code>SET_TEXT_SLOPE angle</code>	Sets the text slope with a number indicating the angle of the text in degrees counterclockwise from horizontal. The angle is a number representing the degrees of rotation.
<code>SET_LINE_DISPLACEMENT xd yd</code>	Moves a line drawn by <code>GENERAL_LINE</code> . This keyword displaces all points in the x and y directions by the number of IA graphical Units in xd and yd (default=0).
<code>SET_LINE_ANGLE angle x y</code>	Specifies the angle of a line drawn with <code>GENERAL_LINE</code> . The parameters x and y give the x,y coordinates of the center point of the rotation (which need not be the center of the line). The angle is a number representing the degrees of rotation.
<code>SET_LINE_SCALE xsc ysc xs ys</code>	Expands or shrinks a <code>GENERAL_LINE</code> and, if desired, changes its distance from the scale point at the same time. xs and ys are reals such that 1 is full size, .5 is half size, etc. xsc and ysc indicate the location of the scale center point; the default is 0 (centered).

16.3.6 Animation Statements

The following statements can be used in the ANIMATION GRAPHICS and POINTER_ACTION sections.

```
ABSOLUTE_ICON_POSITION  xptr  yptr
```

Places the bottom left corner of an icon at the screen coordinates specified in IA graphical units.

```
MOVE_ICON  xdptr  ydptr
```

Moves/displaces an icon from its current location in the x and y direction by the specified number of pixels. If both ABSOLUTE_ICON_POSITION and MOVE_ICON are specified, the moves take place in the order in which the statements are executed.

```
MOVE_AREA  x1  y1  x2  y2  xd  yd
```

Defines a rectangular area whose lower left corner is found at x1, y1 and whose upper right corner is specified as x2, y2 then moves the area so that its lower left corner is at the point specified by xd, yd.

```
COMPLIMENT_AREA  x1  y1  x2  y2  mode
```

Defines a rectangular area whose lower left corner is at x1, y1, and whose upper right corner is x2, y2. If mode is 0 the area will be made a complementary color; if it is 1, the area will flash in the current color.

```
ERASE_AREA  x1  y1  x2  y2
```

Erases a rectangular area whose lower left corner is at x1, y1, and whose upper right corner is at x2,y2.

```
REQUEST_POSITION  mode  xpos  ypos  tclick/status  button
```

This keyword allows you to get the position of the pointer and find out what the mouse is doing.

mode is 0 (previous) when this keyword is used for pointer action, and 1 (current) when used for animation graphics. xpos and ypos store the location of the pointer.

If mode is 0, tclick stores a value for mouse button action. tclick is 0 if the button is depressed, 1 if single-clicked and 2 if double-clicked.

If mode is 1, button status is 0 (OK) or 1 (out of window).

If mode is 0, button is a variable identifying the button used. 1 is the left button, 2 the middle button, and 3 the right button. If no button is pressed the variable is 0.

16.3.7 Pointer Action Statements

The `HOT_SPOTS` statement is used to define a rectangular area in an icon that has an associated line or block of code that is to be executed when the hot spot area is clicked with the mouse.

```
HOT_SPOTS    ptr  npts  H1-x1    H1-y1    H1-x2    H1-y2
              H2-x1    H2-y1    H2-x2    H2-y2
              Hn-x1    Hn-y1    Hn-x2    Hn-y2

INQUIRE     ptr  prompt

LOAD        string

CHAIN_DOWN  string

CHAIN_UP
```

where:

```
HOT_SPOTS ptr npts H1-x1 H1-y1 H1-x2 H1-y2
              H2-x1 H2-y1 H2-x2 H2-y2
              Hn-x1 Hn-y1 Hn-x2 Hn-y2
```

`ptr` is a value (calculated elsewhere) that tells which hot spot is being pointed at. `npts` is an integer specifying how many hot spots there are on the icon. Each icon hot spot is a rectangular area specified by bottom left corner (`H1-x1`, `H1-y1`) and top right corner (`H1-x2`, `H1-y2`).

<code>INQUIRE ptr prompt</code>	Creates a dialog box containing a prompt string you specify; <code>ptr</code> accesses what the user types into the dialog.
<code>LOAD string</code>	Loads a <code>.pic</code> file without bothering to keep track of previous <code>.pic</code> files
<code>CHAIN_DOWN string</code>	Loads a <code>.pic</code> file, keeping track of its position in a hierarchical stack.
<code>CHAIN_UP</code>	Reloads the last <code>.pic</code> file.

16.3.8 Palette Definition

The palette definition for all icons in a given file is placed at the end of the file. For examples, look at the end of any of the IA .src files in \$SYSBLD/src.

```
BEGIN_PALETTE_DEFINITION
```

```
WINDOW_SIZE:      width height
```

```
PALETTE_OBJECT:  type xpos ypos page vars_diff var_value_pairs
```

```
END_PALETTE_DEFINITION:where
```

width width of window in pixels

height height of window in pixels

type icon type number (❶ on [page 16-8](#))

xpos x position of icon on window

ypos y position of icon on window

page Page number of palette. This parameter must be sequential. Numbers can range from 1 to 20 (practical limit) in order to define pages in the palette.

#vars_diff These parameters allow you to have different forms of the same icon on the palette without defining them all separately by allowing you to create a new icon that is the same as an existing icon except for changes in its default settings.

var value pairs

If a non-zero entry is specified for vars_diff, the compiler looks for that number of value pairs to follow.

For example, the following palette object description,

```
PALETTE_OBJECT: 3 900 295 5 2 I[2] 4 R[1] 3
```

creates a new icon based on icon number 3 by changing two default variables. Integer variable 2 is given a value of 4 and real variable 1 a value of 3.

16.4 animation.cfg

You can modify the supplied configuration file \$SYSBLD/etc/animation.cfg (shown in [Example 16-17 on page 16-22](#)) so that IA uses your custom icons. Do not change anything in the starred (**) banner area; these definitions are defaults. Rather, copy definitions from the default area and place them in the user definition area below the line reading "STATE YOUR DEFINITIONS BELOW", remove the asterisks, and make your changes to the copied material. User definitions, such as the

typical ones shown below, supersede the defaults, so there is no need to remove or change the defaults.

Definitions in the default area that contain “==>” are *not* activated, and may be used as comments. For example, copy the line:

```
* BUILD_LOAD_PICTURE ==> 'pict1.pic'
```

to the user definition area and remove “==>” and the asterisk, and add a colon as shown below, with the spacing exactly as shown:

```
BUILD_LOAD_PICTURE: 'pict1.pic'
```

EXAMPLE 16-17: Typical Animation .cfg File

```
WS_DRAW CONFIGURATION FILE (ANIMATION.CFG) VERSION 6.0

*****
* The first line of the config file must be:                               *
*     WS_DRAW CONFIGURATION FILE (ANIMATION.CFG) VERSION X                 *
* All meaningful lines must start with the key words and COLON,           *
* followed by the name of the file in single quotes.                     *
* By convention, files are lower case, keywords and others                *
* are upper case. Blank and comment lines are allowed.                   *
*                                                                           *
* THESE KEYWORDS MAY HAVE MORE THAN ONE FILE POSSIBLE:                   *
*                                                                           *
*     ICON_DATA_FILE      ==> 'project1.sog'                               *
*     ICON_DATA_FILE      ==> 'project2.sog'                               *
*     ICON_DATA_FILE      ==> 'project3.sog'                               *
*                                                                           *
*     BUILD_LOAD_PICTURE  ==> 'pict1.pic'                                  *
*     BUILD_LOAD_PICTURE  ==> 'pict2.pic'                                  *
*     BUILD_LOAD_PICTURE  ==> 'pict3.pic'                                  *
*                                                                           *
*     PROCESS_PICTURES    ==> 'proc1.pic'                                  *
*     PROCESS_PICTURES    ==> 'proc2.pic'                                  *
*     PROCESS_PICTURES    ==> 'proc3.pic'                                  *
*                                                                           *
* THESE KEYWORDS INCLUDE THE FOLLOWING DEFAULTS:                          *
*                                                                           *
*     ICON_DATA_FILE:      'etc:isim.sog'                                  *
*     ICON_DATA_FILE:      'etc:moni.sog'                                  *
*     ICON_DATA_FILE:      'etc:coni.sog'                                  *
*     ICON_DATA_FILE:      'etc:graf.sog'                                  *
*     ICON_DATA_FILE:      'etc:spec.sog'                                  *
*     ICON_DATA_FILE:      'etc:orig.sog'                                  *
```

```

*      ICON_DATA_FILE:          'etc:alarm.sog'          *
*
*      ICON_SOURCE_FILE:       'myicon.src'             *
*
*      BUILD_CONTROL_PANEL:    'etc:control.sog'        *
*
*      SAVE_FILE_FORMAT:       'ASCII' or 'BINARY' or 'MATRIX' *
*
*      PICTURE_SCALE_FACTOR:    '1.0' (+ relative, - absolute) *
*
*
* THE FOLLOWING COMMANDS SHOULD ONLY BE USED FOR HARDWARE:
*
*      I/O_PROCESSING           ==> 'I/O PROCESSING ON'   *
*
*      CODE_GENERATION_OUTPUT_FILE ==> 'pict.ada'         *
*
*      ADA_LIBRARY              ==> 'pict.alb'           *
*
*      FREQUENCY_SCALE_FACTOR   ==> '1.0'               *
*
*      HARDWARE_CONNECTION_EDITOR_FILE ==> 'pict.ioc'     *
*
*****
*
*              STATE YOUR DEFINITIONS BELOW
*
*****
ICON_SOURCE_FILE:          'myicon.src'

ICON_DATA_FILE:           'myicon.sog'

BUILD_LOAD_PICTURE:       'pict.pic'

SYSTEM_BUILD_RTF_FILE:    'pict.rtf'

SIMULATION_DATA_FILE:     'pict.sim'

SAVE_FILE_FORMAT:         'ASCII'

ALARM_PROCESSING:         'ALARM PROCESSING OFF'

ALARM_WINDOW_PICTURE:     'alarm.pic'

```


Note that the `ICON_SOURCE_FILE` and the `ICON_DATA_FILE` lines are the only change from the Version 4.1 distribution version of the file.

16.4.1 Important animation.cfg Keywords for Customized Icons

This section describes the `animation.cfg` keywords you might need to alter.

<code>ICON_DATA_FILE:</code> 'myicons.sog'	You supply the compiled and translated source file that defines which icons are available in IA.
<code>PROCESS_PICTURES:</code>	List all the '*.pic' files you want to access while running IA. at run time. Use the Process icon to access '*.pic' files listed here.
<code>ICON_SOURCE_FILE:</code> 'myicons.src'	Supply the name of the source file which contains your icon definitions. When you have finished creating and debugging your icons, create a new .src and .sog file so that you can use your new icons.
<code>BUILD_LOAD_PICTURE:</code>	Allows you to give the default (first) file you want supplied (displayed) when 'SAVE PICT' or 'LOAD PICT' is chosen from the IA Builder control panel. Making your file the default will save time if this is the primary icon you use.
<code>SYSTEM_BUILD_RTF_FILE:</code>	Give the default file you want supplied when you choose 'RTF NAMES' from the animation control panel. The '*.rtf' file is the run-time file your SystemBuild file generates. It must be loaded into IA before connections can be made. Entering the name of your '*.rtf' here can save time.

16.4.2 icon.src Field for Customized Icons and New Palettes

`IDENTIFICATION:` 'u p to 30 Characters ' Each time you start a new palette, change this field to a new identifier, but keep it under 31 characters. If you don't change the name you may encounter errors which are the result of conflicting `ICON_TYPE` numbers (see  in [Section 16.3.1 on page 16-10](#)).

16.5 Procedure for Building Your Own IA Custom Icons

Figure 16-4 illustrates the steps for building custom IA icons. This process is summarized below; see the *Interactive Animation User's Guide* for a detailed explanation.

1. Use a text editor to edit a copy of one of the icon `.src` files that come with IA. Change the `IDENTIFICATION:` field to something unique and save the file with a different name; e.g., `infile.src`. the Identification is the name of the new palette whose icons' source statements are contained in `infile.src`.

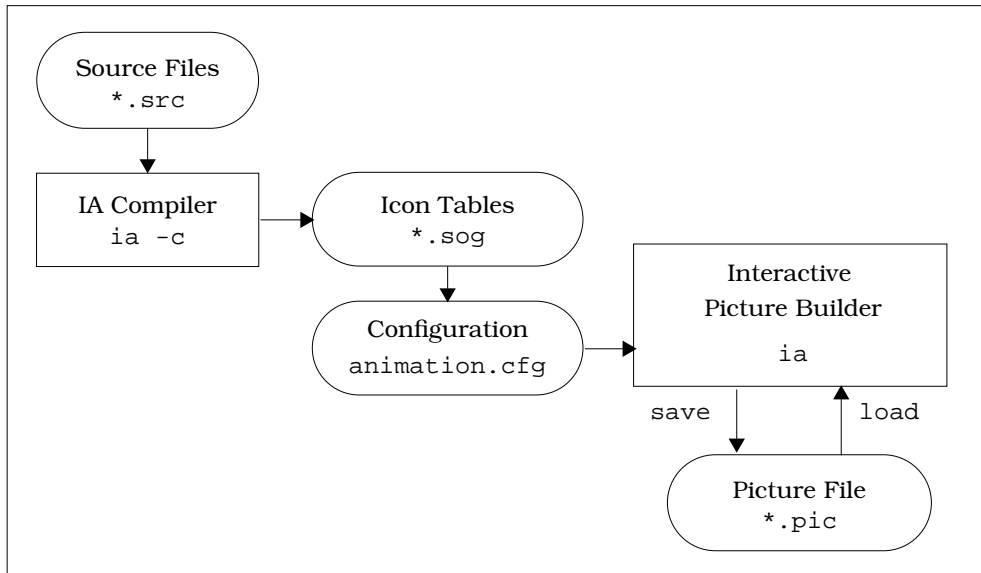


FIGURE 16-4 Building Custom IA Icons

2. Using your version of the `.src` file as a template you may now modify existing icons or build a new one. The syntax rules of the source code language are defined earlier in this chapter.
3. You can create and debug the compilation of a single icon in a separate file before adding it to the palette `.src` file. After it is added, specify its position on a palette page using the `PALETTE DEFINITION` keyword at the end of the `.src` file.
4. After you have your icon in source code, run the IA compiler by typing:

```
ia -c infile.src outfile.sog a/b
```

For more on the IA compiler, see the *Interactive Animation User's Guide*.

5. Copy the file `$$SYSBLD/etc/animation.cfg` to your local directory. Note that if you are testing icons within SystemBuild ISIM, you must start Xmath from the directory containing this local copy of `animation.cfg`.
6. Add the line `ICON_DATA_FILE: 'outfile.sog'` to the user definition portion (towards the end) of `animation.cfg`. When you bring up IA you will see the new icon in the palette you created.

17

Components

This chapter describes the SystemBuild components feature. A component encapsulates a SystemBuild SuperBlock hierarchy. Within a model, a component interacts with other blocks much like a conventional SuperBlock does, with the notable difference that component information is stored in a separate catalog. By default, objects or %Variables within a component do not affect objects in the Main SuperBlock hierarchy; however, components can be parameterized using %Variables and variable blocks. These can be exported through the component interface so they are accessible to component users.

Components provide a mechanism for archiving, distributing, and licensing SystemBuild SuperBlock hierarchies. They can be used within a development team to create libraries of commonly-used SystemBuild SuperBlock hierarchies, thereby promoting greater re-use. A component can be encrypted to prevent users from viewing or altering its internal details. It can also be licensed, so that use is restricted to those with a valid license key.

This chapter explains how to both create and use components. A distinction is made between the component user and the component creator.

17.1 Introduction

Components are encapsulated SuperBlock hierarchies. Like pre-defined ISI blocks, they can be referenced in a model, connected to other blocks, and parameterized using %Variables and Variable blocks that are exported through the component's interface. Components with the same number of inputs and outputs and the same parameterization interface can be used interchangeably in your model.

Components have a local *namespace*. Namespace is the scope within which a name can refer to only one entity. By default, a SuperBlock hierarchy (for example, the

Main catalog) can contain only uniquely named items. If an object is introduced into the hierarchy and there is a namespace conflict, then the new item will overwrite the old.

Because a component is an encapsulated hierarchy, the names of all entities within its hierarchy have local scope, therefore, a component can be introduced into a model without contaminating its namespace. For example, a SuperBlock called “foo” in the model will have no effect on a SuperBlock called “foo” within a component in that same model, and vice versa.

You can make a component from an existing SuperBlock hierarchy, as long as it does not contain elements that use resources that can't be saved with the component catalog. [Section 17.3.1 on page 17-9](#) discusses these restrictions.

17.1.1 Component Scope

Encapsulation is another important component trait. We've already mentioned that variables within a component are in a separate namespace, which prevents conflicts with SystemBuild catalog items. To extend that thinking, the catalog in which the component exists must not have another item of the same name, or a conflict will occur.

Sometimes, however, it's useful to have access to parameters within the component scope. For example, you might want to alter %Variables or Variable block values.

A parameter can be visible outside the component's local scope if it is explicitly *exported* by the component's creator. Exported %Variables and variable blocks form the parameters for the component. Any %Variables and variable blocks that are not explicitly exported are not visible outside the component's local scope and are referred to as *contained* variables. The details of exporting variables are discussed in [Section 17.3.7 on page 17-12](#).

17.1.2 Component Interface

A component's interface provides the connectivity between the component's encapsulated SystemBuild SuperBlock hierarchy and the model that references it. A component's interface consists of two parts:

- Inputs and Outputs
- Parameters (exported %Variables and variable blocks)

Component inputs and outputs have a similar behavior to a “built-in” block's inputs and outputs. They are a means of providing data to and obtaining data from the Su-

perBlock hierarchy contained in the component. The component is parameterized using exported %Variables and/or variable blocks that take up namespace in the item (in most cases the user's model) that contains the component's reference. The user "tunes" the component by assigning values to these parameters.

17.1.3 Component Parameter Sets

A component parameter set, or PSET, is a set or subset of values for the component parameters. Component parameter sets are stored in files and can be loaded into an active Xmath session. Parameter sets make it convenient to set the parameters of a component to a known configuration. The details for using parameter sets can be found in [Section 17.2.3 on page 17-7](#) and in [Section 17.2.4 on page 17-7](#).

17.1.4 Component References

Component references are classified based on the location of their definition with respect to the current model. The definition of a component is the catalog that contains the SystemBuild SuperBlock hierarchy that the component encapsulates. Component references can be classified as:

- Regular Component References
- File Component References

Regular component references are those references whose component definition is contained within the current model. File component references are those component references whose component definition is contained in another model file. This is analogous to SuperBlock references and File SuperBlock references. File component references are generally used to refer to a component that is part of a library. The definition of the library component is located in a separate file that the user may or may not have permission to write to. The user includes the model file containing the file component definition in the SETSBDEFAULTS SBLIBS library path.

File component references and regular component references are used in a similar manner, except that one cannot change scope into a file component's catalog. The details of changing scope into a component are discussed in [Section 17.2.5 on page 17-8](#).

17.1.5 Component Access

Components provide different levels of access to the user. These access restrictions combined with encapsulation make components a useful mechanism for archiving, distributing, and licensing SystemBuild hierarchies. A component's access level de-

termines how the user interacts with it in a model. The following sections describes the different types of components and level of user access provided.

NOTE: All components, regardless of access level, encapsulate the SystemBuild SuperBlock hierarchy they represent.

Open Components

Open components provide complete access to the user. The user is able to view the internal details of the open component by changing scope to its catalog. Simulation, code generation and documentation generation are possible for models that reference open components. Open components are generally used within a development team to create libraries of commonly-used SystemBuild hierarchies.

Encrypted Components

Encrypted components provide a convenient way of sharing SystemBuild model information outside the development team, while protecting sensitive information and intellectual property.

A component user cannot view the internal details of encrypted components. By design, encrypted component references are file component references, thus the user cannot change scope into an encrypted component's catalog. The user can simulate and generate documentation for models that reference encrypted components, but the model output will not expose the internal details of the encrypted component.

- Code cannot be generated for models that contain references to encrypted components.
- Encrypted component files cannot be loaded into the SystemBuild editor.

Licensed Components

Encrypted components can be licensed in the public domain as packaged libraries, so that only users with valid license keys are allowed to analyze and simulate models that reference these licensed components. Licensed components behave just like encrypted components except that they require a valid license key to be simulated as part of a SystemBuild model.

TABLE 17-1 Summary of Component Types

Feature	Open	Encrypted	Licensed
Provides Encapsulation	Yes	Yes	Yes
Can be referenced as a regular component reference	Yes	No	No
Can be referenced as a file component reference	Yes	Yes	Yes
Component's details accessible	Yes	No	No
Can change scope into component	Yes	No	No
Can simulate models containing component	Yes	Yes	Yes
Can be simulated without a valid license key	Yes	Yes	No
Can block-step into component during interactive simulation	Yes	No	No
Can generate code for models containing component	Yes	No	No
Can generate documentation for models referencing components	Yes	Yes	Yes
Component details shown in generated documentation	Yes	No	No

17.2 Using Components in SystemBuild Models

This section describes the use of components in a model. Component creation is covered in [Section 17.3 on page 17-9](#).

References to components are similar to references to “built-in” ISI blocks (for example, a gain block). They can be used in the same manner as ISI blocks.

17.2.1 Viewing Components

This section briefly describes how to view components in SystemBuild's Catalog Browser and editor windows. See the Catalog Browser online help for more details.

Figure 17-1 shows a component reference in the SystemBuild editor window. The representation of component references is similar to that of SuperBlock references.

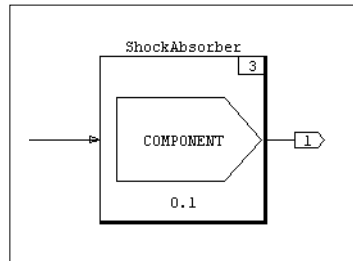


FIGURE 17-1 Default Component Icon

Component references are displayed in the SystemBuild Catalog Browser's Contents view, just as with other block references. If a model contains regular components, then the component's definition is listed in the Catalog view Component folder.

17.2.2 Creating References to Components

References to components can be created in many ways depending on whether the component definition exists in the current model, a reference to the component already exists in the current model, or a custom block for the component exists on a custom palette.

NOTE: References can be created using the standard drag and drop mechanism.

The following methods may be used to create component references in the editor:

1. To reference a component whose definition exists in the current model, go to the Catalog view and open the Components folder. Drag and drop a component from the Components folder into the editor.
2. To create a new SuperBlock reference to a component item, go to the Catalog Browser Catalog view and click on a SuperBlock that references a component. The Contents view will list the blocks. Drag the reference from the Contents view to the editor.
3. To create a file reference to a component defined in a library catalog, go to the Catalog view, click on the Library heading and choose a library catalog. A list of components available in the library catalog is displayed. Drag and drop a component from the Catalog browser list view into the editor.

4. To create reference to a component that exists as an item on a custom palette, drag it from the palette and drop it into the editor, just as with a pre-defined ISI block. See [Chapter 18](#).
5. All SuperBlock references become component references to the top SuperBlock in the Component. Making a component out of a SuperBlock hierarchy from the parent SuperBlock creates a reference to the newly created component. The new component reference will appear in any SuperBlock that contained the component's top-level SuperBlock (the top SuperBlock in the component's hierarchy before the hierarchy was converted into a component).

17.2.3 Controlling Component Parameters

To view the parameters of a component, bring up the Component block dialog for a component reference. Any exported %Variables are listed on the Parameters tab.

To change the value of a component's parameter, an Xmath partition must be specified for the component reference in the Component block **Partition** field. Once this Xmath partition has been specified, the parameters of the component reference will obtain their values from it. If an Xmath variable with the same name as the component's parameter is not present in the Xmath partition, the parameter's default value is used. The column **In Partition** specifies whether the corresponding parameter is present in the specified Xmath partition. To change the values of a variable in the component reference's Xmath partition, change it in Xmath, or, alter the value in the Component block dialog.

17.2.4 Loading Component Parameter Sets

Parameter sets can be used to load in a particular set of values for a component reference's parameters. Before you can do this, you must specify an Xmath partition and store the parameter sets for the component there. Once the Xmath partition for a component reference is specified, the user can choose between **Palette** or **User** parameter sets in the Component block Parameters tab.

- Palette parameter sets are those that are contained as part of the custom block packaging in the custom block directory specified by the palette
- User parameter sets are created by the end-user and loaded into the current working session of SystemBuild using `Psets_Load`. See [Section 17.4 on page 17-14](#).

Available parameters sets are listed in the combo box in the **Parameter Sets** section. To load the parameter set into the component reference's Xmath partition, choose one of the listed parameter sets and press the **Load PSET** button. Loading a parameter set

creates the variables specified in the parameter set in the specified Xmath partition [if they do not already exist] and assigns them the values specified in the parameter set. The set of values specified in a parameter set may be a subset of the component's parameters.

NOTE: Loading a parameter set changes the values in an Xmath partition, therefore all references that use the Xmath partition will be affected. If this is undesirable, then the references should specify different Xmath partitions. This mechanism provides a way of coupling related component references by specifying the same Xmath partition to these references.

[Section 17.4 on page 17-14](#) contains more details on PSETs.

17.2.5 Changing Scope into a Component Catalog

Typically, an end-user does not need to know the internal details of a component, but if the need arises you can change scope into a component's catalog to view its contents. To do this, select a component in the component section of the Catalog browser, then select View→Component Catalog, or, raise the Quick Access menu and select Component Catalog. Note, not every component allows a user to scope into its catalog (see [Table 17-1 on page 17-5](#)).

When you enter a component's catalog scope, the text field just above the Catalog view reflects the current catalog. The contents of the user's model are replaced with the component's SystemBuild SuperBlock hierarchy.

The catalog browser treats a component catalog exactly the same as other catalogs. If a component contains other components, you can change scope into the child components' catalogs in the same fashion. To navigate down, open a component catalog; to navigate to the parent, click the directory icon. To return directly to the main model, choose View→ Main Catalog.

NOTE: If any changes are made within a component catalog, the component needs to be re-componentized. See [Section 17.3.8 on page 17-13](#) for details.

17.2.6 Simulating Models with Components

Models that contain components are simulated in the same way as other models. If the model contains licensed components, however, the user must possess a valid license key for the licensed component in order to analyze and simulate the model.

17.3 Creating Components

This section first describes component concepts so that you, as the component creator, can accurately design the SystemBuild SuperBlock hierarchy that will become a component. The name of the component created will be that of the top-level SuperBlock in the SuperBlock hierarchy that is transformed into a component. The actual creation of the component is done using the Component Wizard, which is discussed in [Section 17.3.7 on page 17-12](#).

17.3.1 Restrictions on Component SystemBuild Hierarchies

SystemBuild components encapsulate SuperBlock hierarchies. Therefore, any modeling element that breaks this encapsulation is disallowed in a component hierarchy. These elements cannot be included in components:

- Data Stores
- References to File SuperBlocks or File Components
- IA blocks
- SuperBlock or Component references that specify a partition
- UCBs or MathScript blocks that use global variables

If a hierarchy contains one of these elements, the componentization will fail.

Technically, the timing attributes of a discrete SuperBlock hierarchy (sampling rate and time skew) also invalidate encapsulation, because they cannot be controlled through the component's interface but are still visible outside the component. However, SystemBuild allows you to componentize a discrete SuperBlock hierarchy, with the restriction that its sampling rate and time skew are fixed at the time of component creation. A better method, however, is to convert the top-level SuperBlock in the discrete hierarchy to a Procedure SuperBlock, so that the component's encapsulation is not violated. A component that contains a Procedure SuperBlock as the top-level SuperBlock will assume the timing characteristics of its parent.

17.3.2 Understanding Parameterization of Components

A component user can tune SystemBuild components by assigning values to the parameters exported through the component's interface by the component creator. These exported parameters can be %Variables or variable blocks. The creator needs to identify the component parameters that are important for tuning, ensure that they are either %Variables or variable blocks (else create %Variables or variable

blocks and use them in the hierarchy), and export them at the time of component creation.

All %Variables and variable blocks that are not exported at the time of component creation assume local scope and are referred to as contained variables. Contained variables are not visible to the user, so it is important that the component creator carefully assign appropriate default values. The default values used for the contained variables are the block defaults unless the component creator specifies otherwise at the time of creation. These default values are used if the component user does not define the exported variables in the component reference's partition.

17.3.3 Understanding the Component Scope

The component creator must understand the concept of component scope in order to manage the component scope hierarchy. A component's scope is the collection of the items in the component's private catalog and the parameters (%Variables and variable blocks) that are present in the component. These parameters include all of the exported variables of child components that the component may contain. The component's scope is associated with two namespaces -- a catalog namespace and a variable namespace. The names of the component's catalog items (SuperBlocks, State Transition Diagrams and child components) occupy the component's catalog namespace. This implies that all names must be unique within a component's private catalog. Within a component all variables should refer to the same entity; component variables with the same name are assumed to have the same datatype and dimensions. If this assumption is violated, the Component Wizard will report an error at the time of component creation (see section [Section 17.3.7 on page 17-12](#)).

When a component contains another component, the exported variables of the contained component assume the scope of the parent component. The parent component in turn may export some or all of the contained component's exported variables. If any of the contained component's exported variables are not exported by the parent component, they become the parent component's contained variables and will assume the value specified by the parent component, if any. Otherwise they will default to the values assigned by the child component's creator. This is important to note because all the exported variables of the child component are no longer accessible to the user.

If a component contains two child components "foo" and "bar" and each child exports a variable with the same name, then the two components are coupled. As stated in [Section 17.3.1 on page 17-9](#), component references within a component cannot specify a partition. Therefore the references to "foo" and "bar" cannot specify partitions. The assumption is that the two components are coupled because they have exported the same parameters.

17.3.4 Mapping Exported Variables

Mapping is a mechanism by which the component creator can parameterize a component with parameters that do not exist inside the component's SuperBlock hierarchy. The component creator creates variables called mapping parameters and makes them visible to the component user through the component's interface. The component creator then uses these mapping variables to set the values of %Variables in the component's SuperBlock hierarchy using mapping equations (valid Xmath statements). To the component user, the mapping variables look the same as any other exported variable. The component user can assign values to the mapping parameters which in turn are used to assign values to the %Variables in the component's SystemBuild SuperBlock hierarchy through the mapping equations.

If a component uses mapping, then the only parameters visible to the component user are the mapping variables and the exported variable blocks. The component creator must decide which of the %Variables in the hierarchy to export. Mapping can be performed only on %Variables identified as exported variables.

If a component uses mapping, every exported %Variable must be mapped. For example, if a %Variable "foo" is designated as an exported variable in a component that uses mapping, and no mapping variable is assigned to "foo", an inconsistency occurs because "foo" is not visible to the component user and therefore its value can never be modified.

One of the advantages of using mapping, as opposed to modeling the mapping equation inside the component's SuperBlock hierarchy, is that mapping allows constants without complicating the model. Another major advantage is that the exported variables are not computed every time step during simulation but only at the start of the simulation, or if the mapping variable is changed during RVE (Runtime Variable Editing). Mapping is, therefore, more efficient during simulation and keeps the model simple.

17.3.5 Customizing the Component Dialog

There may be situations where the component creator wants to have a custom dialog associated with the component's references. For example, the creator may want to use a graphical representation of the component's parameters so that the component user can modify the parameters by dragging points on a graph rather than entering them in the fields provided by SystemBuild's native dialog.

Components allow the creator to associate a custom dialog with the component's references. This custom dialog can either replace or augment the native SystemBuild dialog for component references. If the component creator decides to override the native dialog, the custom dialog is displayed when the component user brings

up the dialog for component references. If the component creator decides to augment the native SystemBuild dialog, then the custom dialog is displayed when the user brings up the dialog for component references, followed by the native dialog when the user dismisses the custom dialog.

A custom dialog must be programmed by the component creator with an MSF. The interface to the MSF is the same as that for the function `sysbldEvent` (see the on-line help), although the interface is not used when you specify an MSF. The event string will be `CustomDialog`.

17.3.6 Documenting the Component

The component creator should adequately document a component so that it is self-describing to the component user. Proper and complete documentation is essential to ensure component re-use, which is the primary motive for creating components. The list of good documentation practices include:

- Name the component so that it accurately reflects the abstraction that it represents. Avoid commonly-used names so that the user does not have naming conflicts when the component is included in a model.
- Name the exported variables so that their meaning is clear to the component user.
- Create a document that describes the component's inputs/outputs, interface, and functionality. To assist the user, distribute the document as part of the component. See [Section 17.6 on page 17-16](#).
- Label the component's inputs and outputs.

17.3.7 Creating Components Using the Component Wizard

To transform a SystemBuild SuperBlock hierarchy into a component, choose the top-level SuperBlock in the Catalog Browser, then select `Tools→Make Component`. (Note, you will not be able to create a component if any object in the future component catalog is open in an editor.) The Component Wizard is invoked. Enter the requested information in each of the fields in the Wizard's pages to create the component.

Before invoking the wizard make sure that you are prepared to answer the following questions:

- Does the top-level SuperBlock of the hierarchy have the name that you want the component to assume?
- Does this component require a custom dialog? If yes, then what is the name of the MSF that invokes the custom dialog?
- Will this component use mapping? If yes, gather the mapping parameters for this component in a single Xmath partition. The Component Wizard will use the designated partition to obtain the parameter dimensions. The parameter values will become the default values for the component's mapping parameters. The new component will no longer need the original Xmath partition or its contents.
- Which of the %Variables and variable blocks in the SystemBuild SuperBlock hierarchy will be exported through the component's interface?
- Do you want to specify default values for the %Variables other than the block defaults? If yes, gather the variables into a single Xmath partition. The Component Wizard will use the values of the variables found in the specified partition. Once created, the component will no longer need or refer to the partition that contains the defaults for the exported variables.
- Does the component use mapping? If so, make sure that you prepare the equations you will use to map the mapping variables to the component's exported variables. Mapping can also be performed by invoking a user-defined Xmath command or function; write and test your MathScript, and have the calling syntax ready before you componentize.

17.3.8 Modifying Components

Once a component has been created it can be modified (assuming it is not encrypted).

To modify a component's interface go to the Catalog browser, select the component, then select Tools→Edit Component. Again, you cannot modify a component if any member object is currently displayed in the editor. The Component Wizard is invoked and the component is re-componentized with the new interface that the user specifies.

To modify the internal details of a component, change scope to the component's catalog as discussed in [Section 17.2.5 on page 17-8](#) and make the necessary changes. When the scope is changed back to the component's parent catalog, the Component Wizard is invoked. This because the component's interface may need to be altered to

reflect the changes made to its internal details. Failure to re-componentize may result in inconsistencies if the changes to the component's hierarchy affected the component's interface. The steps in re-componentizing a component are similar to creating a component, as described in [Section 17.3 on page 17-9](#).

17.3.9 Unmaking a Component

To reduce a non-encrypted component to a SuperBlock hierarchy in its parent catalog, select the component, then select Tools→Unmake Component. This will replace the component in the parent catalog with the SuperBlocks within the component. If there are any name conflicts, the user will be asked to choose whether to keep the existing SuperBlock in the parent catalog or overwrite it with the one from the component.

17.4 Creating and Using Parameter Sets

A parameter set is a set of values for the parameters, or a subset of the parameters, of a component. Component parameter sets (PSETs) can be stored in files and loaded into an active SystemBuild session. Parameter sets make it convenient to set the a component's parameters to a known configuration.

This section describes the creation and loading of parameter sets for components.

A parameter set is stored in an Xmath file. To create a parameter set file, use the `Psets` function to create a MathScript object that represents the parameter set. Note, this is not the same as using the Xmath SAVE command.

EXAMPLE 17-1: Creating and Saving PSETs

The following example creates, saves, and loads a parameter set called “Monster-Shocks” for a ‘shock_absorber’ component that has exported the parameters ‘spring rate’ and ‘setting’.

1. The first step is to define the PSET using the `Psets` function. The `Psets` function takes as arguments the component name, the PSET name, and the name and values of variables that will make up the PSET.

```
pset1 = Psets("shock_absorber", "MonsterShocks",  
            {setting = 2, spring_rate = 3})
```

The `Psets` function formats the inputs, producing a `MathScript` object that is assigned to the output variable `pset1`. Note, the name 'MonsterShocks' will only be visible from the component reference dialog.

2. The variable `pset1` now contains the PSET 'MonsterShocks'. The newly created `MathScript` object needs to be registered as a parameter set by calling the `Psets_AddToList` command.

```
Psets_AddToList pset1
```

3. The component interface expects a PSET to be the only object in a single file. To isolate the object in a file, call the `Psets_Save` command.

```
Psets_Save pset1, "steve.pset"
```

Once the parameter set has been created and saved into a file they can be distributed along with a component as a custom block on a custom palette; see [Section 18.2.1 on page 18-6](#).

EXAMPLE 17-2: Loading PSETS

To load a `pset` as a user parameter set for a component, issue the `Psets_Load` command from `Xmath`.

```
Psets_Load "steve.pset"
```

Once the parameter set has been loaded it is visible in the Component block dialog Parameters tab combo box if the **user** radio button is selected. A partition for a component reference must be specified before in order to activate the Parameter Set section in the Component block dialog.

The parameter sets that are distributed with the component (as a custom block on a custom palette) are automatically loaded when a partition is specified for a component reference. The palette parameter sets can be viewed in Component Block Dialog Parameter Set area if the **Palette** option is selected.

17.5 Using SBA with Components

The following is a list of SBA commands that support components.

- `deletecomponent`
- `makecomponent`
- `querycomponent`
- `querycomponentoptions`

For a complete description of the commands see the online help.

17.6 Distributing SystemBuild Components

A component can be distributed either as a model file or as a custom block on a custom palette.

- If it is distributed as a model file then the user can load the component like any other model file, or add the model file to the File SuperBlock Library by modifying the `SBDEFAULTS SBLIBS` keyword.
- A component can also be made into a custom block and put on a custom palette.

The latter method is more versatile because the component creator can include all the auxiliary files such as a custom dialog file, documentation file, etc. For a detailed description of distributing a component as a custom block, refer to [Section 18.2 on page 18-6](#).

Encrypting and Licensing Components

Components can be encrypted using the `ENCRYPT` utility described in the online help.

The `ENCRYPT` utility can only be used on a SystemBuild file that contains a single top-level component. To isolate a component, load the model that contains the definition of the component that needs to be encrypted. In the Catalog Browser, select the Component, then select `File→SaveAs`. In the Save dialog SuperBlocks field, choose **Selected**, then press **OK**. The saved component will be the top-level component in the new file.

In addition to a simple encryption, which merely keeps component users from altering the component, you can optionally specify a license feature name. Users must then have a valid license key to use it.

17.7 Examples

The examples in this section show common component creation tasks.

17.7.1 Encapsulating a SuperBlock Hierarchy

In this example, a simple SuperBlock hierarchy named 'proto' is converted to a component. A SuperBlock, 'test', exists in the main catalog and is referenced within the SuperBlock proto. This procedure demonstrates that encapsulating the proto SuperBlock as a component creates a second definition of test inside the component; changes made to the SuperBlock test in the component catalog will not affect the definition of test in the main catalog.

1. Load the model `encap.dat`:

```
copyfile "$SYSBLD/examples/components/encap.dat"  
load "encap.dat"
```

2. In the Catalog Browser, highlight the SuperBlock 'proto'. Select Tools→Make Component.
3. The component wizard appears. To accept the default settings, press **Finish**.

In the Main Catalog view, observe that the SuperBlock 'proto' has moved from the SuperBlocks folder to the Components folder.

4. In the Catalog view, click the components folder to display the components in the Contents view. Click the proto component to display a flat list of its members in the Contents view.
5. Open the component 'proto'. Navigate into the SuperBlock 'test'. Change the gain block value to 99. Press **OK**.
6. Click the Parent icon to return to the SuperBlock then click the parent icon again to change scope back to the Main catalog. Since the component has been changed, the Component Creation Wizard will appear as you leave the component's scope. Click **Finish** to re-package the component.
7. In the SuperBlock 'ex1', navigate into the 'test' SuperBlock and inspect the gain value. The value is still the original default value of 1.0. Changing the Super-

Block inside the component had no effect on a SuperBlock with the same name outside the component scope.

17.7.2 Exporting Component Parameters

This example uses a simple model that contains %Variables and variable blocks to demonstrate exporting parameters in components.

1. In Xmath, create a partition named ex2, then make it the current partition.

```
new partition ex2
set partition ex2
```

2. Load the example data:

```
load "$SYSBLD/examples/components/param.dat"
```

3. In the Catalog view, expand ex2, locate the SuperBlock 'top' and make a component out of it. In the first window of the Component Wizard, click **Next** to accept all default settings.
4. The Exporting Definition screen appears. Export the %Vars 'temperature' and 'humidity', and the Var Block 'varblock2'. At the bottom of the form, click **Replace**. Type **ex2** in the partition name field. Click **Finish**.

Component Creation Wizard

Choose Variables To Be Exported

Contained % Vars	Exported % Vars
friction	humidity temperature

Contained Var Blocks	Exported Var Blocks
varblock1	varblock2

Replace Variable Defaults From Xmath Partition

Replace Partition Name:

< Back Finish Cancel

- Open the SuperBlock 'ex2', then open the block dialog for the component reference 'top'. Note the exported %Variables are visible (see [Figure 17-2](#)). Click on the parameter name to view the value in the spreadsheet below.

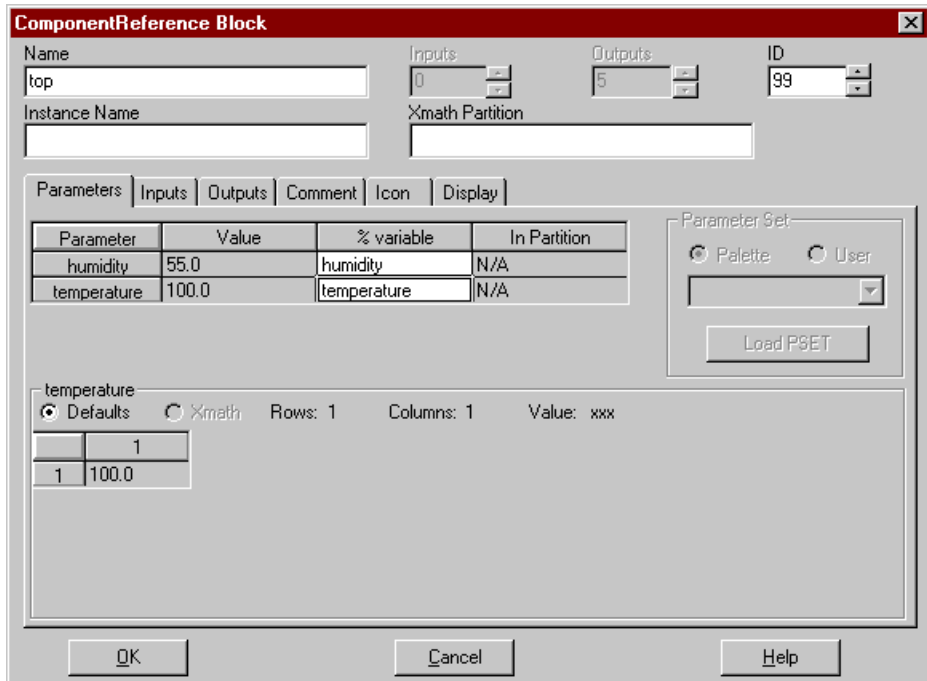


FIGURE 17-2 Exported Variables in a Component Reference Dialog

17.7.3 Using the Parameter Set Interface

This example will use parameter sets (PSETs) with the component 'top' created in [Section 17.7.2](#).

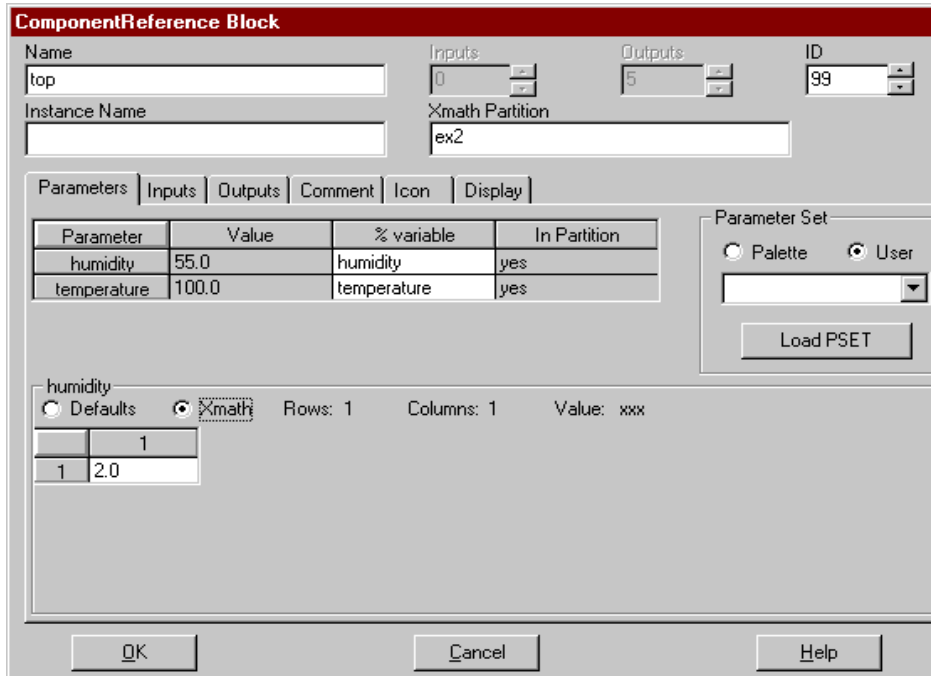
- In the Xmath command area, make sure you are in partition ex2 then define PSETs for the component 'top':

```
p1=psets("top","p1",{temperature=9,humidity=5,varblock1=-2});
p2=psets("top","p2",{temperature=3,humidity=4,varblock1=-6});
p3=psets("top","p3",{temperature=6,humidity=2,varblock1=-9});
p4=psets("top","p4",{temperature=7,humidity=1,varblock1=-1});
```

- Now register the new PSETs:

```
psets_addtolist p1;
psets_addtolist p2;
psets_addtolist p3;
psets_addtolist p4;
```

- From the SuperBlock 'ex2', open the 'top' component reference dialog. In the **Xmath Partition** field, define a new partition 'test' to receive copies of the pset variables. This activates the Parameter Set options on the Parameters tab. (Note, the fields will not activate until you tab out of the **Xmath Partition** field.)
- In the Parameter Set area, select **User**. The combo box will display the psets defined above. Select a pset, then press **Load PSET**.
- PSET values are now available in the block dialog. Select a parameter (either humidity or temperature). In the display area below, select **Defaults** to view the original %Var, or select **Xmath** to view the value from the PSET you loaded.



- Click **OK**. The PSET is unpacked to the partition specified in the form. Any simulation or code generation will use the parameters in the new partition for the exported variables.

17.7.4 Interface Mapping

This example demonstrates how a component creator can use mapping to modify component outputs without changing the model itself.

Assume the temperature in the sample model is in units of degrees Fahrenheit, but a component user needs an interface that has temperature in units of Celsius. Mapping can be used to accommodate the user.

1. Load the example data:

```
load "$SYSBLD/examples/components/param.dat"
```

2. Create a new mapping partition `ex_map`. Inside this partition, create a variable `c` and give it a value of 8:

```
new partition ex_map
ex_map.c = 8;
```

3. Make a component out of the SuperBlock `top`. In the first dialog of the component wizard, select **Use Mapping**. In the field **Xmath Partition to Get Mapping Variables From**, enter `ex_map` as the partition to get mapping variables from. Press **Next**.

The screenshot shows the 'Component Creation Wizard' dialog box. It is divided into three main sections:

- Component Name:** A text box containing the value 'top'.
- Dialog:** Two radio buttons are present: 'SystemBuild' (which is selected) and 'Xmath Callback'. Below these is a text box labeled 'Xmath Command' which is currently empty.
- Exported Variable Mapping:** A checkbox labeled 'Use Mapping' is checked. To its right is a text box labeled 'Xmath Partition to get Mapping Variables from' which contains the value 'ex_map'.

At the bottom of the dialog, there are three buttons: '< Back', 'Next >', and 'Cancel'.

4. In the next dialog, export the parameter `temperature` only. Click **Next**.
5. The parameter mapping form appears next. In the **Parameter** field, enter the previously specified mapping variable `c`. Click **Add**.

Enter the following equation in the **Mapping Expression** field:

```
temperature = (9/5) * c + 32;
```

The screenshot shows a dialog box titled "Component Creation Wizard (Xmath Parameter Definition)". It contains the following fields and controls:

- Mapping** section:
 - Exported Vars [Reference Only]**: A text box containing "temperature".
 - Xmath Mapping Partition:** A text box containing "ex_map".
- Parameter** section:
 - Parameter**: A text box containing "c".
 - Add**: A button.
 - Remove**: A button.
 - Parameter List**: A list box containing "c (1,1)".
- Mapping Expression: Exported Vars = f (Parameters)**: A text box containing "temperature = (9/5) * c + 32;".

At the bottom of the dialog are three buttons: "< Back", "Finish", and "Cancel".

- Click **Finish**. Open the block dialog form for the component reference, and note the Parameters tab displays only the %Variable c, with an initial value of 8.

17.7.5 Using a Custom Dialog

In this simple example, the Xmath Dialog commands `GetChoice` and `GetLine` are used to set up a special dialog where the user can change the Instance Name of the component reference. Once the new instance name is entered, a `GetChoice` dialog prompts the user for the regular SystemBuild component reference dialog.

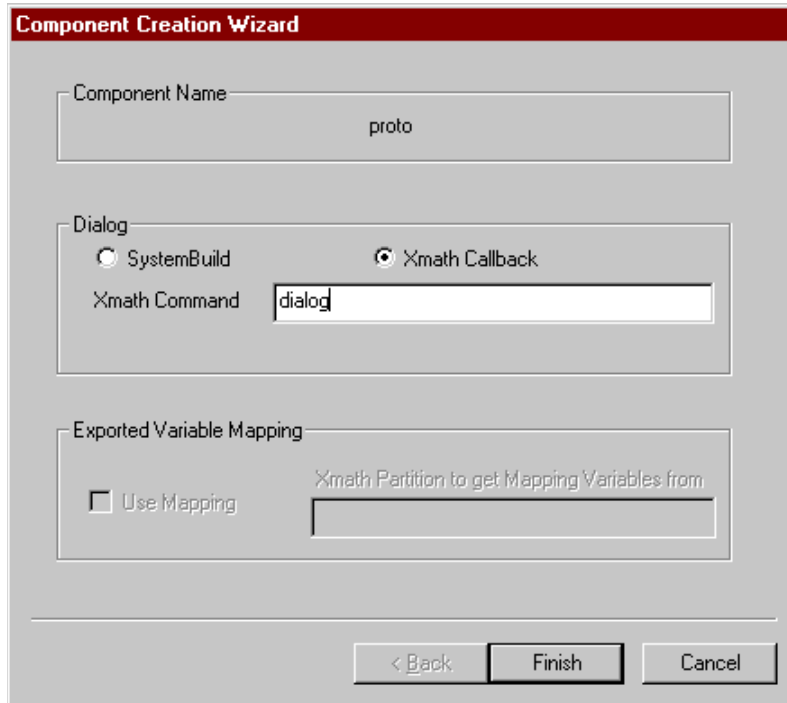
- Copy the sample MathScript function to your local directory.

```
copyfile "$SYSBLD/examples/components/dialog.msf"
```

- Load a simple demonstration model:

```
load "$SYSBLD/examples/components/encap.dat"
```

- In the SuperBlock `ex1`, make a component out of the SuperBlock 'proto'. In the component creation wizard go to the **Dialog** section and enable **Xmath Callback**. In the **Xmath Command** field, type `dialog`.
- Click **Finish**. In the Catalog browser, open `ex1` for editing.



The screenshot shows a dialog box titled "Component Creation Wizard" with a dark red header. It contains three main sections:

- Component Name:** A text input field containing the text "proto".
- Dialog:** Two radio buttons are present: "SystemBuild" (unselected) and "Xmath Callback" (selected). Below them is a text input field labeled "Xmath Command" containing the text "dialog".
- Exported Variable Mapping:** A checkbox labeled "Use Mapping" is unchecked. To its right is a text input field labeled "Xmath Partition to get Mapping Variables from" which is currently empty.

At the bottom of the dialog, there are three buttons: "< Back", "Finish", and "Cancel".

5. Open the component 'proto' block reference. A single text entry box will appear from the Xmath Command window. Enter a SuperBlock name and click **OK**.

The next dialog gives you the option of opening the standard SystemBuild dialog for the component reference, or bypassing it.

18

Palettes and Custom Blocks

The palette browser organizes blocks in logical groupings represented as tree nodes (shown on the left side of the palette browser). Blocks in the current grouping are shown on the right.

This chapter describes how to customize the palette browser tree structure (with custom palettes) and contents (with custom blocks). The tree structure can have multiple levels, each defined by a separate palette file. Palette files can contain personalized organizations of predefined ISI blocks, customized versions of ISI blocks, and other palettes. A custom block is a block that has been saved so that it includes specified parameter values, labels, or names. Functional blocks, STDs, Super-Blocks, Components and DataStores can become custom blocks.

[Section 18.1](#) shows how to create and load a palette that uses predefined ISI blocks. [Section 18.2](#) shows how to create a custom block, including how to add special startup, help, and bitmap files. Examples will show how to incorporate custom blocks into custom palettes then load them into the palette browser. The last section discusses new SBA support for custom blocks and palettes.

18.1 Custom Palettes

By default, the palette browser displays all ISI defined blocks in the palette Main. The Main palette contains a set of palettes that each contain a collection of blocks with similar properties (algebraic, dynamic, etc.) You can create your own customized palette file that contains blocks or palettes organized as you see fit. Customized palettes can be added to or deleted from the existing tree structure. Each tree node in the palette browser is defined by a separate custom palette file. To create multiple levels in the tree hierarchy, define a new palette file for each level.

18.1.1 Creating Palette Files

A palette file is composed of lines of palette items. Each palette item points to an ISI built-in block, a custom block, or another palette file. The following example shows how to create a simple custom palette file.

EXAMPLE 18-1: Trivial Custom Palette Example

This example shows how to create a new palette that shows only the Dynamic Blocks palette and the Gain block.

1. In the current working directory, create a text file named `example.pal` that contains the following two lines:

```
palettefile="ISI_dyn.pal" title = "My Dynamic"  
blockdirectory="ISI_Gain" title = "My Gain"
```

Save the file.

2. Open the palette browser. Choose File→Open. Select `example.pal`. Click **OK**.

The new palette appears under the heading **example**; click on **example**, and the ISI Gain block appears to the right. Double-click on the label **example** to see the Dynamic palette.

3. To close the palette, highlight **example**, then select File→Close.

The example's tree structure is very simple. A base node, **example**, has one node, **My Dynamic**, attached to it.

EXAMPLE 18-2: Trivial Custom Palette Example with Nesting

This example shows how to create multiple levels in the palette browser hierarchy.

1. In the current working directory create a text file named `levelone.pal` that contains the following line:

```
palettefile="leveltwo.pal"
```

2. Create a text file named `leveltwo.pal` that includes the following line:

```
palettefile="example.pal"
```

(This file, `example.pal`, was created in [Example 18-1](#)).

3. Open the palette browser. Choose File→Open. Select `levelone.pal`. Click **OK**.

The new palette appears under the heading **levelone**. Double-click on **levelone**. A node with label 'leveltwo' will appear. Double-click on **leveltwo**. The label **example** appears as a child of **leveltwo**. Single-click on the **example**, and the ISI Gain block appears to the right. Double-click on **example** (or click on the + next to the label **example**) to see the Dynamic palette.


Palette files can be called recursively; for example, **leveltwo** can contain an entry for **levelone**. When recursion is detected, the palette browser displays a single level of recursion (infinite levels of recursion will not be represented).

4. To close the palette **example**, highlight **levelone**, then select File→Close.

NOTE: Close can only remove entire palettes; there is no way to close off **leveltwo**, but keep **levelone**.

EXAMPLE 18-3: Closing and Reloading the Default ISI Palette

You can completely remove the default ISI palette at any time.

1. Highlight the label **Main**, then select File→Close, or click the  icon.
2. To reload the default palette, select File→Load SystemBuild Palette.

18.1.2 Palette File Syntax

The examples above show the basic syntax of all palette files. Palette files can contain statements that define blocks (`blockdirectory`) and lines that refer to other 'nested' palette files (`palettefile`).

The exact statement syntax for palette entries are:

```
blockdirectory = "BlockDirectory" title="block-title-on-palette"
                help = "path-to-html" icon="path-to-bitmap"

palettefile = "path-to-palette" title="palette-title"
```

- Each statement must be on a single new line (above we show a continued line, but in the file it must be a single line per item). No commas or line terminators (for example, a semicolon) are required.
- Valid keywords are `title`, `help`, and `icon`. These keywords are explained on [page 18-12](#).

- Block and palette statements may be interspersed in the file.
- Comments are not supported in custom palette files.

Here are some examples of valid custom palette file statements:

```
PaletteFile = "/homes/test/mypal.pal"
BlockDirectory = "ISI_LinearInterp"
BlockDirectory = "F:\blocks\custgain"
PaletteFile = "../mypal.pal" title = "test"
PaletteFile = "ISI_trg.pal"
```

PaletteFile

PaletteFile points to a palette file. Built-in palettes can be referenced with the following PaletteFile values:

ISI_sysbld.pal	Includes all of the default SystemBuild palettes.
ISI_sup.pal	SuperBlock
ISI_alg.pal	Algebraic
ISI_pwl.pal	Piece-wise Linear
ISI_dyn.pal	Dynamic
ISI_imp.pal	Implicit
ISI_trg.pal	Trigonometric
ISI_pel.pal	Power Exponential Logarithmic
ISI_trn.pal	Coordinate Transformation
ISI_sgn.pal	Signal Generator
ISI_log.pal	Logical
ISI_usr.pal	User Programmed
ISI_kbb.pal	Artificial Intelligence
ISI_ntp.pal	Interpolation
ISI_sc.pal	Software Constructs
ISI_mtx.pal	Matrix Equations
ISI_iai.pal	Interactive Animation (requires separate license)
ISI_arc.pal	Archived (obsolete blocks)

In addition to ISI palettes, `PaletteFile` can also be the path to a palette file. The path can be in any of the following format:

1. An absolute path, such as `c:\palettes\my.pal` (Windows), or `/homes/Sun-Platform/users/usrl/palettes/my.pal` (UNIX).
2. A relative path from the location of the current palette file.
3. A path prefixed by an environment variable, such as `${PALETTE_PATH}\palettes\my.pal`, or `%PALETTE_PATH%/palettes/my.pal` where `PALETTEPATH` is the path to a palette file and `BlockPath`.

BlockDirectory

The block directory is used to specify native ISI blocks and custom blocks (see [Section 18.2 on page 18-6.](#)) To specify a default ISI block, simply prepend `ISI` onto the name of the block. See [Palette Icon on page 18-7](#) for hints on using graphics on a custom palette.

18.1.3 Defining the Default SystemBuild Palette (startup.pal)

Instead of loading a custom palette file every time SystemBuild is started, you can define a file named `startup.pal` which will be read automatically when SystemBuild starts. When the palette browser is started, it looks for the filename `startup.pal` in different directories in the following order:

1. In the current working directory.
2. `%HOME%\xmath\startup.pal` on Windows or `$HOME/xmath/startup.pal` on UNIX.
3. The path defined by the environment variable `%PALETTE_PATH%` on Windows, or `$PALETTE_PATH` on UNIX. You must manually define `PALETTE_PATH` locally; it is not predefined at install.
4. The default SystemBuild palette directory, `%SYSBLD%\palettes` on Windows, or `$SYSBLD/palettes` on UNIX.

EXAMPLE 18-4: Startup Palette file.

1. Copy the file `example.pal` (created in [Example 18-1 on page 18-2](#)) to `startup.pal`.
2. Exit SystemBuild, then restart it.

You will see that the label Main now contains only the blocks you defined in `startup.pal`. The default ISI palette is not loaded.

If you want to add the default ISI palette, add the following line to `startup.pal`:

```
palettefile="ISI_sysbld.pal"
```

18.2 Custom Blocks

This section describes how to create and instantiate custom blocks. A custom block can start as one of the pre-defined blocks provided with SystemBuild, or a SuperBlock or component created from pre-defined blocks. A block is a good candidate for becoming a custom block if you find yourself repeatedly setting the same parameters for a given block type. With the Custom Block wizard, you can save block parameters as part of a custom block definition.

18.2.1 What Kinds of Blocks Can Be Customized?

Any block, SuperBlock, STD, DataStore, or Component present in the SuperBlock Editor may be defined as a custom block.

- | | |
|--------------------|---|
| Basic block | If a block is selected, you may define any combination of labels, parameter values, datatypes or other block attribute to be the default values for the custom block. If blocks such as UserCode Blocks are used, the source file and other dependent files are stored with the custom block. |
| SuperBlock | If a SuperBlock is used for the custom block, the entire SuperBlock hierarchy is included. |
| STD | The custom block will contain the entire State Diagram. |
| DataStores | DataStores are similar to basic blocks. You can define any DataStore attribute to the default parameters of the custom block. |
| Components | Components can be transformed to custom blocks. Parameter Sets and a custom dialog may be associated with the component. |

18.2.2 Additional Custom Block Features

In addition to the basic features listed above, each custom block may optionally have defined a startup file, a bitmap that appears in the palette browser, and a custom help file.

Startup Files

When a custom block is instantiated, an optional MathScript function defined by the custom block can execute. This allows 'last minute' customization of the block before it is drawn in the editor window. The Startup Function must have the following interface:

```
function [result] = funName(sbname, blkid)
...function body....
endfunction
```

NOTE: The root name of the startup file and the startup function must be the same as the name of the custom block.

Palette Icon

Blocks on the custom palette can be assigned a bitmap for display in the *palette browser*. (Note, the icon that appears on the block in the SystemBuild editor is assigned when the custom block is defined. Custom icons created with IA will not display on the palette.) If the custom block imports a bitmap image, there is no cross-platform restriction; SystemBuild will display any of the accepted bitmaps (a .gif, .jpg, .bmp, or .xpm file) on any platform.

CAUTION: **Although the SystemBuild editor displays all supported bitmaps on any SystemBuild platform, icon file paths in palettes are platform dependent.**

If you have an incompatible path in a palette file, you will not be able to instantiate your block in the SystemBuild editor because the icon file cannot be found.

You can use the same icon file for display in the editor and the palette as long as relative paths are used. If relative paths are properly used, custom blocks and palettes can be shared across platforms.

For example, look at the directory structure in [Figure 18-1 on page 18-8](#). The palette file is the parent, and a custom block resides in a subdirectory. Both the custom block and the palette file (shown in the overlay) can use a relative path to refer to the icon. This hierarchy will work across platforms whenever the top of the hier-

archy (in this case /palettes) is the current working directory. The current working directory is set in Xmath before SystemBuild is started:

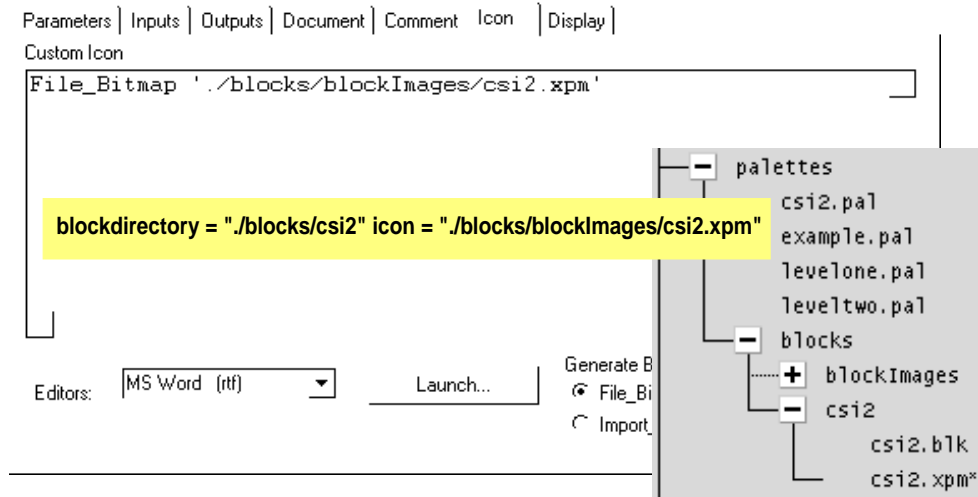


FIGURE 18-1 Sample Directory Structure where Palette File Uses Custom Block Graphic

Note, although the custom block process copies the file bitmap to the block directory, you cannot get rid of the directory the image came from; the block looks in the location specified on the Icon tab.

Help Files

File that appears when help is requested for the custom block.

18.2.3 Creating a Basic Custom Block

This section describes the procedure to create a basic custom block. Subsequent sections will show how to create custom blocks with startup, help, and custom icon files.

Creating and using custom blocks is a three step process. First, use the Custom Block Wizard to create the custom block. The results of this step are stored in a directory defined by data entered in the wizard. Second, modify a custom palette file to include the new custom block. Third, load and display the custom palette file in the palette browser.

1. Create a custom block using the Custom Block Wizard:
 - a. In the SystemBuild Editor, create or select a block that has the properties you want to reuse.
 - b. Select **Edit** → **New Custom Block...** to bring up the Custom Block Wizard. The Custom Block Wizard displays multiple panels to specify information about the custom block related files.

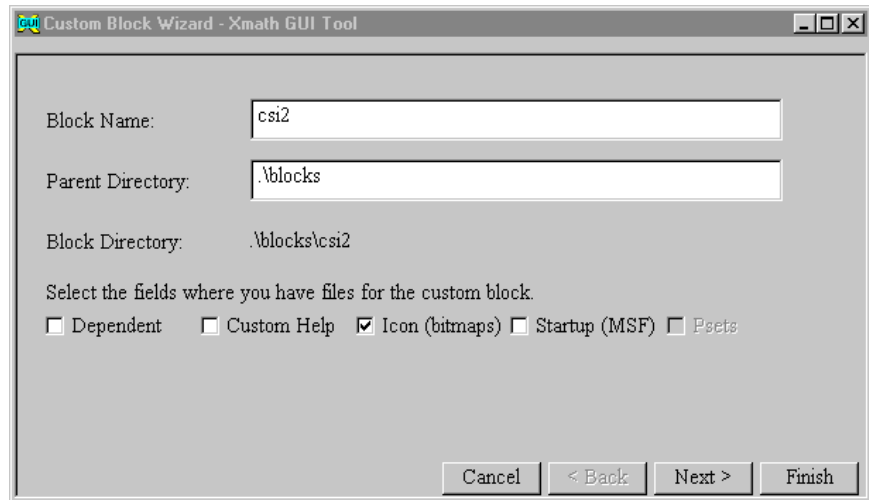


FIGURE 18-2 Custom Block Wizard Fields

- c. By default the **Block Name** field of the Custom Block Wizard shows the name of the block selected in the editor. You can accept or change this name; if no name is specified, the wizard displays the block type in the name field. The block name will be used to name a subdirectory in the directory specified in **Parent Directory**.
- d. On the **Parent Directory** field, type in the location where the new custom block subdirectory will be created. The default is the current Xmath working directory. If you want your files to be portable, specify a relative path (as discussed in [Palette Icon on page 18-7](#), and as shown in [Figure 18-2](#)). Press **Return** to update the **Parent Directory** field.
- e. Select one or more fields to indicate you have auxiliary files you want included in the block.

- f. Click **Next** if you have auxiliary files, otherwise click the **Finish** button. The Custom Block Wizard will create a new directory to contain any custom block files.
2. Add your custom block to a custom palette file (a text file with a `.pal` extension):

Using the syntax discussed in [Section 18.1.2 on page 18-3](#), define a single line in the custom block palette file for each new block. For example the following string should be on one line:

```
blockdirectory = "./blocks/csi2" icon = "./blocks/blockImages/csi2.xpm"
```

3. Load the custom palette file into the palette browser.

EXAMPLE 18-5: Simple Custom Block Example

For your first custom block, create a custom gain block.

1. Create a new SuperBlock. Add a gain block to the SuperBlock. Name the block `customgain` and make the gain value 100. Don't worry about connections.
2. Select the gain block, then select Edit→New Custom Block.

The Custom Block Wizard appears. Deselect any of the check boxes currently selected in files area. Enter **mygain** for the Block Name. Enter a valid name for the Parent Directory (for example, `./blocks`). Click **Finish**. The new custom block directories will be created within the Parent Directory, for example,

```
current-working-directory/blocks/mygain
```

3. Modify or create the palette file `example.pal`. Add the following lines to the file:

```
blockdirectory="./blocks/mygain"
blockdirectory="./blocks/mygain" title = "gain100"
```

Start the palette browser, and load the file `example.pal` from examples [18-1](#) and [18-2](#). Your new custom block file appears under the label **example**.

The custom block appears twice in the palette: the first block displays the name of the directory, because no title was specified. The second block displays the string specified with the title keyword.

4. Drag the custom block off the palette into the SystemBuild editor window.
-

18.2.4 Creating More Sophisticated Custom Blocks

If the custom block has other files (such as C files, startup files, bitmap files, custom help files, etc.) associated with it, the Custom Block Wizard requires additional information about the files.

Step 1: Create a Custom Block Using the Custom Block Wizard

1. When the Custom Block Wizard appears, a set of toggle buttons are displayed in the lower section of the window. Specify the name and directory for your custom block, as outlined in the previous section. Click **Next >** until the appropriate page appears. The following types of files can be attached to a custom block:
 - **Dependent File:** The files that are associated with a User Code Block (UCB) or a BlockScript block etc. can be specified in this section. You can specify the platform dependency by selecting the **Platform** combination box to specify the appropriate platform for the selected files.
 - **Custom Help File:** HTML files that form the help for the custom block. Since these files are platform-independent, the **Platform** combination box will not be available while specifying these files. Help can be a file hierarchy, complete with graphic files. The name of the top-level help file should be `block_name.html`, where `block_name` is the name of the block. If the name of the top-level file is not same as the block name, then its name must be specified in the palette file as described in “[Step 2: Add a Custom Block to a Custom Palette File.](#)”
 - **Icon File:** Icon displayed for the custom block while it is in the editor (custom icons do not appear in the Editor). You cannot use the **Platform** combo box for icon files. Files with the extension `.BMP` will be used in the Windows environment, while files with the extension `.XPM` will be used on all UNIX platforms.
 - **Parameter Set:** One or more files containing values pertinent to the block. Parameter sets allow a user to select different prespecified values for the block. A custom block can have multiple parameter sets.
 - **MathScript File:** An optional MathScript Function (MSF) file having the same name as the custom block. For example, if the custom block is called “engine”, then this MSF file must be called “engine.msfc”. This file can be used to further customize the custom block during instantiation. See [Sys-bldEvent on page 18-15](#).

2. On the appropriate file list page, click the **Browse...** button. This will bring up the file dialog and will allow you to select the file that will be added to the list. Once you select the file to be added to the list and press **OK**, the full path name of the file will be displayed in the Custom Block Wizard. Click the **Add** button to add this selected file to the list. Alternatively, type in the full path to the file directly in the Custom Block Wizard.

To delete a file from the list single click on the file entry, then click the **Delete** button.

3. Click the **Next >** button to add other files. Click the **Finish** button, or, click the **Cancel** button to cancel the process.

Step 2: Add a Custom Block to a Custom Palette File

```
blockdirectory = BlockName [[keyword=value] [keyword=value]...]
```

The valid keywords are:

- title** Specifies the title of the block to be displayed in the editor.
- block** Specifies the block file to be loaded for the block object to be displayed. This is necessary if you have multiple block files in the custom block directory.
- help** Specifies the help file to be used for displaying help for the custom block. The default name of the help file is `block_name.html`. A file specified using this keyword takes precedence over a default help file.
- icon** Specifies the bitmap shown for the block while it is on the palette. This file can be platform dependent. If the filename is specified as `name.platform`, then the name will automatically be expanded to `name.bmp` on Windows platforms, and `name.xpm` on UNIX platforms.

Some examples for the syntax of the block objects are:

```
ISI_gain
../custGain
/homes/pal/custGain title="Gain"
c:\users\user1\palettes\custGain title="Gain"
/homes/pal/blkdir block="sig_1.blk" icon="sig_1.platform"
```

Remember to use a relative path if portability is required.

Step 3: Load the Custom Palette File

Load the custom palette file into the palette browser.

EXAMPLE 18-6: Custom Block Example with Startup File

We'll now extend the simple gain custom block to include a startup file. The startup file and the custom block must have the same root name. The startup file will simply change the color of the block based on user input.

1. In your local directory, create a text file named file `startgain.msf` that contains the following lines into the file:

```
function [result] = startgain(sbname, blkid)
    color = getchoice("Choose a color:", ["red", "green"]);
    modifyblock blkid, {color = color};
endfunction
```

This function will raise a dialog to prompt you for a block color whenever the “startgain” custom block is dragged off the palette.

2. Create a test SuperBlock. Add a gain block to the SuperBlock. Name it “startgain” and set the value of the gain to 100.
3. Select the gain block. Select Edit→New Custom Block...

The Custom Block Wizard appears. Make sure Startup (MSF) is selected in the first screen. Deselect any other options. Enter “startgain” for the Block Name. Enter a valid name for the parent directory (for example, `./blocks`.)

Click on **Next**. In the Startup Files dialog, Enter the path to the file `startgain.msf`. Select **Add** to complete the selection.

Click **Finish**. The new custom block directories will be created within the parent directory. The file `startgain.msf` is also copied to the custom block directory.

4. Edit an existing palette file to include helpgain. Start the palette browser, and load the file `example.pal`. Your new custom block file will be under the label **example**.
 5. Drag the custom block off the palette into the SystemBuild editor window. A small Xmath dialog will appear. Enter one of the options, and hit **OK**. The gain block with the appropriate color is drawn in the editor window.
-

EXAMPLE 18-7: Simple Custom Block with Help

We'll now extend the simple gain custom block to include a help file. Like the startup file, the help file must have the same root name as the block.

1. In your local directory, edit the file `helpgain.html`. Enter the following lines into the file:

```
<pre>
Hello world. This is help for my helpgain block.
</pre>
```

Save the file. Note, the extension should be `.html` (`.htm` is not accepted on all systems).

2. Create a new SuperBlock. Add a gain block to the SuperBlock and give it some unique properties: `blockname = "helpgain"`, `gain = 99`.
3. Select the gain block. Select Edit→New Custom Block...

The Custom Block Wizard appears (it might be hidden behind other windows.) Make sure **Custom Help** is selected in the first screen. Deselect any other options. Enter "helpgain" for the block name. enter a valid name for the parent directory.

Click on **Next**. The dialog prompting for help file name appears. Enter the path to the file `helpgain.html` (or use the browser).

Click **Add**, then click **Finish**. The new custom block directories will be created within the parent directory. Your simple help file will be copied to the `help` subdirectory in the custom block directory.

4. Modify or create the palette file `example.pal`. Add the line to the file:

```
blockdirectory="path-to-your-custom-block"
```

Start the palette browser, and load the file `example.pal`. Your new custom block appears under the label **example**.

5. Drag the custom block off the palette into the SystemBuild editor. Open the block dialog for the custom block. Click on **Help**. Your HTML help text will be displayed in a local help window.
-

18.3 Supporting Commands and Functions

Custom blocks can also be manipulated with commands and functions issued from the Xmath command area.

18.3.1 SystemBuild Access Support

The following SBA commands accommodate custom blocks and palettes. For a complete description of each command, see the online help.

- CreateBlock** Handles the instantiation of custom blocks. The `palette` keyword is used to specify a palette.
- ModifyBlock** Handles the modification of custom blocks. The `customhelp` keyword allows you to specify the name of a help file.
- QueryBlock** This function can be used to query custom blocks.

18.3.2 SystemBuild Utilities (sysbldEvent, sysbldRelease)

SysbldEvent

```
answer = sysbldEvent (event, mode, {MSFName});
```

`sysbldevent` lets you register Xmath MSFs to be called for specific SystemBuild events. The MSF called can replace or supplement an action normally performed by SystemBuild. `SysbldEvent` returns 0 for successful completion and 1 for failure.

`SysbldEvent` has the following characteristics:

- Only one MSF at a time can be associated with an event.
- Once you enable a SystemBuild event, it is called each time that event is invoked.

- `event` An event must be specified; the value can be either `BlockOpen` or `Navigate`.
- `mode` Assign a value to mode; 1 tells SystemBuild to send Xmath the event specified by the `event` parameter; 0 disables the event. If you are enabling an event and you have not specified a function to handle the event, then the `MSFName` parameter is required.

MSFName A string containing the name of an Xmath MSF that handles the specified event.

The Xmath MSF must accept two parameters: `block_id` and `event`, where `block_id` is the block identification number of the block selected by the editor, and `event` is the same event specified by the calling `SysbldEvent` command.

The return value from this MSF is 0 or 1, where 0 has SystemBuild continue by handling the event which was passed to this MSF and 1 has SystemBuild continue by ignoring the event.

The following example shows the `SysbldEvent` parameters.

```
SysbldEvent ("BlockOpen",mode=1,MSFName="UserBlockOpen");
```

When you try to open a block dialog from the editor, it calls Xmath, which calls the MSF as follows:

```
UserBlockOpen("BlockOpen", block_id)
```

In the above example, `block_id` is the number of the block selected in the editor. In the MSF `UserBlockOpen`, the user can use SBA to query or modify the model. Then, if the user returns 0 from the MSF, the editor displays the dialog. If the user returns a 1, the editor continues without displaying the dialog.

As soon as the SystemBuild Editor issues the event to be handled to Xmath, all user actions in the editor are disabled. When the MSF returns, the editor is enabled. Disabling the UI ensures that no conflict exists between the user input and any editor-interactive commands, such as SBA, issued by the MSF.

SysbldRelease

During a `SysbldEvent` callback you are prevented from interacting with the editor. The `SysbldRelease` function is used to release the editor when the callback no longer needs it. `SysbldRelease` can also be used if the user interface becomes locked and does not release during `SysbldEvent` operations.

Index

A

actiming 7-27
Adams-Bashforth-Moulton integration 11-21
addition and subtraction 15-6
advanced load 2-3
algebraic loop 7-13, 11-10
 and trim function 9-16
 detected 7-15
 initial conditions for 7-16
 integration algorithms for 7-14
analyze function 7-10
 Editor tool 7-12
 from SystemBuild editor tool 7-12
 items displayed 7-10
 sbInfo 7-10
 with implicit outputs 11-4
animation.cfg
 BUILD_LOAD_PICTURE 16-24
 ICON_DATA_FILE 16-24
 ICON_SOURCE_FILE 16-24
 PROCESS_PICTURES 16-24
 SYSTEM_BUILD_RTF_FILE 16-24
animation.cfg file, IA 16-21
asynchronous trigger SuperBlock 5-15
AutoSave capability 2-9
auxiliary constraints 11-3

B

Background Procedure SuperBlock 5-6
background simulation 7-4
basic time period 9-7
blocks
 Code tab 4-7
 comment editor 6-5
 create (Palette Browser) 4-1
 creating 4-1
 defining 4-2
 described in online help 4-1
 icon colors 4-12
 icon types 4-12
 ID determined by location 13-11
 ID number 4-6
 inputs 4-5
 name 4-5
 outputs
 labels 4-8
 names 4-8
 type name, block dialogs 4-9
 parameters, specifying 4-7
 special 4-40
 special behaviors 4-40
 states field 4-6
 supported in RVE 8-21
 tabs 4-8
 unsupported in RVE 8-22
 using matrix editor 4-13

- BlockScript 12-1
 - Abort function 12-15
 - Abs function 12-15
 - ABSTOL environment variable 12-19
 - Acos function 12-15
 - Asin function 12-15
 - assignment statements and expressions 12-9
 - Atan function 12-16
 - Atan2 function 12-16
 - Bclear function 12-16
 - BitAnd(a,b) function 12-16
 - BitLshift function 12-16
 - BitNot(a) function 12-16
 - BitOr(a,b) function 12-16
 - BitRshift function 12-16
 - Bound function 12-16
 - Bset function 12-16
 - Btest function 12-16
 - Btoggle function 12-16
 - Ceiling function 12-17
 - colon wildcard 12-7
 - compiling blocks 12-20
 - Cos function 12-17
 - Cosh function 12-18
 - datatypes 12-5
 - datotyping, implied 12-8
 - environment variables 12-19
 - EPSILON environment variable 12-19
 - examples 12-22
 - exit statement 12-14
 - Exp(a) function 12-16
 - float function 12-15
 - Floor function 12-17
 - for loop 12-12
 - IF clause 12-13
 - INIT environment variable 12-19
 - Integer function 12-15
 - intrinsic functions 12-15
 - iterate statement 12-15
 - Log(a) and Log10(a) functions 12-16
 - looping constructs 12-12
 - Max(a,b) and Min(a,b) functions 12-17
 - Mod(a,b) function 12-17
 - Nrand function 12-18
 - operators and precedence 12-9
 - OURand function 12-18
 - OUTPUT environment variable 12-19
 - PI environment variable 12-19
 - Quad function 12-17
 - RELTOL environment variable 12-19
 - Round function 12-17
 - select clause 12-13
 - Sign function 12-17
 - Sin function 12-17
 - Sinh function 12-18
 - Sqrt function 12-18
 - STATE environment variable 12-19
 - Swap function 12-18
 - Tan function 12-17
 - Tanh Function 12-18
 - TIME environment variable 12-19
 - Trg function 12-18
 - Truncate function 12-17
 - TSAMP environment variable 12-20
 - TSTART environment variable 12-20
 - Urand function 12-18
 - variables 12-3
 - Var.columns 12-15
 - Var.rows 12-15
 - Var.size functions 12-15
 - Var.type function 12-15
 - while loop 12-12

BOOLEAN datatype 4-21
 Break block 4-41
 build 2-1
 building IA custom icons 16-25

C

catalog browser 2-1
 catalog view 2-5, 2-7, 2-12
 contents view 2-6, 2-7
 drag copies 2-12
 sort 2-7
 creating SuperBlock reference 3-12
 drag and drop 2-12
 force update 2-13
 modifying catalog 2-15
 navigating 2-5
 quick access menu 2-10
 save selected 2-8, 2-9
 tools 2-14
 transform SuperBlock tool 5-26
 using advanced load 2-3
 catalog, browsing 2-5
 classical analysis tools 10-1
 color assignment (UNIX) 6-8
 colors, block icons 4-12
 comment editor 6-5
 compilers 14-46
 component 2-5, 17-1
 creating 17-9, 17-12
 encrypted 17-4
 encryption 17-16
 exported variables 17-11
 interface 17-2
 licensing 17-4
 mapping parameters 17-21
 modifying 17-13
 open 17-4
 parameter mapping 17-11
 parameters 17-7
 PSET 17-14
 references 17-3
 restrictions 17-9
 SBLIBS 17-16
 scope 17-10
 unmake 17-14
 %Variables 17-10
 computational attributes 9-1
 Condition block 4-40, 5-3
 can contain Standard procedure 5-4
 procedures referenced from 5-3
 connection editor 4-31
 Add button 4-31
 Cancel button 4-33
 Del button 4-32
 Done button 4-33
 routing 4-33
 connections 4-28
 display 4-33
 rules 4-29
 constrained DAEs 14-3, 14-6
 constraints, auxiliary 11-3
 constraints, in implicit models 11-2
 constraints, required 11-3
 continuous subsystem 7-23
 linearizing 9-2
 TimeDelay block, linearizing 9-5
 continuous SuperBlocks 5-1
 control.sog, IA control panel file 16-3
 copy
 SuperBlock (copy and paste) 3-14
 SuperBlock (rename) 3-16
 SuperBlock (via SuperBlock properties) 3-14
 creatertf 7-18

- createusertype 15-42
- Ctrl-p 7-5
- custom block 18-6
 - add to custom palette file 18-10, 18-12
 - creating 18-8, 18-11
 - SBA support 18-15
 - using Custom Block Wizard 18-9, 18-11
- custom dialog, component 17-23
- custom icons
 - adding to block diagram 16-2
 - building 16-25
 - keywords 16-24
- custom palette 18-1

D

- DASSL 7-14, 11-20, 11-21
- DASSL integration 14-4
- DataPathSwitch block 4-41
- datastores 5-20
 - in catalog 2-5
 - timing 5-20
 - using 5-11
- datatypes 3-8, 4-8, 4-19, 4-20, 4-21
 - BlockScript 12-5
 - classes of data 4-22
 - floating point 4-21
 - how specified 4-22
 - integer 4-21
 - logical 4-21
 - RT_BOOLEAN 4-21
 - RT_FLOAT 4-21
 - rules 4-25
 - type-checking issues 4-24
- deleteusertype 15-42
- Differential Algebraic Equations (DAEs) 11-15, 14-3

- differential equation solvers 7-18
- direct terms 7-14
- direct terms with UCBs 14-10
- discrete subsystems 5-1
 - linearizing 9-4
 - types 5-1
- display icons 8-11
- drag and drop, to attach an IA icon to a block 16-2

E

- Editor
 - block label names 4-18
 - changing font size 4-34
 - classical analysis tools 10-1
 - color assignment (UNIX) 6-8
 - connection menu 4-30
 - connections 4-30
 - creating blocks 4-1
 - creating SuperBlock reference 3-12
 - custom menus 6-5
 - default window position 6-9
 - force catalog update 2-13
 - modifying block diagram 4-33
 - resizing and repositioning the display 6-9
 - scope 2-11
 - shortcuts 4-34
 - SuperBlock transform tool 5-26
 - tool bar 4-34
- EDIT_COMMENT environment variable 6-4
- enabled periodic subsystem 7-23
- exact linearization 9-5
- exiting SystemBuild 2-1
- explicit models 11-2
- explicit UCB 14-2

F

- File SuperBlocks 5-10
- finite difference linearization 9-5
- fixed-point 15-6
 - arithmetic 15-1
 - addition and subtraction 15-6
 - division 15-8
 - comparing with floating point 15-16
 - compatible blocks 15-1
 - compatible blocks, with datatypes 15-23
 - datatype conversion 15-6
 - datatype rules 15-23
 - Gain block radix position 15-32
 - linearization 15-44
 - multiplication 15-8
 - number representations 15-3
 - overflow 15-10, 15-13
 - relational operations 15-9
 - simulation, ITypes 15-26, 15-28
 - with simout 15-44
 - 32-bit operations 15-32
- fixpt 4-21
- floating point datatype 4-21
- Floating Point Greatest Common Divisor 7-24
- FTP to ISI Technical Support xxvi
- functional blocks 3-2
- FuzzyLogic block, linearizing 9-6

G

- Gear's method 11-30
- Gear's Method (GEARS) 11-30

H

- hierarchy of SuperBlocks 3-2

I

- IA 16-25
 - Builder
 - icon source code syntax 16-25
 - palette files 16-2
 - compiler 16-25
 - icon source file 16-8
- icon definition
 - ANIMATION_GRAPHICS 16-9
 - ANIMATION_POINTER 16-8
 - BACKGROUND_SECTION 16-9
 - DO 16-12
 - DRAW_ARC 16-15
 - DRAW_LINE 16-15
 - DRAW_RECT 16-13
 - DRAW_TEXT 16-14
 - ERASE_VALUE 16-14, 16-15
 - FORM_DEFINITION 16-9
 - GENERAL_LINE 16-16
 - ICON_HEIGHT 16-8
 - ICON_PRIVILEGE 16-8
 - ICON_WIDTH 16-8
 - IF/THEN 16-12
 - INITIALIZATION_SECTION 16-9
 - INTEGER_VARIABLE 16-8
 - MATH_FUNCTION 16-12
 - OUTPUT_POINTER 16-8
 - PALETTE_DEFINITION 16-9
 - REAL_VARIABLE 16-8
 - RELATIVE_POSITION_LINE 16-16
 - ROTATE_LINE 16-15
 - STATE_POINTER 16-8
 - STATIC_GRAPHICS section 16-9
 - STRING_VARIABLE 16-8
 - types 16-10
- icon source file, IA 16-8
- IconScript keywords 16-13

ICON_SOURCE_FILE keyword definition
 16-24
 ID number, block 4-6
 IfThenElse block 4-40
 implicit models 11-2
 implicit outputs 11-4
 implicit states 11-3
 implicit UCB 14-2, 14-3
 initial condition transformations 5-28
 Inline Procedure SuperBlocks 5-5
 input
 datatypes, how set 4-22
 names, block dialog 4-7
 signals, block dialog 4-8
 vector, u 7-3
 integer datatype 4-21
 integration algorithm 11-15
 Adams-Bashforth-Moulton 11-32
 DASSL
 estimating err 11-32
 infinity norm 11-32
 solving DAE 11-25
 default 11-15
 Euler 11-16
 fixed-step Kutta-Merson 11-18
 GEARS 11-30
 ODASSL 11-24
 estimating err 11-32
 solving DAE 11-25
 QuickSim 11-22
 Runge-Kutta 11-17, 11-18
 selecting 7-18
 stiff system solver (DASSL) 11-19
 operating point 11-20
 variable-step Kutta-Merson 11-19
 Interactive Animation (IA) 16-1
 Interactive Root Locus tool 10-12

interactive simulation. See ISIM.
 Interrupt Procedure SuperBlock 5-7
 ISIM 8-1
 and RVE 8-15
 color settings 6-9
 compared to IA 8-1
 Control Panel 8-4
 debugging 8-5
 default icon palette 8-3
 default window position 6-9
 example 8-12
 icon
 Digital Monitoring LED 8-10
 Multiple Line Graph 8-10
 Numeric Display 8-10
 Pushbutton Switch 8-10
 Slide Output Controller 8-11
 standard 8-7
 Strip Chart 8-9
 Text 8-11
 update times 8-5
 keywords 8-6
 monitoring block outputs 8-5
 pause 8-4
 resume 8-4
 run in background 8-19
 scaling aid blocks 15-45
 terminating 7-5
 view output labels 8-5
 window 8-4
 isim.sog, icon file 16-3
 IS_Jacobian 14-19
 IUSR01 template 14-30

K

Kalman-Bertram State-Space Method
9-1

L

labels

- creating 4-14
- defined at source 4-15
- initializing matrix 4-17
- matricizing 4-16
- propagating 4-11
- shortcuts for editing 4-18
- showing 4-11
- SuperBlock external inputs 4-15
- vectoring 4-16

Lagrange Multipliers 14-5

lin 7-16

linearization

- at a non-zero time 9-6
- continuous time delay 9-5
- exact vs. finite difference 9-5
- FuzzyLogic Block 9-6
- implicit form 9-2
- in fixed-point mode 15-44
- Kalman-Bertram state-space method 9-7
- lin 7-16
- multirate 9-7
- Procedure SuperBlocks 5-4, 9-6, 10-1
- resettable integrator 9-6
- single-rate 9-2
- single-rate continuous 9-2
- single-rate discrete 9-4
- single-rate explicit 9-3
- single-rate implicit 9-3
- State Transition Diagram 9-5

STDs 9-5

UCB 9-6

linksim 14-44

listusertype 15-42

LOAD 2-2

load, advanced 2-3

logical datatype 4-21

M

Macro Procedure SuperBlock 5-4

matrix editor, SystemBuild 4-13

menus, custom 6-5

MIL-STD-2167A 6-5

MinMax

- dataset, save 15-38

- display 15-38

- logging 15-37, 15-38

- restrictions 15-38

modifyusertype 15-42

multirate linearization 9-7

N

name

- block 4-5

- SuperBlock 3-4

names

- creating 4-14

- initializing matrix 4-17

- matricizing 4-16

- vectoring 4-16

namespace 17-1

O

ODAE 11-15

ODASSL 7-14, 11-24

ODASSL with implicit UCB 14-4

ODE 11-15

open SuperBlock 2-11
 Open-Loop Frequency Response tool
 10-3
 operating point
 DASSL 14-9
 Jacobian matrix computation 14-27
 ODASSL 14-9
 output datatypes, how set 4-22
 Outputs Tab View 4-22
 Overdetermined Stiff System Solver
 (ODASSL) 11-24

P

palette browser, using 4-1
 palette file 18-2
 block directory 18-5
 built-in 18-4
 default 18-3
 default palette 18-5
 multiple level 18-2
 syntax 18-3
 palette file, IA 16-2, 16-24
 palette, custom 18-1
 PALETTE_PATH 18-5
 panning the display 4-34
 Parameter Root Locus tool 10-13
 parameter set. See PSET.
 parameter variable scoping 7-6
 parameterized variables 7-5
 Point-to-Point Frequency Response tool
 10-9
 printer settings 6-4
 procedure subsystem 7-23
 Procedure SuperBlocks 5-3
 Background 5-6
 in linearization 9-6
 Inline 5-5
 Interrupt 5-7

linearization 5-4, 10-1
 Macro 5-4
 standard 5-4
 Startup 5-6

Processor Group ID 7-23
 propagate labels 4-11
 PSET 17-14
 example 17-19
 Psets_AddToList 17-15
 Psets_Load 17-15
 Psets_Save 17-15

R

Read From Variable Block 5-4
 reference, component 17-6
 rename 3-16
 rename SuperBlock 3-15
 repositioning the display 6-9
 required constraints 11-3
 resettable integrator, linearization 9-6
 Root Locus tool, interactive 10-11
 RTF (Real-Time File), creating 7-18
 RT_BOOLEAN datatype 4-21
 RT_FLOAT datatypes 4-21
 RT_INTEGER datatype 4-21
 running simulations 7-1
 Run-time Variable Editor. See RVE.
 RVE 8-15
 commands and functions 8-20
 invoking 8-17
 supported blocks 8-21
 unsupported blocks 8-22
 variable browser 8-17

S

- SAVE 2-8
 - AutoSave feature 2-9
 - usertype keyword 15-43
- SBA 13-1
 - command syntax 13-3
 - error handling 13-8
 - function syntax 13-3
 - keyword formats 13-9
 - sample scripts 13-6
 - specifying mimo names 13-9
 - updating Editor display 13-6
- SBLIBS 5-10, 5-11
- sbsim command 7-9
- sb_uext reserved variable 10-3
- scheduler 7-23
 - minor cycle 5-21
- Sequencer block 4-41
- SETSBDEFAULT 7-8
 - autosavefile 2-10
 - autosavetime 2-10
- setsbdefault 11-15
- shortcuts 4-5
- showsbdefault 11-16
- signal 4-15
- sim function 7-2
 - bg 7-4
 - function syntax 7-2
 - iahold 8-7
 - initmode 11-10
 - interact 8-6
 - PDM outputs 7-3
 - sbsim 7-9
 - sbview 8-6
 - ucbcode loc 14-45
 - vars 7-6
- SIMAPI 14-14
 - debug example 14-28
 - debugging functions 14-20
 - debugging simulation 14-18
 - debugging with 14-18
 - source files 14-14
 - UCB reference information 14-15
 - variable access 14-16
- simAPI.h 14-14
- SIMAPI_FlushVars 14-18
- SIMAPI_GetBlockId 14-15
- SIMAPI_GetBlockInputLabel 14-15
- SIMAPI_GetBlockInputType 14-15
- SIMAPI_GetBlockName 14-15
- SIMAPI_GetBlockOutputLabel 14-15
- SIMAPI_GetBlockOutputType 14-15
- SIMAPI_GetDefaultOutputLabel 14-15
- SIMAPI_GetExternalInputDimension 14-19
- SIMAPI_GetExternalInputName 14-19
- SIMAPI_GetExternalInputValue 14-19
- SIMAPI_GetExternalOutputDimension 14-19
- SIMAPI_GetExternalOutputName 14-19
- SIMAPI_GetExternalOutputValue 14-19
- SIMAPI_GetImplicitOutputDimension 14-19
- SIMAPI_GetImplicitOutputName 14-19
- SIMAPI_GetImplicitOutputValue 14-19
- SIMAPI_GetImplicitSolverJacobian 14-19
- SIMAPI_GetNumVars 14-16
- SIMAPI_GetOperatingPointJacobian 14-19
- SIMAPI_GetSBName 14-15
- SIMAPI_GetSimStatus 14-20
- SIMAPI_GetStateDerivativeValue 14-19
- SIMAPI_GetStateDimension 14-19
- SIMAPI_GetStateName 14-19

- SIMAPI_GetStateValue 14-19
- SIMAPI_GetUCBBlockInfo 14-15
- SIMAPI_GetVarData 14-17
- SIMAPI_GetVarDatatypeName 14-17
- SIMAPI_GetVarDimension 14-17
- SIMAPI_GetVarIndexByName 14-17
- SIMAPI_GetVarName 14-16
- SIMAPI_GetVarPartition 14-17
- SIMAPI_GetVarStorageSize 14-17
- SIMAPI_GetVarUserTypeName 14-17
- SIMAPI_InitializeUserDebug 14-19
- SIMAPI_IsVarEditable 14-17
- SIMAPI_PutVarData 14-17
- SIMAPI_ResetVar 14-18
- SIMAPI_TerminateUserDebug 14-19
- simout function 7-17, 14-4
- simout function, with fixed-point 15-44
- simulation 7-1
 - abort 7-5
 - background 7-4
 - DataStores in 5-12
 - errors 7-21
 - extracting values (simout) 7-17
 - fixed-point intermediate types 15-25
 - from Editor 7-1
 - from the OS command line 7-8, 7-9
 - initial conditions 11-11
 - initialization mode 11-10
 - input vector (u) 7-3
 - interactive 8-1
 - maximum integration step size 11-33
 - operating point 11-10
 - changing 11-11
 - continuous subsystem 11-10
 - discrete subsystem 11-10
 - scheduler 7-23
 - See sim function.
 - stopping background job 7-4
 - termination 7-5
 - time vector (t) 7-3
 - timing properties 5-12
 - with ISIM 8-14
- simulation API. See SIMAPI
- single-rate systems
 - linearizing 9-2
- source palette files, IA 16-2
- standalone simulation (sbsim) 7-9
- starting SystemBuild 2-1
- Startup Procedure SuperBlock 5-6
- startup.pal 18-5
- state event
 - continuous system 11-42
 - UCB 14-13
 - UCB (continuous) 11-44
 - ZeroCrossing block 11-43
- state events, UCB 14-13
- State Transition Diagrams 2-5
 - linearizing 9-5
- stdwrt 14-48
- Stiff System Solver
 - DASSL 11-19
 - ODASSL 11-24
- Stop block 4-41
- subsystem
 - continuous 5-1
 - discrete 5-1
 - enabled periodic 7-26
 - free-running periodic 7-26
 - trigger 5-12, 7-26
 - asynchronous 5-19
 - priority 5-19
- priorities 5-19
- procedure 7-23

- processor Group ID [7-24](#)
- pseudo-rate sample interval [7-24](#)
- SuperBlock [2-4](#)
 - Attributes tab [3-7](#)
 - Block [3-9](#)
 - Code tab [3-7](#)
 - Comment tab [3-9](#)
 - continuous [5-1](#)
 - copy, paste, and rename [2-15](#)
 - create [3-3](#)
 - create new [2-13](#), [3-4](#)
 - creating from catalog browser [3-4](#)
 - creating from Editor [3-5](#)
 - Document tab [3-9](#)
 - documentation generation [6-5](#)
 - Editor coordinates [13-10](#)
 - expand [3-5](#)
 - Inputs tab [3-8](#)
 - instances [3-3](#)
 - i/o datatypes [4-22](#)
 - label names [4-15](#)
 - editing [4-18](#)
 - vectoring [4-16](#)
 - label, propagate [3-10](#)
 - making [3-5](#)
 - name [3-4](#)
 - new [2-13](#)
 - open [2-11](#)
 - open in Editor [2-11](#)
 - output labels [3-9](#)
 - output names [3-9](#)
 - Procedure [5-3](#)
 - Background [5-5](#)
 - execution sequence [5-8](#)
 - Inline [5-5](#)
 - Interrupt [5-7](#)
 - Macro [5-4](#)

- Standard [5-4](#)
- Startup [5-6](#)
- properties [3-3](#), [3-6](#), [5-27](#)
- reference [3-1](#), [3-3](#), [3-12](#)
- rename [3-15](#)
- top-level [3-1](#), [3-3](#)
- transform [2-14](#)
- transformation, undoing [5-28](#)
- transforming [2-14](#), [5-24](#)
- triggered [5-2](#)
- SuperBlock Editor. See [Editor](#)
- SuperBlock reference
 - in catalog browser [3-2](#)
 - in Editor [3-12](#)
- support
 - contacting by e-mail [xxvi](#)
 - contacting by phone [xxvi](#)
 - filing a problem report [xxvi](#)
 - sending files via FTP [xxvi](#)
- syntax rules, IA icons source code [16-25](#)
- Sysbld resource file [6-8](#)
- SysbldEvent [18-15](#)
- SysbldRelease [18-16](#)
- SystemBuild
 - exiting [2-2](#)
 - launching [2-1](#)
 - resource file [6-8](#)
- SystemBuild Access. See [SBA](#).

T

- Technical Support (see [support](#)) [xxvi](#)
- text editor program, changing the default [6-4](#)
- text editor, setting default [6-4](#)
- Time Response tool [10-7](#)
- time vector (t) [7-3](#)
- timing attributes, UCB [14-23](#)

- top-level SuperBlock [3-3](#)
- transforming
 - gain block [5-25](#)
 - integrators [5-25](#)
 - PID controller [5-25](#)
- triggered subsystem [7-23](#)
- trim [9-12](#), [9-14](#)
 - for systems with algebraic loop [9-16](#)
 - free integrators [9-15](#)
- trim function [9-11](#)
- typecheck with BlockScript block [12-9](#)
- TypeConversion block [4-23](#)

U

- UCB [14-1](#)
 - code location [14-45](#)
 - compile and link [14-46](#)
 - compilers, supported [14-46](#)
 - continuous [11-44](#)
 - debug example [14-28](#)
 - debugging [14-18](#)
 - debugging user code [14-47](#)
 - direct terms [14-10](#)
 - error messages, how to generate [14-48](#)
 - execution order [14-23](#)
 - explicit [14-2](#)
 - implicit [14-2](#), [14-3](#)
 - simulation [11-13](#)
 - with sim, lin, or simout [14-8](#)
 - in SystemBuild vs. AutoCode [14-1](#)
 - initialization [14-24](#)
 - interface [14-2](#)
 - linearization [9-6](#)
 - makefile [14-45](#)

- modes
 - EVENT [14-31](#)
 - INIT [14-30](#)
 - LAST [14-32](#)
 - LIN [14-31](#)
 - MONIT [14-31](#)
 - OUTPUT [14-31](#)
 - STATE [14-30](#)
- multi-use [14-43](#)
- numerical integration algorithm [14-25](#)
- programming considerations [14-43](#)
- see variable interface UCB [14-37](#)
- specifying source code [14-44](#)
- state event [11-44](#)
- state events [14-13](#)
- timing attributes [14-23](#)
- variable names in [14-43](#)
- unconstrained DAEs [14-5](#)
- update Catalog Browser [2-13](#)
- User Defined Types. See usertype.
- User Parameters [4-10](#)
- UserCode Blocks. see UCB
- UserCode function [14-2](#)
 - explicit arguments [14-32](#)
 - IINFO vector [14-33](#)
 - mode parameters [14-35](#)
 - RINFO vector [14-34](#)
- usertype [15-2](#)
 - create [15-40](#)
 - delete [15-41](#)
 - editor [15-40](#)
 - modify [15-41](#)
- user.ini [6-1](#)
- USR01 template [14-30](#)

V

variable

parameterized [7-5](#)scoping [7-6](#)Variable blocks [5-4](#)variable interface UCB [14-37](#)C wrapper required [14-37](#)C wrapper, writing [14-38](#)specifying interface [14-41](#)

variables

component mapping [17-11](#)variable-step integration [11-21](#)Adams-Moulton method [11-21](#),
[11-32](#)Kutta-Merson method [11-31](#)**W**While block [4-41](#)Write To Variable Block [5-4](#)**Symbols**\$SYSBLD/etc directory [16-3](#)%Variables [7-5](#), [7-7](#): wildcard in BlockScript [12-7](#)

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources include the following:
 - **Self-Help Resources**—For immediate answers and solutions, visit our extensive library of technical support resources available in English, Japanese, and Spanish at ni.com/support. These resources are available for most products at no cost to registered users and include software drivers and updates, a KnowledgeBase, product manuals, step-by-step troubleshooting wizards, conformity documentation, example code, tutorials and application notes, instrument drivers, discussion forums, a measurement glossary, and so on.
 - **Assisted Support Options**—Contact NI engineers and other measurement and automation professionals by visiting ni.com/support. Our online system helps you define your question and connects you to the experts by phone, discussion forum, or email.
- **Training**—Visit ni.com/training for self-paced tutorials, videos, and interactive CDs. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.