

MATRIX_x[®]

7.0

SYSTEMBUILD™ USER'S GUIDE



Copyright © 2000 Wind River Systems, Inc.

ALL RIGHTS RESERVED. No part of this publication may be copied in any form, by photocopy, microfilm, retrieval system, or by any other means now known or hereafter invented without the prior written permission of Wind River Systems, Inc.

AutoCode, Embedded Internet, ES*p*, Fast*J*, IxWorks, MATRIX_X, pPRISM, pPRISM+, pSOS, RouterWare, Tornado, VxWorks, *wind*, WindNavigator, Wind River Systems, WinRouter, and Xmath are registered trademarks or service marks of Wind River Systems, Inc.

BetterState, Doctor Design, Embedded Desktop, Envoy, How Smart Things Think, HTMLWorks, MotorWorks, OSEKWorks, Personal JWorks, pSOS+, pSOSim, pSOSystem, SingleStep, SNiFF+, VxD*COM*, VxFusion, VxMP, VxSim, VxVMI, Wind Foundation Classes, WindC++, WindNet, Wind River, WindSurf, and WindView are trademarks or service marks of Wind River Systems, Inc. This is a partial list. For a complete list of Wind River trademarks and service marks, see the following URL:

<http://www.windriver.com/corporate/html/trademark.html>

Use of the above marks without the express written permission of Wind River Systems, Inc. is prohibited. All other trademarks mentioned herein are the property of their respective owners.

Corporate Headquarters

Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

toll free (U.S.): 800/545-WIND
telephone: 510/748-4100
facsimile: 510/749-2010

For additional contact information, please visit the Wind River URL:

<http://www.windriver.com>

For information on how to contact Customer Support, please visit the following URL:

<http://www.windriver.com/support>

Contents

1	Introduction	1
1.1	Product Overview	2
1.2	Starting SystemBuild	3
1.3	Exiting SystemBuild	4
1.4	Running SystemBuild Demos	4
2	Catalog Browser	7
2.1	Loading Data	8
2.1.1	Loading Data from Xmath	8
2.1.2	Loading Data from the Catalog Browser in SystemBuild	8
2.2	Examining Catalog Components in the Catalog View	10
2.2.1	Main Catalog	11
	Model	11
	SuperBlocks	11
	BetterState Charts	12
	State Diagrams	12
	DataStores	12
	Components	13

	Variables	13
	UserTypes	13
2.2.2	Libraries	13
2.2.3	Xmath Partitions	14
2.3	Working with Catalog Views	14
2.4	Saving Data	18
2.4.1	Saving Data from Xmath	18
2.4.2	Saving Data from the Catalog Browser	19
2.4.3	Saving All Data	19
2.4.4	Using AutoSave to Save Data	20
2.5	Working with the Catalog Browser	21
2.5.1	Using the Shortcut Menu in the Catalog Browser	21
2.5.2	Opening a SuperBlock or BetterState Chart in an Editor	22
2.5.3	Creating a New SuperBlock	23
2.5.4	Creating a New BetterState Chart	25
2.5.5	Creating a New UserType	27
2.5.6	Modifying a Catalog	27
2.5.7	Modifying a UserType	28
2.5.8	Dragging SuperBlock and BetterState Chart Icons from the Catalog Browser to the Editor	29
	Dragging SuperBlock Icons from the Catalog View	29
	Dragging SuperBlocks and BetterStateChart Icons from the Contents View	29
	Using the Drag and Drop Feature	29
2.5.9	Updating the Catalog Browser Data from an Editor	30
2.5.10	Using the Tools Menu	31

3	File and Configuration Management	33
3.1	File Management	33
3.1.1	Putting New Items into the Untitled Default File	34
3.1.2	Modifying an Item in a File	36
3.1.3	Deleting an Item in a File	36
3.1.4	Moving an Item from One File to Another	37
3.1.5	Overwriting Items in a File	38
3.1.6	Creating a New File	39
3.1.7	Saving All Files	40
3.1.8	Saving Xmath Partitions	41
3.2	Configuration Management	41
3.2.1	Preparing to Use a CM Tool with SystemBuild	41
	Windows Operating Systems	42
	UNIX Operating Systems	42
3.2.2	Getting Messages from Your CM Tool	43
3.2.3	Connecting to Your Configuration Management Tool	43
3.2.4	Opening a File	43
3.2.5	Putting a File Under Configuration Control from SystemBuild ..	44
3.2.6	Using Other Configuration Management Options	45
3.2.7	Problems with Extension Names in Different Applications	45
3.2.8	Additional Resources	46
4	SuperBlocks	47
4.1	SuperBlock Hierarchies	48
4.2	Creating SuperBlocks	50
4.2.1	Creating a New SuperBlock from the Catalog Browser	50
4.2.2	Making a New SuperBlock from Existing Blocks	50
4.2.3	Creating a Copy of a SuperBlock	51

	Creating a Copy with Copy and Paste	52
	Creating a Copy by Modifying the SuperBlock Properties	53
4.2.4	Defining a SuperBlock's Properties	53
	SuperBlock Properties	53
	Using the SuperBlock Properties Dialog	54
	Attributes Tab	55
	Code Tab	56
	Inputs Tab	56
	Outputs Tab	57
	Document Tab	57
	Comment Tab	58
4.3	Creating a SuperBlock Reference	58
4.3.1	Creating a Reference from the Catalog Browser	59
4.3.2	Creating a Reference from the Editor	61
4.3.3	Defining the Reference SuperBlock's Properties	61
4.4	Renaming SuperBlocks	64
4.4.1	Replacing a SuperBlock with Catalog Browser Rename	64
4.4.2	Renaming SuperBlocks in the Editor	65
4.5	Using File SuperBlocks	66
5	Blocks	69
5.1	Types of Blocks	70
5.1.1	Basic Functional Blocks	70
5.1.2	Conditional Execution (Condition, IfThenElse Blocks)	70
5.1.3	Repetitive Execution (While, Break Blocks)	71
5.1.4	Terminating Execution (Stop Block)	71
5.1.5	Execution Ordering (Sequencer Block)	71
5.2	Creating a Model with Blocks	72
5.3	Assigning the Basic Properties to Your Block	72

5.3.1	Raising the Block Dialog	73
5.3.2	Defining the Basic Properties	74
5.3.3	Using the Common Buttons	74
5.4	Connecting Blocks	75
5.4.1	Connection Rules	75
5.4.2	Creating Connections	76
	Creating a Simple Connection	76
	Using the Connect Menu and the Toolbar Buttons	77
5.4.3	Using the Connection Editor	79
	Creating Connections	79
	Deleting Connections	80
	Altering the Number of External Inputs or Outputs	80
	Displaying Connections	81
	Exiting the Connection Editor	81
5.5	Defining Your Block	82
5.5.1	Block Dialog Overview	82
	Differences Across Platforms	84
	Dialog Navigation and Shortcuts	84
5.5.2	Block Dialog Fields	84
	Parameters	85
	Code	85
	Inputs	86
	Outputs	86
	States	88
	Document	88
	Comment	88
	Icon	89
	Display	89
5.5.3	Entering Matrix Data in Block Dialogs	95
	Invoking the Matrix Editor	95
	Entering a Matrix	96
	Editing a Matrix	97

5.5.4	Specifying Labels and Names	97
	Specifying SuperBlock External Input Labels	98
	Propagating Labels in a Hierarchy	98
	Creating Sequential Names for Vectors and Matrices	99
	Shortcuts for Editing Labels or Names	102
5.5.5	Specifying Data Types	103
	Traditional Data Types	104
	Data Type Checking	105
	Rules for Data Type Usage	108
5.6	Modifying Block Diagram Appearance	112
5.6.1	Automatic and Manual Connection Routing	113
5.6.2	Improving the Appearance of a Cluttered Diagram	114
5.7	Creating a SuperBlock That Uses the Connection Editor Extensively	116
6	SuperBlock Timing and Transformation	123
6.1	Types of SuperBlocks	123
6.1.1	Continuous SuperBlocks	123
6.1.2	Discrete SuperBlocks	124
6.1.3	Triggered SuperBlocks	125
6.1.4	Procedure SuperBlocks	126
	Standard Procedure	127
	Macro Procedure	127
	Inline Procedure	128
	Asynchronous Procedure	128
6.1.5	Effects of Nesting on Enabled and Trigger SuperBlocks	130
6.2	Using DataStores	131
6.3	Simulation Timing Properties	132
6.3.1	Timing of Discrete Subsystems	132
6.3.2	Timing of Trigger Subsystems	134
	At Next Trigger	135

	At Timing Requirement	135
	As Soon As Finished	135
	Asynchronous	135
6.3.3	Example: Using an Asynchronous Triggered SuperBlock	136
6.4	Execution of Procedures During Simulation	140
6.5	AutoCode Timing Properties	142
6.5.1	AutoCode Real-time Application Execution Sequence	142
6.5.2	AutoCode Timing Features Associated with Using DataStores	144
6.5.3	AutoCode and the Asynchronous Triggered System	147
6.6	SuperBlock Transformation	147
6.6.1	Transformation Limitations and Implications	148
	Limitations	148
	Dynamic Blocks	148
	Gain Block	149
	Integrators and PID Controller	149
6.6.2	Transformation Methods	150
	Transformation from the Catalog Browser	150
	Transformation from the SuperBlock Properties Dialog	151
	Initial Condition Transformations	152
6.6.3	Undoing a Transformation	152
7	SystemBuild Access	153
7.1	Overview	153
7.2	SBA Syntax	155
7.2.1	Command Syntax	155
7.2.2	Function Syntax	156
7.2.3	Inputs, Optional Inputs, and Keywords	156
7.3	Basic SBA Tasks	157
7.3.1	Create	157

7.3.2	Query	157
7.3.3	Modify	158
7.3.4	Display	158
7.3.5	Delete	159
7.3.6	Sample Scripts	159
7.4	Using SBA	159
7.4.1	Keyword Ordering	159
7.4.2	Block Parameters	161
7.4.3	Error Handling	161
7.4.4	Input Formats	161
	Typical Input Formats	162
	Multiple Input/Output Specification	162
7.4.5	SuperBlock Editor Coordinate System	163
7.5	Tutorial	164
7.5.1	Building the Predator-Prey Model	164
7.5.2	Simulating the Predator-Prey Model	166
8	Simulator Basics	167
8.1	Dividing Your Model into Subsystems	168
8.1.1	How SystemBuild Divides Your Model Into Subsystems	168
8.1.2	Assigning SuperBlocks to Additional Subsystems	169
8.2	Scheduling Subsystems	170
8.2.1	Scheduling Continuous Subsystems	171
8.2.2	Scheduling Discrete Subsystems	171
	Properties of Discrete Scheduled Subsystems	172
	Matching the Timing of AutoCode for Discrete Systems	173

8.3	Setting Options and Parameters for Your Model	174
8.3.1	Simulation Time Lines, Inputs, and Outputs	174
	Input Time Line	174
	Internal Time Line	175
	Computing External Input Values	175
	Output Time Line	175
8.3.2	Changing Parameters for Repeated Simulations	176
	The Simulation vars Keyword	176
	Parameter Variable Scoping	177
8.3.3	Selecting an Integration Algorithm	179
	Integration Algorithms	179
	Integration Algorithm Recommendations	180
8.4	Analyzing Your Model Prior to Simulation	182
8.4.1	Working with Algebraic Loops	182
8.4.2	Using the analyze() Function	185
	Invoking analyze from the Xmath Command Area	186
	Invoking analyze from the SuperBlock Editor	188
8.4.3	Saving Your Model with the CREATERTF Command	188
8.5	Some Additional Tools	189
8.5.1	Extracting Dynamic State Values with the simout() Function ...	189
8.5.2	Showing and Setting Keyword Default Options	190
8.6	Simulating Your Model	191
8.6.1	SuperBlock Editor Simulation Interface	191
8.6.2	Xmath Command Area Simulation Interface	192
	Sim Function Syntax	192
	Background Simulation	194
8.6.3	Operating System Command Line Simulation Interface	195
8.7	Terminating Your Simulation	196
8.8	Simulation Errors	197

8.8.1	Simulation Software Errors	197
8.8.2	Hardware Errors	198
8.8.3	Operating System Errors	199
9	Interactive Simulation	201
9.1	Interactive Simulation Versus Interactive Animation	202
9.2	Constructing an ISIM Model	203
9.2.1	Using the IA Palettes	203
9.2.2	Building an ISIM Car Model	204
9.3	Running ISIM	207
9.3.1	Keywords and Syntax for Running ISIM	207
	Invoking ISIM	207
	Invoking ISIM for a Specific SuperBlock	207
	Invoking Non-Interactive Simulation with IA Blocks	208
	Pausing ISIM at a Non-Zero Time	208
9.3.2	ISIM Window	208
9.3.3	Special Notes on ISIM	209
9.3.4	Simulating the Car Model	211
9.4	Using the Run-Time Variable Editor	211
9.4.1	RVE and ISIM	213
9.4.2	RVE Commands and Functions	217
9.4.3	RVE-Compatible Blocks	219
10	Linearization	223
10.1	Linearizing Single-Rate Systems About an Initial Operating Point	225
10.1.1	Continuous Systems	225
	Explicit Form	225
	Implicit Form	226
10.1.2	Discrete Systems	227

10.2	Exact Versus Finite Difference Linearization	228
10.3	Special Linear Models	228
10.3.1	Continuous Time Delay	228
10.3.2	State Transition Diagrams	229
10.3.3	FuzzyLogic Block	229
10.3.4	Integrator Block (Resettable)	229
10.3.5	UserCode Blocks	229
10.3.6	Procedure SuperBlocks Referenced from Condition Blocks	229
10.4	Linearizing Single-Rate Systems About a Final Operating Point	230
10.5	Multirate Linearization	230
10.5.1	Kalman-Bertram Method	231
	Interpretation of Multirate lin Results	232
10.5.2	Subsystem Method	234
10.5.3	Linearizing Fixed-Point Blocks	236
11	Operating Point Computation	237
11.1	trim() Syntax	238
11.2	trim() Algorithm	240
11.3	trim() Behavior	242
11.3.1	Stability	242
11.3.2	Free Integrators	242
11.3.3	Algebraic Loops	242
11.4	trim() Examples	243
12	Classical Analysis Tools	245
12.1	Using the Tools	246

12.2	How SystemBuild Proceeds to Analyze Your Model	247
12.3	Open-Loop Frequency Response	248
12.4	Time Response	252
12.5	Point-to-Point Frequency Response	255
12.6	Root Locus	258
12.6.1	Application to a Linear System	260
12.6.2	Application to a Multirate, Nonlinear System	262
12.7	Parameter Root Locus	266
13	Advanced Simulation	271
13.1	Explicit vs. Implicit Models	271
13.1.1	Explicit Models	272
13.1.2	Implicit Models	272
	Constraints	273
	Simulation State	273
	Implicit Outputs	274
	Initialization	274
	Examples	274
13.2	Operating Points	280
13.2.1	Continuous Subsystem	280
13.2.2	Discrete Subsystems	280
13.3	Inserting Initial Conditions	282
13.4	Matrix Blocks in the Simulator	283
13.5	Sim Integration Algorithms	284
13.5.1	Comparing Integration Algorithms	285

13.5.2	Overview of the Algorithms	286
	Euler Integration Method	286
	Second Order Runge-Kutta (Modified Euler) Method	287
	Fourth-Order Runge-Kutta Method	288
	Fixed-Step Kutta-Merson Method	288
	Variable-Step Kutta-Merson Method	289
	Stiff System Solver	290
	Variable-Step Adams-Bashforth-Moulton Method	292
	QuickSim Method	292
	Over-determined Differential Algebraic System Solver	294
	Gear's Method	301
13.5.3	Absolute and Relative Tolerances	302
	Variable-Step Kutta-Merson Method	303
	Stiff System Solvers (DASSL and ODASSL)	303
	Variable-Step Adams-Bashforth-Moulton Method	304
13.5.4	Computing the Maximum Integration Step size in Variable-Step Integration Algorithms	305
13.5.5	Sample Simulation	305
13.6	State Events	313
13.6.1	ZeroCrossing Block	314
	Example Using a Sinusoid Signal	315
	Example Using a Bouncing Ball	316
13.6.2	Continuous UserCode Blocks	318
14	UserCode Blocks	323
14.1	The Numerics of UCBs	324
14.1.1	Explicit UCBs	324
14.1.2	Implicit UCBs	325
14.2	The Structure of UCBs	326
14.2.1	Modes of Operation	326
	INIT Mode	327
	STATE Mode	327
	OUTPUT Mode	327

	MONIT Mode	328
	EVENT Mode	328
	LIN Mode	328
14.2.2	UCB Templates	329
14.2.3	UserCode Function Calling Arguments	329
	IINFO Array	330
	RINFO Array	332
	Mode Parameters	332
14.2.4	Direct Terms	335
14.2.5	State Events	338
14.3	How SystemBuild Executes UserCode Blocks	339
14.3.1	Execution of STATE Versus OUTPUT Modes in the UserCode ..	339
14.3.2	Timing Attributes	340
14.3.3	Initialization	340
	Simulation INIT Modes	340
	Impolite UCB Initialize Mode	341
14.3.4	Numerical Integration Algorithm	342
14.3.5	Operating Points	344
	Implicit Integration Algorithm Operating Point	344
	Computing the Operating Point Jacobian Matrix	345
	Computing the Implicit Solver Jacobian Matrix	346
14.4	Variable Interface UserCode Blocks	346
14.4.1	Using a Wrapper for SystemBuild to Simulate Code Written for AutoCode	347
14.4.2	Writing a Wrapper	349
	Converting Data from the sim() Interface	349
	Converting Data Back to the sim() Interface	350
14.4.3	Specifying the Variable Interface	350
	Setting Variable Interface Parameters	350
	Specifying Data Types	351
	Specifying Input Shapes	351

	Specifying Output Shapes	351
14.4.4	Running a Variable Interface Example	352
	Simulating the Variable Interface UCB in SystemBuild	352
	Generating Code for a Variable Interface UCB in SystemBuild ..	352
14.5	UCB Programming Considerations	353
14.6	Building, Linking and Debugging UCBs	353
14.6.1	Collecting UserCode Files	354
	Parameters Tab of the UserCode Block Dialog	354
	CSOURCE and FSOURCE	355
	Specifying Sources in the makefile	355
	Reusing Sources from the Previous Simulation	356
	Specifying Another Location for UCB Code	356
14.6.2	Compiling and Linking User Code	356
14.6.3	Debugging User Code	357
14.7	Posting Error Indications	359
14.7.1	Writing User Messages to the Xmath Window	359
14.7.2	Simulation Errors	360
14.8	Simulation API	361
14.8.1	Gathering UCB Reference Information	362
14.8.2	Accessing and Modifying Variables	363
14.8.3	Accessing Simulation Debugging Information	366
	Functions to Initialize and Terminate Debug Data	367
	Functions to Return the Dimensions of the SystemBuild Model	367
	Functions to Return Signal Names of the SystemBuild Model ...	367
	Functions to Return Signal Values of the SystemBuild Model	368
	Functions to Return the Jacobians of the SystemBuild Model	368
	Function to Return the Simulation Status of simexe()	369
14.8.4	SIMAPI Debug UserCode Block Example	370
14.8.5	SIMAPI Debugging Notes	371

15	BetterStateChart Blocks	373
15.1	Using BetterStateChart Blocks in SystemBuild: The Basics	373
15.1.1	SuperBlock Types and BetterState Control Implementations	374
15.1.2	Creating References to BetterState Charts in SystemBuild Models	374
15.1.3	Using SystemBuild Variables in BetterState Charts	377
15.1.4	Using Procedure SuperBlocks in BetterState Charts	377
	Using Procedure SuperBlocks as Actions	377
	Navigating to Procedure SuperBlocks from BetterState	380
15.2	Models That Use BetterStateChart Blocks	380
15.2.1	Modeling Conditional Logic	383
15.2.2	Event-Based Controller	387
15.2.3	Plants Whose Operating Point Changes Based on Conditions ...	392
15.3	BetterStateChart Blocks in Simulation Models	396
15.4	BetterStateChart Blocks in Models for Which You Use AutoCode	396
16	Fixed-Point Arithmetic	399
16.1	Introduction to Fixed-point Arithmetic	401
16.1.1	Fixed-point Number Representation	401
16.1.2	Conversion Between Fixed-point Numbers	404
16.1.3	Addition and Subtraction	405
16.1.4	Multiplication	405
16.1.5	Division	406
16.1.6	Relational Operations	407
16.1.7	Overflow	408
16.2	SystemBuild Fixed-point	409
16.2.1	User Interface	409
16.2.2	Simulator	410

16.2.3	Building a Model and Demonstrating Overflow	411
16.2.4	Comparing Fixed- and Floating-Point Numbers	416
16.2.5	Comparing the Effects of Different Conversion Sequences	421
16.3	Fixed-point Blocks and I/O Data Type Rules	424
16.3.1	Advanced Simulation Topics	427
	Intermediate Types	427
	Simulation Issues	431
	32-bit Operation Issues	433
	Gain Block: A Special Case	434
16.3.2	Radix Calculations	434
16.4	MinMax Data Logging	439
16.4.1	Activating MinMax Logging	439
	Simulating with the minmax Keyword	439
	Saving MinMax Data Sets to a File	440
16.4.2	MinMax Display Tool	440
16.5	User-Defined Data Types (UserTypes)	442
16.5.1	UserType Editor	442
16.5.2	UserType MathScript Commands	444
16.5.3	Using UserTypes in SystemBuild	444
16.5.4	Storing UserTypes	445
16.6	SystemBuild Functions in Fixed-Point	446
16.6.1	Linearization Function	446
16.6.2	Simout Function	446
16.7	Scaling Aid Blocks	447

17	Components	449
17.1	Introduction	449
17.1.1	Component Scope	450
17.1.2	Component Interface	451
17.1.3	Component Parameter Sets	451
17.1.4	Component References	451
17.1.5	Component Access	452
	Open Components	452
	Encrypted Components	452
	Licensed Components	453
17.2	Using Components in SystemBuild Models	454
17.2.1	Viewing Components	454
17.2.2	Creating References to Components	454
17.2.3	Controlling Component Parameters	455
17.2.4	Loading Component Parameter Sets	456
17.2.5	Changing Scope into a Component Catalog	456
17.2.6	Simulating Models with Components	457
17.3	Creating Components	457
17.3.1	Restrictions on Component SystemBuild Hierarchies	458
17.3.2	Understanding Parameterization of Components	458
17.3.3	Understanding the Component Scope	459
17.3.4	Mapping Exported Variables	460
17.3.5	Customizing the Component Dialog	460
17.3.6	Documenting the Component	461
17.3.7	Creating Components Using the Component Wizard	461
17.3.8	Modifying Components	462
17.3.9	Unmaking a Component	463

17.4	Creating and Using Parameter Sets	463
17.5	Using SBA with Components	465
17.6	Distributing SystemBuild Components	465
17.7	Examples	466
17.7.1	Encapsulating a SuperBlock Hierarchy	466
17.7.2	Exporting Component Parameters	467
17.7.3	Using the Parameter Set Interface	469
17.7.4	Interface Mapping	471
17.7.5	Using a Custom Dialog	474
18	SystemBuild Customization	477
18.1	User Initialization File	477
18.1.1	File Format	478
18.1.2	Printer Settings (UNIX)	480
18.1.3	Default Text Editor	480
18.1.4	Comment Editor	481
18.1.5	Custom Menus	481
18.1.6	A Typical Template for User Menus	482
18.1.7	Using the Sample User Initialization File that Calls MSCs	484
18.2	SystemBuild Resource File (UNIX)	484
18.2.1	Controlling Colors	485
	Foreground and Background	485
	SystemBuild and ISIM Color Settings	485
18.2.2	Resizing, and Repositioning the Display	486

19	Custom Icons	487
19.1	IA Basics	487
19.1.1	Adding a Custom Icon to a Block Diagram	488
19.1.2	Sample Icon Source	488
19.2	Defining Custom SystemBuild Icons	489
19.2.1	Importing or Referencing an External Bitmap	489
19.2.2	Creating or Attaching an IA Source Icon	490
19.3	Icon Source File	494
19.3.1	Icon Identification	496
19.3.2	Types	497
	Hardcoded Integer Type	497
	Hardcoded Real Types	497
	Hardcoded String Type	497
19.3.3	General Control and Calculation Statements	498
19.3.4	General Graphic Statements and Coordinate System	499
19.3.5	General Graphic Characteristic Statements	503
19.3.6	Animation Statements	505
19.3.7	Pointer Action Statements	506
19.3.8	Palette Definition	507
19.4	Animation Configuration File	508
19.4.1	Important Animation Configuration Keywords for Customized Icons	510
19.4.2	Icon Source File for Customized Icons and New Palettes	511
19.5	Building Your Own IA Custom Icons	512

20	Custom Palettes and Blocks	515
20.1	Custom Palettes	516
20.1.1	Creating Palette Files	516
20.1.2	Palette File Syntax	518
	PaletteFile	518
	BlockDirectory	520
20.1.3	Defining the Default SystemBuild Palette	520
20.1.4	Closing and Reloading the Default Palette	521
20.2	Custom Blocks	521
20.2.1	What Kinds of Blocks Can Be Customized?	521
20.2.2	Creating a Basic Custom Block	522
20.2.3	Creating More Sophisticated Custom Blocks	525
	Step 1: Create a Custom Block Using the Custom Block Wizard	525
	Step 2: Add a Custom Block to a Custom Palette File	526
	Step 3: Open the Custom Palette File	526
20.2.4	Including a Startup MathScript File with the Custom Block	527
20.2.5	Including a Custom Help File with the Custom Block	528
20.2.6	Additional Custom Block Features	530
	Dependent File	530
	Icon for the Palette Browser	530
	Parameter Sets	531
20.2.7	Using Relative Paths for Icon Files	531
20.3	Supporting Commands and Functions	533
20.3.1	SystemBuild Access Support	534
20.3.2	SystemBuild Utilities (SysbldEvent, SysbldRelease)	534
	SysbldEvent	534
	SysbldRelease	535
	Bibliography	537
	Index	539

1

Introduction

Congratulations on purchasing SystemBuild™, the industry's most powerful system modeling and simulation package. This manual introduces you to many features of SystemBuild. It focuses primarily on model building and editing tasks, object relationships, and conceptual descriptions of complex topics, such as analysis and simulation.

Although the SystemBuild blocks and SystemBuild functions and commands are introduced here, they are discussed in detail in online Help. For convenient descriptions of block dialogs and the user interface, see the context-sensitive online Help. For information on using online Help, type **help help command** in the Xmath command area.

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. The *Xmath User's Guide* explains how to use the Xmath® analysis and design package.

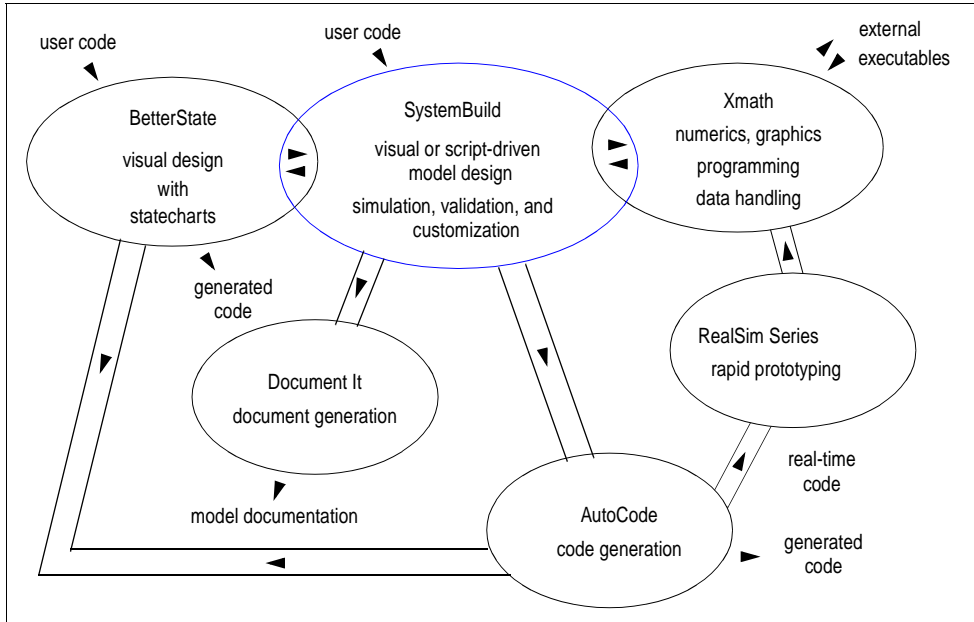
Additional resources are as follows:

- The STD Editor is covered in the *SystemBuild State Transition Diagram Block User's Guide*.
- The detailed operations of BetterState™ are described in the *BetterState User's Guide*. The interface with SystemBuild is described in this manual.
- The *BlockScript User's Guide* covers the usage of the BlockScript language for both SystemBuild and BetterState.

1.1 Product Overview

The SystemBuild design environment plays a central role in the MATRIX_x[®] product family (see Figure 1-1).

Figure 1-1 The MATRIX_x Product Family



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. You must start SystemBuild from the Xmath Commands window, but you can exit from either SystemBuild or Xmath. The first window to appear is the SystemBuild Catalog Browser.

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, a user-friendly graphical interface and extensive library of blocks, utility functions, and tools make SystemBuild the fastest way to express an engineering concept as a graphical model; you can validate a model design 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.

The remainder of this chapter focuses on the following topics:

- [Starting SystemBuild](#)
- [Exiting SystemBuild](#)
- [Running SystemBuild Demos](#)

1.2 Starting SystemBuild

To start SystemBuild:

From the Xmath Commands window, select Window→SystemBuild, or type the following command in the command area:

```
build
```

The Catalog Browser appears.

The **build** command also allows you to specify a SuperBlock name. This syntax is:

```
build "SB_Name"
```

where *SB_Name* is the name of a SuperBlock in string form. When you use the command in this form, the following rules apply:

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

1.3 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 appears. Click the Help button on the dialog if you need assistance.

Exiting from the Catalog Browser leaves Xmath running.

To exit all MATRIX_X processes at once:

From the Xmath Commands window, select File→Quit on UNIX or File→Exit on Windows, or type **quit** in the command area.



NOTE: You are prompted to save, but only a full save to **save.xmd** is performed in this circumstance.

1.4 Running SystemBuild Demos

We provide a collection of demos that demonstrate various features of SystemBuild. User messages are logged in the Xmath message area.



WARNING: Running a SystemBuild demo deletes SystemBuild and Xmath objects currently in your workspace. You might want to save them before you start.

To run a SystemBuild demo:

1. Type **demo** in the Xmath command area.

The Xmath Demos dialog comes on view.

2. Select SystemBuild Demos, and then click OK.

If you have a model loaded, the Save SysBld & Xmath workspace dialog comes on view.

3. Indicate whether you want your model saved.

The SystemBuild Demos dialog comes on view.

4. Select the desired demo, and click OK.

Information is presented to you in the Xmath log area, and additional dialogs are presented to you.

5. Follow the instructions for the selections that you make.

2

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, BetterState, 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 Catalog Browser is the entry point to the SuperBlock Editor, the State Transition Diagram (STD) Editor, and the BetterState Editor (when BetterState charts are used within SystemBuild models).

This chapter guides you through Catalog Browser tasks associated with the Catalog view. The major topics are as follows:

- *[Loading Data](#)*
- *[Examining Catalog Components in the Catalog View](#)*
- *[Working with Catalog Views](#)*
- *[Saving Data](#)*
- *[Working with the Catalog Browser](#)*

The Catalog Browser also allows you to perform file and configuration management; these tasks are associated with the File view; these items are discussed in [Chapter 3](#).

2.1 Loading Data

You can load data into SystemBuild from either Xmath or SystemBuild. Loading data is like importing data; it becomes part of the current catalog without retaining any association of where the data came from. Loading data is different from opening a file. You can use either operation to load and use data in SystemBuild, but opening a file also associates the loaded data with a file for saving it at a later time. See [File Management](#) on p.33 for further information.

2.1.1 Loading Data from Xmath

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

2.1.2 Loading Data from the Catalog Browser in SystemBuild

From the Catalog Browser, you can load a whole file, or you can selectively load data.

To load a file from the Catalog Browser:

1. Select File→Load.

The Load dialog appears.

2. Select the file and the data to be loaded, and then click OK.

Click the ? button for information on using this dialog.

[Example 2-1](#) shows you how to selectively load data using the Advanced Load dialog. The data loaded in the example is used throughout this chapter.

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"
```

We refer to this file throughout this chapter.

2. Start SystemBuild from Xmath by selecting Windows→SystemBuild.

The Catalog Browser is launched.

3. In the Catalog Browser, select File→Load. When the Load dialog appears, select the file **cb_ex1.cat**, but do *not* click OK at this point.

This catalog file contains the folders Model, SuperBlocks, BetterState Charts, State Diagrams, DataStores, Components, Variables, and UserTypes. The most common action is to load an entire catalog; however, in this example we'll load a little at a time.

4. Click the Advanced button to view the contents of the catalog before loading it.

The Advanced Load dialog appears; by default, an alphabetical listing of all objects in the Model folder of the catalog are displayed on the right.

In the Table of Contents, each folder contains a listing of different object types.

5. Click each different folder in the Advanced Load dialog to see what it contains.

6. Load selected SuperBlocks as follows:

- a. Click the SuperBlocks folder.
- b. Enable the Load Hierarchy checkbox.

SuperBlocks can contain other SuperBlocks, forming a SuperBlock hierarchy. Enabling this checkbox ensures that if you load a single SuperBlock, all SuperBlock elements included in its hierarchy, including State Diagrams or DataStores, are also loaded. It does not affect objects other than SuperBlocks, such as components, libraries, STDs, or DataStores that are not included in diagrams.

- c. Select continuous automobile from the Contents pane of the Advanced Load dialog.
- d. Hold down the Ctrl key, and click Cruise Control System to select an additional SuperBlock.

This action selected multiple SuperBlocks because the Load Hierarchy checkbox is enabled.

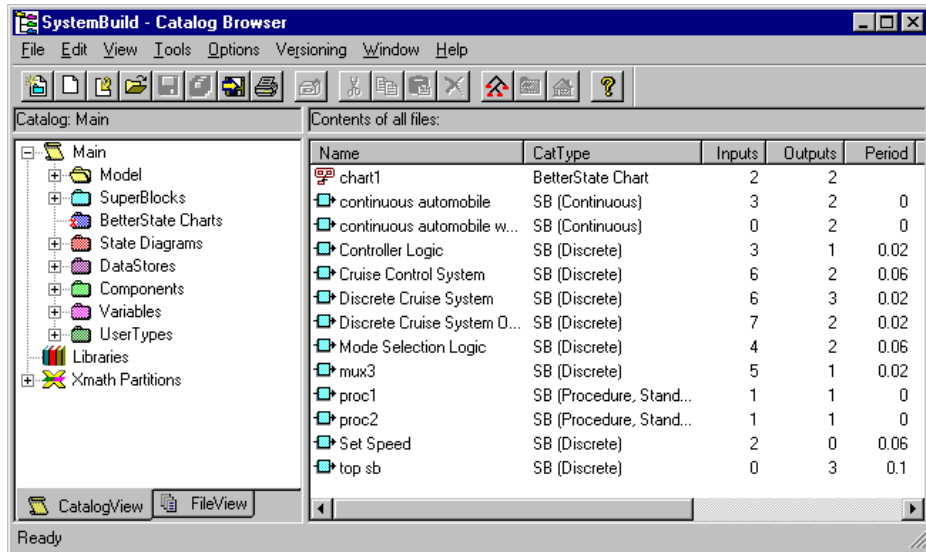
- e. Continue selecting SuperBlocks until you have selected all SuperBlocks.
- f. Click Apply to load all selected SuperBlocks.

7. Select the BetterState Charts folder. Select chart1. Click Apply to load the selected chart.

8. Select the State Diagrams folder. Select the first STD; holding down the Shift key, select the last one. Click Apply to load them.
9. Select the DataStores folder. Select Desired Speed. Click Apply.
10. Select the Components folder. Select the component ninteg, and click Apply to load it.
11. Click the UserTypes folder. Select all the UserTypes. Click Apply.
12. Click OK to complete the advanced load.

The Catalog Browser is shown in [Figure 2-1](#) with `cb_ex1.cat` loaded.

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



2.2 Examining Catalog Components in the Catalog View

All data is loaded into the Main catalog; the catalog name is shown directly above the left pane (see [Figure 2-1](#)) when the CatalogView tab is used; this is the default view. Libraries are shown as a separate catalog.

2.2.1 Main Catalog

The Main catalog has eight folders: Model, SuperBlocks, BetterState Charts, State Diagrams, DataStores, Components, Variables, and UserTypes. These objects must all have unique names. In this section, we examine each of these components.

For all the folders, clicking an element in the left pane of the Catalog Browser causes the display of all the contained elements in the right pane of the Catalog Browser. For example, clicking a particular SuperBlock causes all elements within that SuperBlock to be listed in the right pane of the Catalog Browser.

Model

The Model folder is a hierarchical node that supports the combination of two other folders: SuperBlocks and BetterState Charts. The elements at the left of the hierarchy are the highest level elements. If an element has a lower-level element, a + appears to the left of the element; clicking the + expands the hierarchy below that element. This continues in a tree-like fashion down the hierarchy. Similarly, if an element heads an open hierarchy, a – appears to the left of the element; clicking the – collapses the hierarchy. This works the same way hierarchical folders work in Windows Explorer and other similar products.

At the top level of the model are SuperBlocks and BetterState charts. When you select one of these elements, its contents are displayed in the right pane of the window. SuperBlocks contain blocks of various types; BetterState charts contain states and pages.

SuperBlocks

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*. The collection of primitive blocks can include:

- State Transition Diagrams (STDs)
- BetterState charts
- DataStores
- 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. References within a SuperBlock 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.

You can see and work with the SuperBlock hierarchy in the Model folder. You can find a list of SuperBlocks in the SuperBlocks folder.

We work with SuperBlocks throughout this chapter. Additionally, [Chapter 4, SuperBlocks](#), describes how to create SuperBlocks and SuperBlock references; [Chapter 6, SuperBlock Timing and Transformation](#), describes how the different types of SuperBlocks influence a model's timing.

BetterState Charts

BetterState charts (statecharts) are state transition diagrams extended with hierarchy, concurrence, history states, and event handling. BetterState charts also allow you to use flowchart constructs.

In this folder, you can find a list of BetterState charts; when you click a chart, its states and pages are displayed in the right pane. BetterState charts can also be hierarchical; you can work with this hierarchy in the Model folder.

BetterState is a separate product that runs concurrently with SystemBuild. You can include BetterState charts within your SystemBuild model, or you can run BetterState standalone and create BetterState charts for other applications.

BetterState charts are documented in the *BetterState User's Guide*.

State Diagrams

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

DataStores

DataStores are blocks that provide global data storage. Clicking the DataStores folder displays all DataStores in the model in the right pane.

In the Xmath command area, type **help datastore** for detailed information.

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 unique names. Although each component must have a unique name, objects encapsulated within it are not visible to other catalogs because a component has its own scope. However, variables can be exported from a component. [Chapter 17](#) discusses components in detail.

Variables

Selecting this folder displays all global variables defined at this scope of the catalog. Global variables are either global variable blocks or BetterState global variables. If the same variable appears multiple times as a different data type, SystemBuild indicates an unknown type.

UserTypes

UserTypes is a feature that allows you to assign meaningful names to data types (see [User-Defined Data Types \(UserTypes\)](#) on [p.442](#)). When you define and use UserTypes in your models, you can see what they are by clicking the UserTypes folder. Expanding the folder lists them in the left pane; clicking the folder lists them in the right pane.

UserTypes are stored in the Main catalog and uses the same catalog name space; this means that your UserTypes must have unique names among all catalog items. Prior to Release 7.0, UserTypes were stored in the **Xmath_datatypes** partition. UserTypes in models created prior to this release are automatically converted to catalog objects; if a name collisions occurs, the UserType and any references to it are renamed to **udt_oldUserTypeName**.

UserTypes have full SBA support (**createusertype**, **modifyusertype**, **deleteusertype**). See the SBA topic in online Help.

2.2.2 Libraries

Libraries are SystemBuild model files that are not loaded directly into SystemBuild, but you can reference their contents from SystemBuild. You load

libraries by setting the **sblibs** keyword to the name of the model file(s) in Xmath using the **setsbdefault** command; for example,

```
setsbdefault, {sblibs="file1.cat file2.cat"}
```

where the files can be File SuperBlocks (see [Using File SuperBlocks](#) on p.66) or file components (see [Component References](#) on p.451).

2.2.3 Xmath Partitions

The Xmath Partitions folder provides read-only access to the currently defined Xmath partitions and the variables that exist in those partitions. Clicking the folder itself lists the partitions in the right pane. Expanding the folder shows the partitions in the left pane. Clicking a partition shows the variables in the partition in the right pane.

2.3 Working with Catalog Views

As seen in [Figure 2-1](#), 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 (Model, SuperBlocks, BetterState Charts, State Diagrams, DataStores, and so forth), a listing of all currently defined catalog items of the selected type is displayed in the Contents view.

This section introduces some of the ways you can view and organize catalog items in a tutorial format.

1. If you do not have the model loaded, load it following the instructions provided in [Example 2-1](#).
2. List all SuperBlocks and BetterState charts by clicking the Model folder in the Catalog view.

The Contents view displays all currently defined SuperBlocks and BetterState charts in the catalog.

3. Click the + symbol next to the Model folder to list all top-level SuperBlocks and BetterState charts in the Catalog view.

4. Select the Discrete Cruise System top-level SuperBlock to view the SuperBlock contents in the Contents view.

The Contents view displays a primitive as a simple block icon and lists the block type.

Each top-level SuperBlock (shown as a folder preceded by a +) has child items you can view in the Contents view (the right pane).

5. Click the + symbol to expand a subhierarchy in the Catalog view.

The symbol changes to a minus sign (-); clicking on the minus sign collapses the hierarchy.

6. To select all levels of a hierarchy, select a SuperBlock, and then select Edit→Hierarchy Select Mode, or click the Hierarchical select mode toolbar button



In both the Catalog view and the Contents view, SystemBuild also selects all other referenced elements in the selected SuperBlock's hierarchy. This feature stays on until you toggle it off the same way.

7. Click the SuperBlocks folder.

All SuperBlocks appear in the Contents view.

8. Expand the SuperBlocks folder by clicking the + beside it.

A list of all SuperBlocks in the model appears (with no hierarchy indicated).

9. Select any one of the SuperBlocks.

Its contents appear in the right pane.

10. Select the BetterState Charts folder.

A list of BetterState charts in the model appears in the right pane.

11. Expand the BetterState Charts folder.

A list of BetterState charts appears in the left pane.

12. Select any one of the BetterState charts.

Its contents—states, pages, and any referenced procedure SuperBlocks—appears in the right pane.

13. Select the State Diagrams folder, and then expand the State Diagrams folder.

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

14. Repeat [Step 13](#) for DataStores, Variables, and UserTypes.

Each of these folders displays primitive information.

15. Expand the Components hierarchy. Select nlinteg, and then select View→Component Catalog.



NOTE: Component Catalog is also available from the Shortcut menu, which you can obtain by right-clicking the mouse after you select the object.

The Catalog view changes from Main to nlinteg, and the Contents view changes accordingly.

16. To return to the Main catalog, select View→Main Catalog, or right-click, and then select Main Catalog.
17. Select the Libraries folder to see the filenames of the libraries available to this model in the right pane.
18. Expand the Libraries folder in the Catalog view to see the library filename in the Contents view.
19. Click the filename in the Catalog view to see the SuperBlocks in this library listed in the Contents view.

You can drag any of these SuperBlocks in the Contents view into your model, but you can't open these SuperBlocks.

20. Select the Xmath Partitions folder to see a list of partitions defined in Xmath in the Contents pane.
21. Expand the Xmath Partitions folder to see a list of partitions in the Contents pane.
22. Select one of the partitions in the Catalog pane to see its contents displayed in the Contents pane.
23. To change the width of the panes to accommodate your data, widen the window or adjust the pane width. This task is slightly different on UNIX and Windows:

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

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

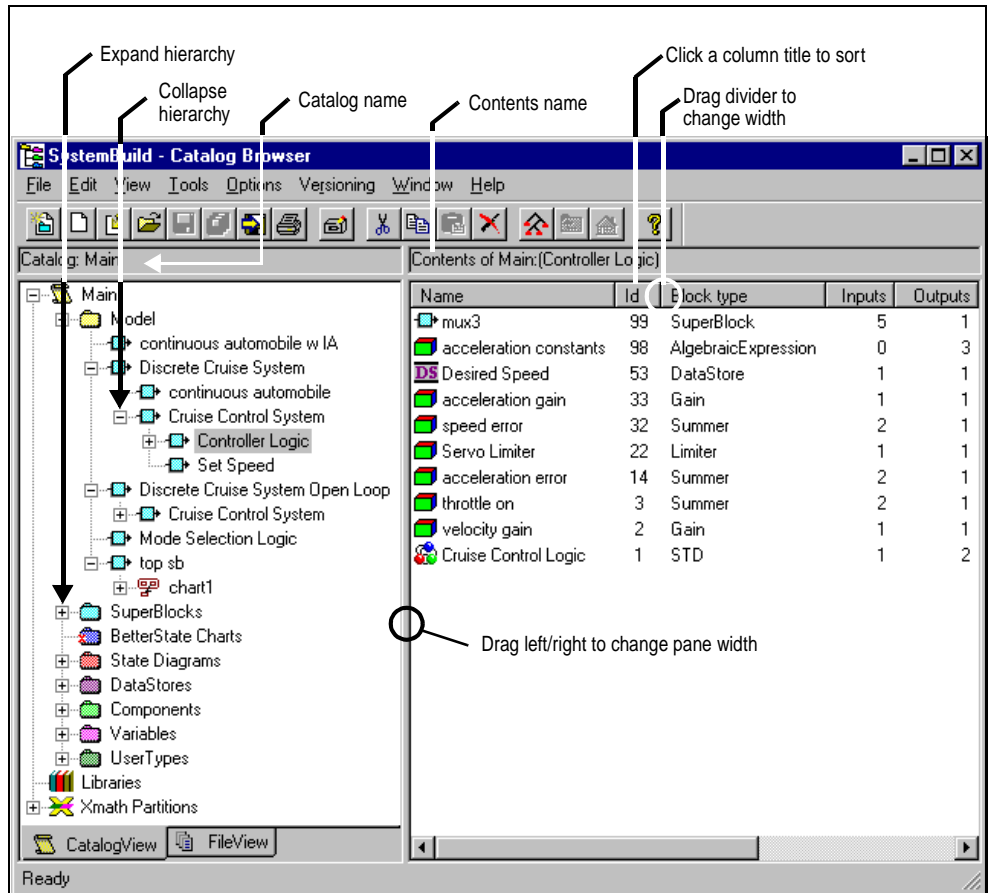
24. To change the width of a column in the Contents view. click directly on the dividing line, and drag left or right.

- 25. Click the heading of any column in the Contents view to sort the contents by that category. Click the heading a second time to reverse the sort.

By default, items in the Contents view are displayed in alphabetical order by name.

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)



2.4 Saving Data

This section describes how to save the current model without using the File view. You can save your catalog from Xmath or the Catalog Browser. In addition, SystemBuild has an automatic save feature, AutoSave, that can automatically save your catalog data at a specified interval.

You can also use file management directly in the File view of the Catalog Browser to save data. For more information, see [File Management](#) on p.33.

2.4.1 Saving Data from Xmath

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}
```

Each keyword specifies a subset of data to be saved. The default behavior is inclusive, as demonstrated in the following syntaxes:

<code>SAVE {<i>ascii</i>}</code>	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.)
<code>SAVE "<i>fname</i>"</code>	Saves all data to a file you specify.
<code>SAVE "<i>fname</i>" {<i>build</i>}</code>	Saves all SystemBuild data, including Xmath data and UserTypes (see 16.5 User-Defined Data Types (UserTypes) , p.442).
<code>SAVE "<i>fname</i>" {<i>superblock</i>="<i>sname</i>",<i>hierarchy</i>}</code>	Saves the specified SuperBlock, all SuperBlocks in its subhierarchy, and all Xmath data.

For a full description, see the Save topic in online Help.

2.4.2 Saving Data from the Catalog Browser



NOTE: The Catalog Browser allows you to save all data or selected data. In the Catalog view only, the File→Save option behaves exactly like the File→Save As option.

2.4.3 Saving All Data

To save all data from the Catalog Browser:

1. Click in the Main catalog to deselect any selected items.
2. Select File→Save As.

This raises the Save As dialog; the basic functionality is identical to that of the **SAVE** command, with the exception that the default save format is ASCII. The default format also saves all data.

3. Specify the filename, and click OK.

Click the dialog's ? button for additional information.

Saving Selected Data

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.

You must make the selection(s) *before* you invoke the Save dialog; when the dialog is raised you must enable the Selected radio button for SuperBlocks. [Example 2-2](#) demonstrates this process:

Example 2-2 Saving Selected Data

In this example you save the file you loaded earlier ([Figure 2-2](#)) by selected catalog items.

1. Select Edit→Hierarchy Select Mode.
2. Click Cruise Control System.

All objects in this system hierarchy are highlighted.

3. Select File→SaveAs.

4. In the Save dialog, enable the Selected SuperBlocks radio button.



NOTE: If you don't enable this option, your preselections are ignored, and SystemBuild saves all data.

5. Specify the name **cb_ex1_ccs.cat**.
6. Click OK.

The preselected objects are saved in the specified file.

2.4.4 Using AutoSave to Save Data

SystemBuild has an automatic save capability that you can activate by resetting the SystemBuild defaults from the Xmath command area with the **SETSBDEFAULT** command. You must specify two parameters: **autosavefile** and **autosavetime**, and both must have value (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 is saved.

The AutoSave feature saves catalog data only; Xmath data is not supported by 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, type **help SETSBDEFAULT** in the Xmath command area, and then look at the Help on the **autosavefile** and **autosavetime** keywords.

Example 2-3 Using AutoSave

1. Start AutoSave by typing the following command from the Xmath command area:

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

2. From the Catalog Browser, select View→Update to update the new values.

AutoSave is now set to save all catalog data to a file named **autosave.cat** every 60 minutes.

3. To change the save time interval to every two hours, type in the Xmath command area:

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

4. Select View→Update in the Catalog Browser to activate the new interval.

5. To check the current AutoSave settings, type the following in Xmath:

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

6. To turn off AutoSave, set `autosavetime` to 0 with the following command in Xmath:

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

7. Select View→Update in the Catalog Browser.

2.5 Working with the Catalog Browser

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

2.5.1 Using the Shortcut Menu in the Catalog Browser

The Shortcut menu, also known as 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. We discuss the usage of this menu first because you are going to use it throughout this section.

To raise the Shortcut menu:

- | | |
|----------------|--|
| UNIX | Right-click an object; drag to select a menu item. |
| Windows | Right-click an object and release; the menu appears. Select a menu item. |

Menu items might be inactive depending on the preselected object(s). In particular, if the hierarchy select mode is enabled, most Shortcut menu items are inactive because the selections are only valid if a single object is selected.


2.5.2 Opening a SuperBlock or BetterState Chart in an Editor

You can open any SuperBlock or BetterState chart listed in the Catalog Browser in the appropriate Editor using any one of several methods.


To open a SuperBlock or BetterState chart from the Contents view of the Catalog Browser:

1. In the Catalog view of the Catalog Browser, select the Model folder, but leave it in the collapsed state (+).

This lists all currently defined SuperBlocks and BetterState charts in the Contents view.

2. In the Contents view, double-click the SuperBlock or BetterState chart that you wish to see in an Editor, or select it and open this SuperBlock or BetterState chart using one of the following techniques:
 - a. Right-click, and select Open from the Shortcut menu.
 - b. Click the Open toolbar button .
 - c. Select Edit→Open.

To open a SuperBlock or BetterState chart from the Catalog view of the Catalog Browser:

1. Expand the model hierarchy in the Catalog view in the Model folder.
2. Select the SuperBlock or BetterState chart that you wish to see in an Editor. Open this SuperBlock or BetterState chart using one of the following techniques:
 - a. Right-click, and select Open from the Shortcut menu.
 - b. Click the Open toolbar button .
 - c. Select Edit→Open.

The following is convenient information to know at this point:

- In the Catalog view, the icon for each SuperBlock or BetterState chart open in the Editor has a magenta check, indicating that it is open.
- You can edit up to 20 SuperBlocks simultaneously. Each SuperBlock opened appears in a separate Editor window.
- You can also have multiple BetterState charts open simultaneously. Each appears in a separate Statechart window (BetterState Editor).

- The scope of the SuperBlock being edited appears in the title bar of the Editor window. 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 Window menu in the Catalog Browser provides the ability to iconify, deiconify, or close all editors. The bottom of this menu displays a list of the SuperBlocks, BetterState charts, and BetterState chart pages being currently edited. Because the name includes the scope, it might be truncated. Clicking the name makes the editor containing that SuperBlock, BetterState chart, or BetterState page the active window.
- You cannot edit the same object in multiple windows. If a SuperBlock is currently open in an editor window, attempting to open the object merely raises the editor that contains it.

2.5.3 Creating a New SuperBlock

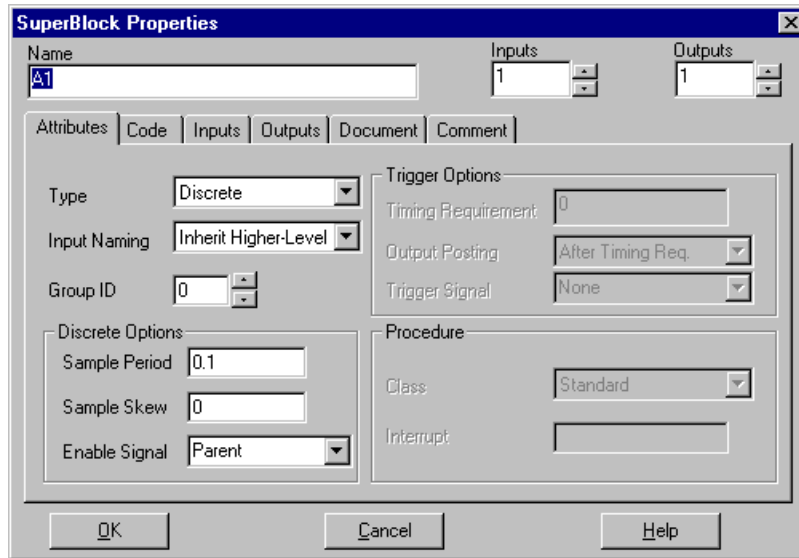
You can create new SuperBlocks from the Xmath or SystemBuild (see [4.2](#), p.50).

To create a new SuperBlock from the Catalog Browser:

1. Select File→New→SuperBlock.

The SuperBlock Properties dialog appears atop an empty editor window.

Figure 2-3 SuperBlock Properties Dialog



Click the Help button for a full description of each field.

2. Give the SuperBlock a unique name.


You can also provide other properties from this dialog, but Name is the only required field. Providing an existing name of a SuperBlock produces an error.

3. Click OK.

An editor window opens with a SuperBlock ID bar, the strip of SuperBlock information that appears below the editor's toolbars and above the diagram work area, with the name and inputs and outputs. The new SuperBlock does not yet appear in the Catalog Browser.

4. Create the SuperBlock to the desired degree at this point.

You must create at least one block in the SuperBlock. We cover this information in [Chapter 4](#).

5. Select File→Update or click the Update toolbar button  from the editor window to place this SuperBlock in the Catalog Browser as a top-level SuperBlock in the Catalog view.

[Example 2-4](#) takes you through the process of creating a new SuperBlock and updating the Catalog Browser. Refer to the procedure above as necessary.

Example 2-4 **Creating a New SuperBlock and Updating the Catalog Browser**

1. Create a new SuperBlock.
2. Name the SuperBlock NewSB, change the type to Discrete, and click OK.
A new blank SuperBlock appears in the editor window.
3. In the Catalog view, click the SuperBlock label to display all currently defined SuperBlocks.
Note that NewSB does not appear.
4. Press F5 to update the Catalog Browser.
The SuperBlock NewSB now appears in the SuperBlock folder.

2.5.4 Creating a New BetterState Chart

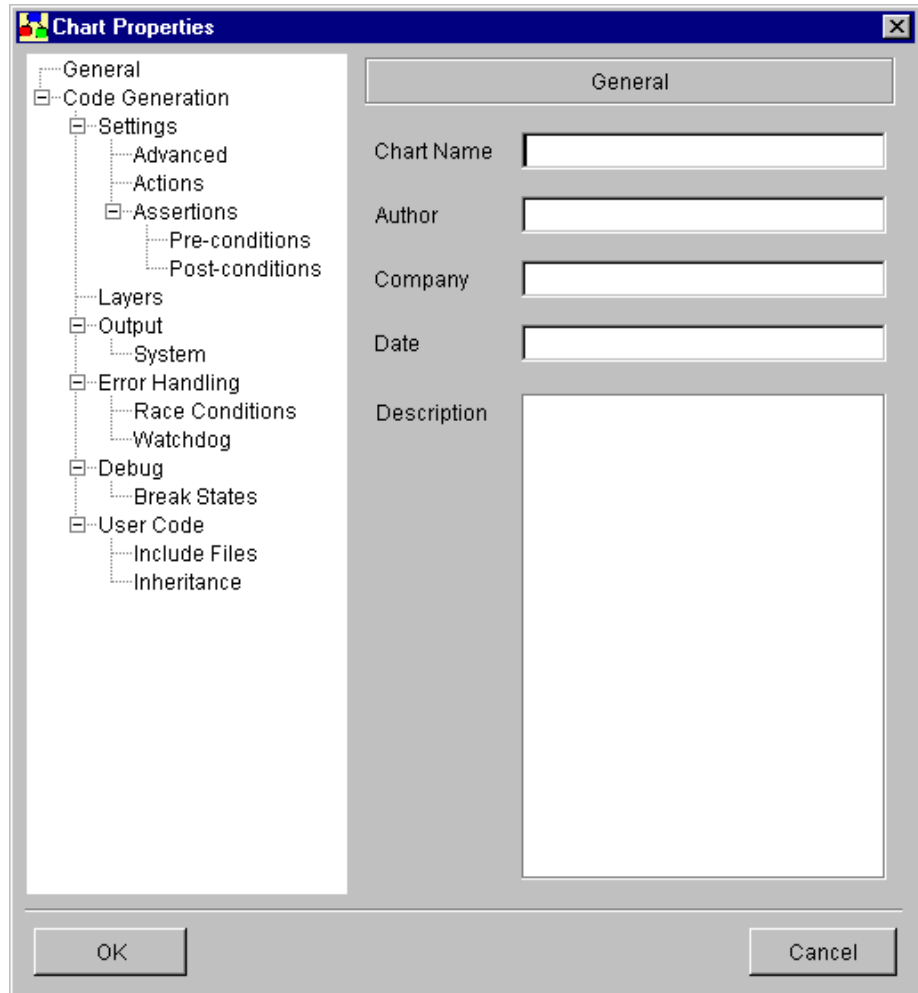
You can create new BetterState charts from the Catalog Browser. You can also drag a BetterStateChart block icon from the Palette Browser into a SuperBlock Editor, creating a reference to a new chart and then open the new chart in the BetterState Editor (see [Creating References to BetterState Charts in SystemBuild Models](#)).

To create a new BetterState chart from the Catalog Browser:

1. Select File→New→BetterState Chart.

The Chart Properties dialog appears atop an empty BetterState Editor window (see [Figure 2-4](#)).

Figure 2-4 Chart Properties Dialog



See the *BetterState User's Guide* for a full description of each field.

2. Give the BetterState chart a unique name, and click OK.

You can also provide other properties from this dialog, but the Chart Name is the only required field. (See the *BetterState User's Guide* for information on this dialog.) Providing an existing name of a BetterState chart produces an error.

2.5.5 Creating a New UserType

To create a new UserType from the Catalog Browser:

1. From the Catalog Browser, select File→New→UserType.

The UserType Properties dialog comes on view.

2. Enter the Name of your new UserType.
3. Select its data type from the combo box.
4. If the data type is fixed point, select the Radix.

The remaining fields on this dialog are read-only and pertain to fixed-point data types; they are dependent on the data type and the radix. These fields appear in the Contents view of the Catalog Browser beside each UserType.

2.5.6 Modifying a Catalog

[Example 2-5](#) shows you how to modify a catalog. If you do not currently have the `cb_ex1.cat` model loaded into the Catalog Browser, return to [Example 2-1](#) and load it.

Example 2-5 Modifying Catalogs

1. Expand the SuperBlocks folder in the Catalog view.
2. Select the Controller Logic SuperBlock. Right-click, and select Copy from the Shortcut menu.

The object is copied to the Clipboard.

3. Select Shortcut→Paste to paste the contents of the Clipboard into the catalog. SystemBuild places a SuperBlock named Copy of Controller Logic in the list of SuperBlocks.

4. Select Copy of Controller Logic, and then right-click to raise the Shortcut menu. Select Rename.

The Rename dialog appears.

5. Supply the name Modified Controller Logic, ensure that Rename all references is enabled, and click OK.

6. Open the new SuperBlock in the editor, and modify it as follows:
 - a. Select File→SuperBlock Properties.
The SuperBlock properties dialog appears.
 - b. Change the Sample Period to **0.01** and the Sample Skew to **0.01**.
 - c. Click OK.
7. From the editor, select Window→Catalog Browser to bring up the Catalog Browser.
8. In the 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 Sample Period and Sample Skew values because the changes haven't been written to the Catalog Browser.

9. In the Catalog Browser, select View→Update, or press F5 to force the update.

We discuss updating the Catalog Browser further in [2.5.9 Updating the Catalog Browser Data from an Editor](#).

2.5.7 Modifying a UserType

To modify a UserType from the Catalog Browser:

1. Select the UserType that you wish to modify in either the Catalog view or the Contents view.
2. Right-click to bring up the Shortcut menu, and select Properties. Alternatively, select View→Properties, or double-click a UserType in the Contents view.

The UserType Properties dialog comes on view.

3. Edit the Name, Data Type, or Radix fields, and click OK.



NOTE: Changing the name creates a new UserType, and the old one remains.

2.5.8 Dragging SuperBlock and BetterState Chart Icons from the Catalog Browser to the Editor

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

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

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

Dragging SuperBlock Icons from the Catalog View

Dragging a SuperBlock from the Catalog view into an existing editor window saves and closes the SuperBlock that was previously open in the editor. The dragged object becomes the current SuperBlock being edited.



NOTE: This technique does not work for BetterState charts.

Dragging SuperBlocks and BetterStateChart Icons from the Contents View

Dragging an object from the Contents view copies it into the SuperBlock being currently edited. If you drag a SuperBlock definition into the editor, a SuperBlock reference to the definition is created instead. Dragging a BetterState chart icon into a SuperBlock Editor creates a BetterStateChart block.

Using the Drag and Drop Feature

[Example 2-6](#) provides an example that lets you try the drag and drop technique. If you do not currently have the `cb_ex1.cat` model loaded in the Catalog Browser, return to [Example 2-1](#) and load it. If you have no editor window open, open one (see [2.5.2 Opening a SuperBlock or BetterState Chart in an Editor](#)).

Example 2-6 **Catalog Browser Drag and Drop Features**

1. Expand the SuperBlocks folder in the Catalog view.
2. 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 SuperBlock Editor window.

You have loaded the Discrete Cruise System SuperBlock into the editor; it has three blocks.

3. In the Catalog view of the Catalog Browser, select Discrete Cruise System Open Loop SuperBlock. From the right pane, drag the SuperBlock reference, Cruise Control System, into the editor.


The SuperBlock does not open. A reference is created, and it becomes part of the diagram.

4. Click the BetterState Charts folder in the Catalog view.
5. Select chart1 in the Contents view.
6. Drag the BetterState chart reference, chart1, into a SuperBlock editor.

A reference to the BetterState chart is created, and a BetterStateChart icon appears on the diagram.

2.5.9 Updating the Catalog Browser Data from an Editor

When you modify a catalog item from the SystemBuild editor, changes are not written to the catalog unless you change view to another object in the same Editor window, close the editor window, load a file, or analyze a model. There are several ways to force an update:

- In an editor, press F5, click the Update toolbar button  , or select File→Update.
- In the Catalog Browser, press F5, or select View→Update.

Save only works for items updated to the catalog (see [2.4 Saving Data](#)).

2.5.10 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:

- SuperBlock transformations are fully explained in [Chapter 6](#).
- 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*.

However, you can try some of them using the current catalog. [Example 2-7](#) shows you how to use the transformation tool.

Example 2-7 Transforming a SuperBlock

This example uses the editing techniques explained in [2.5.6 Modifying a Catalog](#).

1. In the Catalog view of the Catalog Browser, copy the continuous automobile SuperBlock, and paste it into the catalog.
2. Rename the copy discrete automobile.
3. Select discrete automobile, and then select Tools→Transform.
The Transform SuperBlock dialog appears.
4. From the Type combo box, select Discrete.
5. Enable Transform Initial Conditions.
6. Click OK.

The new SuperBlock appears in the SuperBlock hierarchy.

3

File and Configuration Management

In [Chapter 2](#), we discussed using the Catalog view of the Catalog Browser. In this chapter we discuss *File Management* using the File view of the Catalog Browser.

You can think of the File view in the Catalog Browser as an orthogonal view of your model. All the same items exist, but rather than being associated with a particular category, such as SuperBlocks, each item is associated with a file.

The items in the File view exist at the granularity of the items in the Catalog view of the Catalog Browser. For example, you can work with SuperBlocks in the File view, but you can't work with the blocks within a SuperBlock; SuperBlocks are listed in the Catalog view, but the blocks that compose it appear in the Contents view.

Configuration Management is an extension of file management. It is the second major topic in the chapter.

3.1 File Management

In this section, we introduce you to a number of concepts and tasks regarding file management in SystemBuild. We want you to understand the following concepts:

- The difference between the Catalog view and the File view of a catalog
- The difference between loading and opening a file
- The role of the `Untitled.sbd` file

- The Current States of blank, New, Modified, Deleted, Moved, and Overwritten
- That only one item with a given name can exist in a catalog at once
- When files containing items with duplicate names are opened, all of those items are overwritten except one

In this section, we show you how to perform the following tasks:

- *Putting New Items into the Untitled Default File*
- *Modifying an Item in a File*
- *Deleting an Item in a File*
- *Moving an Item from One File to Another*
- *Overwriting Items in a File*
- *Creating a New File*

The following tutorial allows you to see the results; note that this entire section is a “running” tutorial although we divide it into various subsections and multiple examples. To avoid corrupting any examples or other files that you may need to keep, we ask you to establish a new working directory that you can delete at the end of the file management topic.

3.1.1 Putting New Items into the Untitled Default File

In [Example 3-1](#), you create a new working directory, load an example file into this directory, examine what happens in the File view, and see how the file **Untitled.sbd** works in this environment. You also get to see how SystemBuild allocates items to the **Untitled.sbd** file and how to save a copy of it for future purposes.

Example 3-1 Loading a File and Working with the Default Untitled File

1. Create a new directory called **fmlearning**, and make it your current working directory.

One method of changing the current working directory is using the following command from the Xmath command area:

```
set directory ".../fmlearning"
```

where ... denotes the high-level part of your pathname. (The forward '/' works for all operating systems in Xmath.)

2. 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"
```

3. If SystemBuild is not already running, launch it.
4. In the Catalog Browser, select File→Load. When the Load dialog appears, select file **cb_ex1.cat** from your current **fmlearning** working directory.
5. Click the FileView tab in the Catalog view of the Catalog Browser.
Notice that this view contains one file, **Untitled.sbd**.
6. Select **Untitled.sbd**.

Notice that all items from the Catalog view now appear in the Contents view with the Current State showing New. Notice also that the filename is shown in bold font rather than regular font.



NOTE: As you will see later in this chapter, showing the filename in regular font indicates that the contents of the file have not changed, and the file does not require saving.

7. Select File→Save.

SystemBuild requires you to specify a filename for the untitled file; a dialog appears for you to name the file.



NOTE: With the untitled file, selecting File→Save is the same as selecting File→Save As.

8. Name the file **cb_ex2.cat**, and store it in the **fmlearning** directory.

Notice that the new filename appears in the File view, but the Contents view is now empty. **Untitled.sbd** is selected, and the Contents view is empty, indicating that no items are without an associated file.

9. Select **cb_ex2.cat**.

Notice that the Current State of the items in the Contents view is blank, indicating that the contents in memory match the contents on disk.

10. Select View→Properties.

The Properties dialog shows you when you modified the file and whether it is under source control; this file is not.

As you've probably guessed by now, the file **Untitled.sbd** is always shown in the File view, whether or not any items are associated with it. You can think of this file as the "catch-all bucket" for items that need to be allocated to a file. Any items not associated with another file are automatically associated with this file.

3.1.2 Modifying an Item in a File

In this section, [Example 3-2](#) illustrates the effect of modifying an item in a file. You can see that SystemBuild alerts you when a file needs to be saved.

Example 3-2 **Modifying an Item in a File**

1. With the **cb_ex2.cat** file still selected in the File view, double-click the continuous automobile SuperBlock to bring it up in an editor.
2. Change the font size of this SuperBlock, and close that editor.

Returning to the File view of the Catalog Browser, notice that the Current State now says Modified. Notice also that the **cb_ex2.cat** filename now appears in bold, indicating that one or more items in this file has been changed, and the file needs to be saved.

3. With the **cb_ex2.cat** file still selected in the File view, select File→Save.

Notice that Current State in the Contents view is blank for all items, indicating that the contents of the file match the contents of the catalog.

3.1.3 Deleting an Item in a File

In this section, [Example 3-3](#) illustrates the effect of deleting an item from a file. Once again, you can see that SystemBuild alerts you when the file needs to be saved.

Example 3-3 **Deleting an Item from a File**

1. With the **cb_ex2.cat** file still selected in the File view, select the SuperBlock **mux3** in the Contents view.

2. Open the Edit menu.

Notice that the Delete menu item is not active.



NOTE: SystemBuild does not allow you to delete items in the File view.

3. Click the Catalog View tab.

4. Select the SuperBlocks folder.
5. In the Contents view, select the mux3 SuperBlock.
6. Select Edit→Delete.

The mux3 SuperBlock is deleted from the Contents view.

7. Click the FileView tab to return to the File view.

Notice that the SuperBlock remains in the Contents view, but its Current State is Deleted. Notice also that the filename is now in bold font, indicating that its contents do not match the contents in memory (needs saving).

8. With the **cb_ex2.cat** file selected, select File→Save.

The filename is again shown in regular font, and the Current State of every item is again blank.

3.1.4 Moving an Item from One File to Another

In this section, [Example 3-4](#) illustrates moving an item from one file to another without affecting what is in memory.

Example 3-4 Moving an Item from One File to Another

The Catalog Browser is in the File view with the **cb_ex2.cat** file selected.

1. In the Contents view, select the Cruise Control Logic STD.
2. Select Edit→Cut.

You can find Cut on the Shortcut menu by right-clicking as well.

3. Select the **Untitled.sbd** filename.

Notice that the file has no initial contents.

4. Select Edit→Paste.

Notice that the **cb_ex2.cat** file is shown in bold, indicating that its file contents do not match the contents in memory, and it needs to be saved. Notice, too, that **Untitled.sbd** is shown in bold font, and its contents now include the new STD.

5. Select the **cb_ex2.cat** file again.

The Cruise Control Logic STD is still listed, but its Current State is Moved.

6. Save **cb_ex2.cat** again.

A dialog asking you if you want to retain the overwritten (moved) file appears. This gives you the opportunity to keep the item that you moved to another file.

7. Answer No.

3.1.5 Overwriting Items in a File

In this section, [Example 3-5](#) illustrates what happens when you open a file that contains items with the same name as items already in the catalog. The first part of this example is just setup so that you can more easily follow the points being made.

Example 3-5 Overwriting Items in a File

1. Open the continuous automobile SuperBlock in an editor, and change the outputs to 3. Close the editor.

This SuperBlock now has 3 inputs and 3 outputs.

2. Open the continuous automobile w IA SuperBlock in an editor, and change the inputs to 2. Close the editor.

This SuperBlock now has 2 inputs and 2 outputs.

3. Select View→Update to update the Catalog Browser.

You would expect the two items that you directly modified to show that status, and they do. However, you might not expect the Discrete Cruise System to also be marked Modified. It is the parent SuperBlock to continuous automobile; therefore, it is also modified.

4. Go to the Catalog view, and delete all items associated with file **cb_ex2.cat** except the two that you modified directly.

5. Return to the File view, and save **cb_ex2.cat** to clean up what is in the File view.

Other than the Xmath partition, you now have two SuperBlocks in **cb_ex2.cat**, and both have been modified in a clearly identifiable manner from their original versions.

6. Select File→Open, and then select file **cb_ex1.cat** from your working directory.

Since items in the two files have duplicate names, you get a Warning dialog that first asks if you want to replace one of the current SuperBlocks.

7. Click Yes for the continuous automobile SuperBlock.
8. Click No for the continuous automobile w IA SuperBlock.

Notice that the **cb_ex2.cat** file remains selected in the File view. It still contains two SuperBlocks and the Xmath partition, but the SuperBlock that you replaced, as well as the Xmath partition, have their Current Status fields marked Overwritten.

9. Now select the file **cb_ex1.cat** in the File view.

Notice that continuous automobile w IA has its Current Status field marked Overwritten.

10. Select File→Save to save the file in its modified form.

The Retain Item dialog comes on view. It asks you if you want to retain the overwritten (original) version of continuous automobile w IA.

11. Answer No.

In the Contents view, notice that the version of this SuperBlock has 2 inputs and 2 outputs. In the File view, observe that this SuperBlock is associated with file **cb_ex2.cat**.

12. Bring up the Xmath Commands window.

The log area contains a log item concerning the replaced SuperBlock.

The important point to understand is that only one version of an item with a given name can exist in a SystemBuild model at one time. If you open two or more files with conflicting names, then SystemBuild asks you to select the one you want to use. Then the item in the file that you didn't use is overwritten. When you save the file that contained the element that was overwritten, you have the opportunity to save the original version of the element in that file.

3.1.6 Creating a New File

[Example 3-6](#) shows you how to create and populate a new file. In this section, we demonstrate the principle of one item with the same name in the model from a different perspective.

Example 3-6 **Creating a New File**

1. Select File→New→File.

From the File view, note that SystemBuild created **Untitled2.sbd**.

2. Select **Untitled2.sbd**.

Notice that the file is empty.

3. Select **cb_ex1.cat** again.

4. Select the Cruise_Control_Logic STD in the Contents view.

5. Click the Edit menu, and examine the active menu items.

Notice that Copy is not active.

6. Select Edit→Cut.

7. Select **Untitled2.sbd**.

8. Select Edit→Paste.

The **Cruise_Control_Logic** STD is shown as a **New** item in this file, and the file name is shown in bold, indicating that it needs to be saved.

9. Select **cb_ex1.cat** again.

The **Cruise_Control_Logic** STD is shown as a **Moved** item in this file, and the file name is shown in bold, indicating that it needs to be saved.

The **Untitled2.sbd** file—and others created the same way—functions just like **Untitled.sbd**. It is a place holder until you populate it explicitly. You cannot copy items to this file, however, because SystemBuild allows only one item with the same name in a model, and a copy would create two items with the same name. If you want a copy of an item in a file, copy the item directly, so that it has a different name, and then you can move the copy to the new file.



NOTE: At this point, we are formally done with using the **fmlearning** directory and its contents. Delete it if you wish.

3.1.7 Saving All Files

If you have several files open and have not moved or overwritten multiple elements within the files, you can use the File→Save All command, which simply saves all elements back into the files with which they are associated. In more complex situations, you might want to save each file individually.

3.1.8 Saving Xmath Partitions

You can save Xmath data at the level of a partition. If you are opening several files, the last file opened gets ownership of any Xmath partition that the files have in common, including Main. If you have variables with the same names, the last one loaded is used; for example, if two files each have a Main partition with a variable t , the first t loaded is overwritten. Otherwise, the data is merged. You receive no warnings from Xmath about this. Therefore, we encourage you to store Xmath data in partitions other than Main so that you can retain data that is appropriate to your models when you have more than one file open at once.

3.2 Configuration Management

Configuration management in SystemBuild is simply an extension of file management. Configuration management provides an interface to your configuration management tool through the Versioning menu on the Catalog Browser. Currently, SystemBuild supports the following tools:

- ClearCase[®]
- PVCS[®]
- Microsoft Visual SourceSafe[™] (Windows only)

In this section, we provide preparation details for using your CM tool in SystemBuild. Then we teach you how to use the interface to your tool, but we make no attempts to teach you how to use your tool; we assume that you know how to use and have a thorough understanding of the configuration tool that interfaces to SystemBuild. We all assume that you know what various operations, such as checkin and checkout, mean for the tool that you are using.

3.2.1 Preparing to Use a CM Tool with SystemBuild

If you plan to use the configuration management feature, we assume that you have one of the tools above installed on your computer and that it is functioning on your computer (see the *System Administrator's Guide* for your operating system). To use the CM tool with SystemBuild, a few additional steps are necessary.

SystemBuild uses the **Sysbld.ini** file found in **\$\$SYSBLD/etc** to resolve the CM tool to be used when you launch CM from within SystemBuild. More specifically, SystemBuild uses the entry called **CMToolConfig**. Its use is operating-system dependent; appropriate settings are provided below. See [Chapter 18, SystemBuild Customization](#), for additional information about using this file.

Windows Operating Systems

Set **CMToolConfig** to the string **"SCCI"**. SystemBuild figures out from the PC registry which CM tool to use and uses the correct one.

UNIX Operating Systems

Set **CMToolConfig** to the appropriate case-sensitive string:

- **"ClearCase.pl"**
- **"PVCSPCLI"**
- **"FileRW.pl"**

If you are using PVCS, you need to perform some additional functions.

To ensure that PVCS works on UNIX operating systems:

1. Ensure that the directory where PVCS is installed is in your path.
2. In your **.cshrc** file, **source** the **vm65cshrc** file, along with its path.

For example, your **.cshrc** file contain a line that looks like:

```
source ./pvcs/vm65cshrc
```

3. Make a copy of **Sysbld.ini**, and place it in your startup directory.
4. Change the **CMToolConfig** entry from **"ClearCase.pl"** to **"PVCSPCLI"**.
5. Before you start PVCS from SystemBuild, start PVCS standalone. Start a project, specify the directories in which the source files are located, and exit.

You have to perform this step for each project that you create in PVCS.

6. Start SystemBuild, and start CM from SystemBuild.

A dialog prompts you to specify the project database location. Once you specify the location, you can access those files and perform checkin and checkout operations directly from SystemBuild.

3.2.2 Getting Messages from Your CM Tool

Often a CM tool operation delivers a text message when the operation completes successfully or when an error occurs. When using CM within SystemBuild, messages are written to the log area of the Xmath Commands window. It's a good idea to check the log area after each operation.

3.2.3 Connecting to Your Configuration Management Tool

In all cases, you must connect to your CM tool before you can perform other CM operations through SystemBuild. Therefore, this is the only option initially active.

To start any configuration management activity through SystemBuild:

Select Versioning→Connect to CM.

If you are a PVCS user, a dialog that asks you to input the project database and the source location comes on view. You should be aware that files specified in the source location provided in this dialog work fine in terms of checking in and checking out within SystemBuild. Files not in the source location or its subdirectories show up as not under source control within SystemBuild.

At this point, the active items in the Versioning menu are Disconnect from CM and Launch CM Tool.

3.2.4 Opening a File

The next step is to open a file. The file should be either already under configuration management control or in a working directory from which it can be placed under control. In ClearCase, the file should be in a versioned object base (VOB); in PVCS or SourceSafe, it should be in your project's workfile location.



NOTE: If you are using Windows 2000 operating system, it has additional security measures. If you wish to open a file that someone else owns, you need to have **administrator** privileges.

To open a file:

Select File→Open, and then select the file that you want to view from the Open dialog.

The file is now listed in the File view. When you select the file, additional items on the Versioning menu are now active; these options depend upon whether or not you opened a file that is already under configuration management. The icon next to the file indicates the CM status of the file:



White background File not under configuration control



Gray background File under configuration control

either
of
above No check on icon File not checked out of CM system



Black check on icon File is checked out



Red check on icon File is checked out and reserved to you alone

3.2.5 Putting a File Under Configuration Control from SystemBuild

If you opened a file that is not under configuration management, the additional menu option that becomes active is Add to Source Control.

To put a new file under configuration management control:

1. Select the file that you wish to put under configuration control in the File view.
2. Select Versioning→Add to Source Control.

A configuration-tool-dependent dialog appears that allows you to accomplish this task.

3.2.6 Using Other Configuration Management Options

After you have opened a file under configuration control, the following additional menu options become active when you have that file selected in the File view:

- Get Latest Version
- Check Out
- Show History

See [Problems with Extension Names in Different Applications](#) below.

- Remove from Source Control
- Version Properties
- Refresh Status

Once you have a file checked out, you can make any changes to it that you wish. The following additional menu items become active:

- Check In
- Undo Check Out

3.2.7 Problems with Extension Names in Different Applications

On the Windows platform, file extensions are usually registered with certain applications that can be used to open these files (for example, `.doc` indicates that the file is a Word file, `.txt` means it is a text file, and so forth). If the file extension does not correspond to the expected file type, certain operations may result in unexpected behavior.

On certain PCs, files with the `.cat` file extension, the standard extension for Systembuild catalog files, are registered as security catalogs. Trying to view such files by their default viewers causes problems. When using Visual SourceSafe, this anomalous behavior is exhibited when you try to view a `.cat` file from within the History dialog. This behavior is intrinsic to certain Windows applications and is not controlled by MATRIX_X. Use caution when trying to open or view files using default Windows viewers when the file is different from the registered type.

3.2.8 Additional Resources

If the meaning of these topics is not clear, see online Help and/or consult your configuration management tool documentation because many of these items mean something different in each tool.

To get online Help for the Versioning menu:

1. From the Catalog Browser, select Help→Topics.
2. Scroll down the hypertext topics until you come to the Versioning menu, and then click it.

4

SuperBlocks

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

As discussed in [Chapter 2](#), the Catalog Browser manages the status and interrelationships of SuperBlocks, blocks, and all other objects in the catalog. You initiate each editing session from the Catalog Browser; from it you can create and edit a new SuperBlock or open an existing SuperBlock and modify it. This chapter focuses on how to create and edit SuperBlocks.

The major topics in this chapter are as follows:

- *SuperBlock Hierarchies*
- *Creating SuperBlocks*
- *Creating a SuperBlock Reference*
- *Renaming SuperBlocks*
- *Using File SuperBlocks*

4.1 SuperBlock Hierarchies

You can display up to 199 blocks 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 propagates to all references.

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, trigger, and 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 6](#) for more on SuperBlocks and timing.

The operation of SuperBlocks in hierarchies is illustrated in [Figure 4-1](#), which shows how SuperBlocks may be nested, one within another, each containing functional blocks.

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 4-1 shows the SuperBlock ubiquitous occurring at different levels in a hierarchy.

Figure 4-1 SuperBlock References in a SuperBlock Hierarchy

The screenshot shows the 'SystemBuild - Catalog Browser' interface. On the left is a tree view of the 'Main' catalog, showing a hierarchy: Main > Model > top > A1 > B1 > ubiquitous. Below the tree are 'CatalogView' and 'FileView' buttons. The main area is divided into two parts: a table of SuperBlock properties and two detailed block diagrams.

Name	CatType	Inputs	Outputs	Period	Skew	GroupId
A1	SB (Discrete)	1	1	0.1	0	0
A2	SB (Discrete)	1	1	0.1	0	0
B1	SB (Discrete)	1	1	0.1	0	0
top	SB (Discrete)	2	2	0.1	0	0
ubiquitous	SB (Discrete)	1	1	0.1	0	0

Discrete SuperBlock	Sample Period	Sample Skew	Inputs	Outputs	Enable Signal	GroupId
top	0.1	0.	2	2	Parent	0

The diagram for 'top' shows two sub-blocks: 'A1' (ID 3) and 'ubiquitous' (ID 2). 'A1' contains a gain block and a delay block. 'ubiquitous' contains two delay blocks. The diagram for 'B1' (ID 1) shows an input '2' entering a block that contains an 'ubiquitous' sub-block (ID 2), a 'sum' block, and a 'normal' block. The output of 'B1' is '2'.

4.2 Creating SuperBlocks

You can create SuperBlocks in several different ways, which are explained in this section.

4.2.1 Creating a New SuperBlock from the Catalog Browser

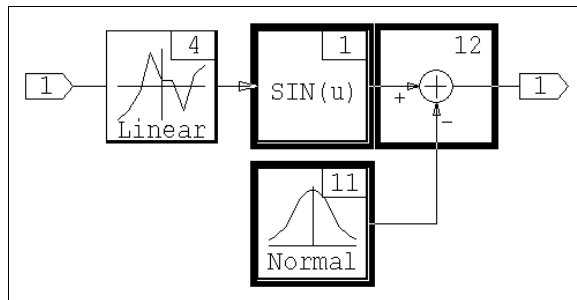
We discussed creating a new SuperBlock from the Catalog Browser in the previous chapter and include it here for completeness. See [2.5.3 Creating a New SuperBlock](#) for the detailed procedure.

4.2.2 Making a New SuperBlock from Existing Blocks

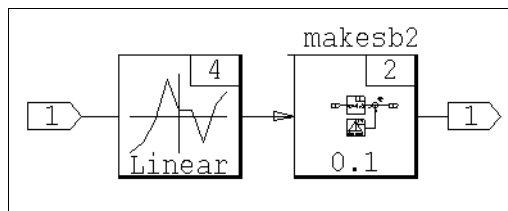
You can create a new SuperBlock by grouping existing functional blocks and SuperBlocks in the SuperBlock Editor.

To create a SuperBlock from existing blocks:

1. With the existing SuperBlock open in the SuperBlock Editor, select the blocks that you want to move to a subhierarchy (the children of the SuperBlock).



2. Select Edit→Make SuperBlock.



The new SuperBlock is given the default name `_makesb#`, where # is a system-supplied digit that ensures a unique name. The child SuperBlock has the timing properties of the current parent SuperBlock.



NOTE: By definition, the view of the SuperBlock that appears in the original diagram is a reference SuperBlock. Double-click this icon to bring up the SuperBlock itself in an editor window.

To ungroup the blocks:

1. Select the SuperBlock in an editor window.
2. Select Edit→Expand SuperBlock



NOTE: You cannot use the **Expand SuperBlock** command within a container block; for a description of container blocks, type **help container** in the Xmath command area.

To display the contents of the SuperBlock in the SuperBlock icon:

1. Select the SuperBlock.
2. Press s.

Pressing s a second time enlarges the contents. Pressing s another time returns the view to the SuperBlock icon.

4.2.3 Creating a Copy of a SuperBlock

Copying a SuperBlock creates a new SuperBlock too. You can create a copy of a SuperBlock in the SuperBlock catalog from the Catalog Browser or the editor.

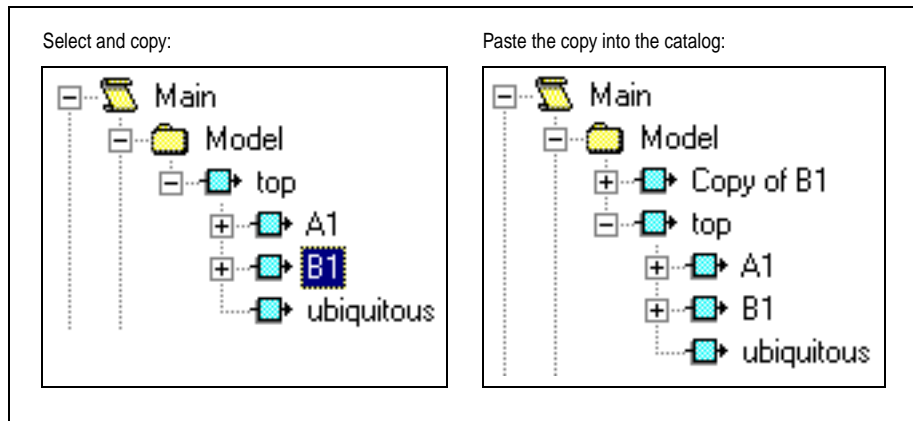
Creating a Copy with Copy and Paste

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

To copy a SuperBlock from the Catalog Browser:

1. Select an existing SuperBlock.
2. Right-click to raise the Shortcut 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 in the Model hierarchy. This process is illustrated below.



4. To rename the copy from the Catalog Browser, select it, and then select Edit→Rename.

The Rename dialog appears.

5. Input the name you wish to use, and click OK.

Notice that Rename all references is enabled by default in this dialog; usually you want to leave it enabled.

Creating a Copy by Modifying the SuperBlock Properties

You can edit a SuperBlock's properties if the SuperBlock is currently displayed in an editor.

To create a copy of a SuperBlock from the SuperBlock properties dialog:

1. Single-click the SuperBlock ID bar to raise the SuperBlock Properties dialog ([Figure 2-3](#), p.24).
2. Change the SuperBlock Name field in this dialog.

A copy of the current SuperBlock with the new name appears in the catalog. The original definition and any references to it are unaffected.

4.2.4 Defining a SuperBlock's Properties

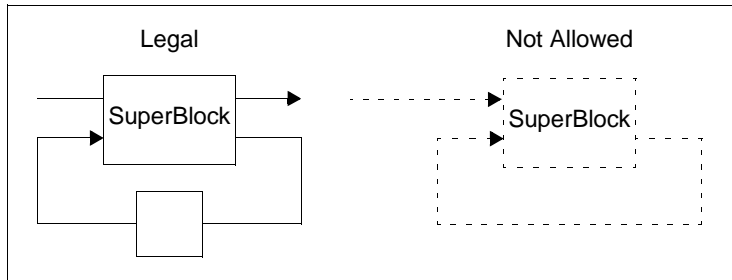
In this section, we discuss SuperBlock properties generally and then discuss the dialog that you use to define them.

SuperBlock Properties

All SuperBlocks have the following properties:

- A SuperBlock must have a unique name (within the current catalog). 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, State Transition Diagrams (STDs), or BetterState charts.
- 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 4-2](#)).

Figure 4-2 Legal and Illegal Connections



Using the SuperBlock Properties Dialog

After you create a SuperBlock, you must define its properties through the SuperBlock Properties dialog. If you create a block from existing blocks (see [4.2.2 Making a New SuperBlock from Existing Blocks](#)), you also create a reference to the SuperBlock in the original diagram. You must also define the properties of the reference; the SuperBlock Block dialog serves this purpose (see [Figure 4-4](#), p.62).

The SuperBlock Properties dialog first appears when you create a new SuperBlock from the Catalog Browser (see [2.5.3 Creating a New SuperBlock](#)). When you are editing a SuperBlock, you can raise this dialog by clicking the SuperBlock ID bar, the strip of SuperBlock information that appears below the Editor's toolbars and above the diagram work area, or by selecting File→SuperBlock Properties.

Refer to [Figure 2-3](#), p.24 as needed as we describe the SuperBlock Properties dialog.

The Name, Inputs, and Outputs fields are always visible. A SuperBlock must have a unique name that starts with an alpha character, is less than 32 characters long, and contains no punctuation characters (such as semicolons, periods, and so forth); if you use them, SystemBuild replaces them with underscores.

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 online Help for the SuperBlock Properties dialog.

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

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. See 6.1.1 Continuous SuperBlocks for additional information.
Discrete	SuperBlocks may run in discrete time, either as free-running or enabled. The Sample Period, Sample Skew, and Enable Signal fields are active for this option. If you specify an enable signal, the block executes only when the enable signal is asserted. See 6.1.2 Discrete SuperBlocks .
Trigger	The SuperBlock is triggered for execution (one-shot) by a specified trigger signal (see 6.1.3 Triggered SuperBlocks).
Procedure	Procedure SuperBlocks allow users to implement standalone 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 6.1.4 Procedure SuperBlocks).
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 you select Enter Local Label Names, you can input local names on other tabs, and they are used. If you are specifying a top-level SuperBlock, you must select Enter Local Label Names if you wish to specify labels on 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 8.1.2 Assigning SuperBlocks to Additional Subsystems for an in-depth discussion.

Code Tab

The Code tab is enabled when the SuperBlock type is procedure, and the procedure class is macro. You specify your macro in the editing area of this tab. You can type directly in the editing area, or you can select an editor from the Editors pulldown menu, and click Launch to use the editor directly. When you exit the editor, the contents appear in the editing area of the tab.

To change the default text editor, see [18.1.3](#), p.480.

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 appears in the block diagram if you select Enter Local Label Names in the Input Naming field on the Attributes tab. You can use !"#\$%'*+,-./>=<?@^ and European characters (ASCII codes up to 128).

If you select Inherit Higher-Level Names in the Input Naming field on the Attributes tab, the parent labels are propagated; this field is inactive for this case.

In either case, the labels appear in the diagram

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. Do not use punctuation characters (!"#\$%&'*+,-./>=<?@^());[]\ '{}~); invalid characters are mapped to underscores.

This name is associated with the signal in the AutoCode code listing; it has no impact on the block diagram. See the *AutoCode User's Guide*.

- Input Data Type** This field allows you to assign a data type for each input. Because data types 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 procedure.

To assign a data type on UNIX, click in the field; a menu of types appears. On Windows, the types are in a combo box; press the triangle beside the field to display the menu. See [5.5.5 Specifying Data Types](#).
- Input Radix** This field is used solely for fixed-point data. If the Input Data Type is set to a signed or unsigned type, you can specify an input radix. See [16. Fixed-Point Arithmetic](#).
- Input UserType** If you want to specify a user-defined type for this input, specify it in this field. See [16.5 User-Defined Data Types \(UserTypes\)](#). As with datatypes, the type may not be relevant for functional blocks within the diagram. However, procedure SuperBlocks lower in the hierarchy can inherit these values.
- 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 is global or local in the generated code. See the *AutoCode 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 occurs there as well.

Information on 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, and then follow the hypertext link to the Comment tab discussion. To change the default comment editor, see [18.1.4 Comment Editor](#).

4.3 Creating a SuperBlock Reference

A SuperBlock definition is saved as part of the catalog hierarchy. References are local instances of a SuperBlock. 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 4-4](#); compare this to the SuperBlock Properties dialog in [Figure 2-3](#), p.24. If the SuperBlock is currently open in the editor, you can edit the SuperBlock properties; any changes are global and affect the model at every instance,

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



CAUTION: If a SuperBlock definition is removed from the catalog, references to it 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 ([4.4 Renaming SuperBlocks](#)) 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.

4.3.1 *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 Contents view and dropping it in the editor where you are editing the SuperBlock in which the reference is to exist.

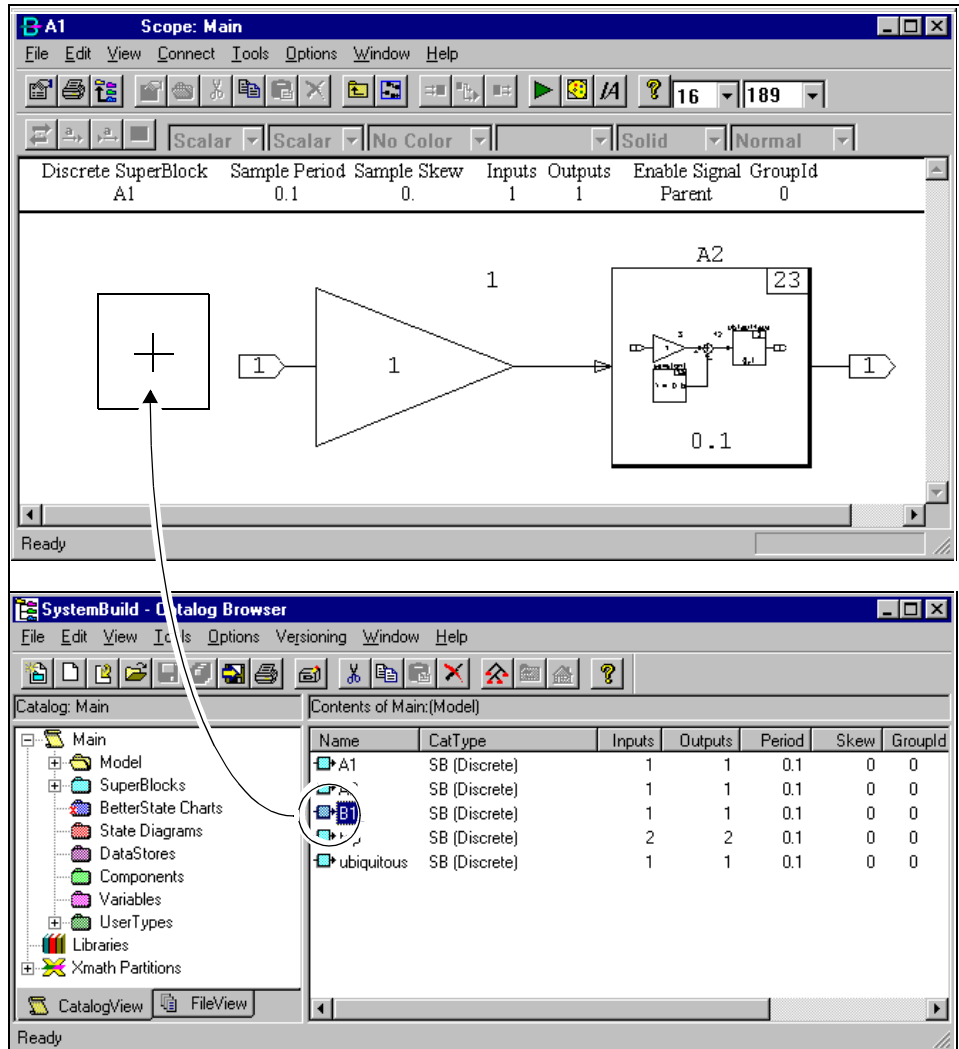
To create a reference from the Catalog Browser:

1. Click the SuperBlocks folder in the Catalog view to view all the SuperBlocks in the current catalog in the Contents view.
2. From the Contents view (the right side), select a SuperBlock icon and drag it into the SuperBlock being currently edited.

On UNIX, select and drag with the middle mouse button; on Windows platforms, use the left mouse button.

When you drop the reference, a SuperBlock block appears in the editor, as shown in [Figure 4-3](#).

Figure 4-3 Drag from the Catalog Browser Contents View, Drop in the Editor



4.3.2 Creating a Reference from the Editor

To create a reference from the editor:

1. Double-click in empty space to raise the Palette Browser, and select the SuperBlocks palette.
2. From the SuperBlocks palette, drag the SuperBlock icon into the editor. (On UNIX, use the middle button; on Windows, use the left button.)

The SuperBlock block icon shows an undefined sample period by default.

3. Select the SuperBlock block icon, and press Return (or Enter).

The SuperBlock Block dialog (see [Figure 4-4](#)) comes on view.

4. If you want to reference an existing SuperBlock, supply the Name of a SuperBlock present in the catalog, its Instance Name (optional), any other parameters that you wish to enter, and press Return (or Enter).

The block's timing attributes are taken from the referenced SuperBlock; the SuperBlock reference name is the name of the SuperBlock, followed by the name of the instance (if any) in parentheses.

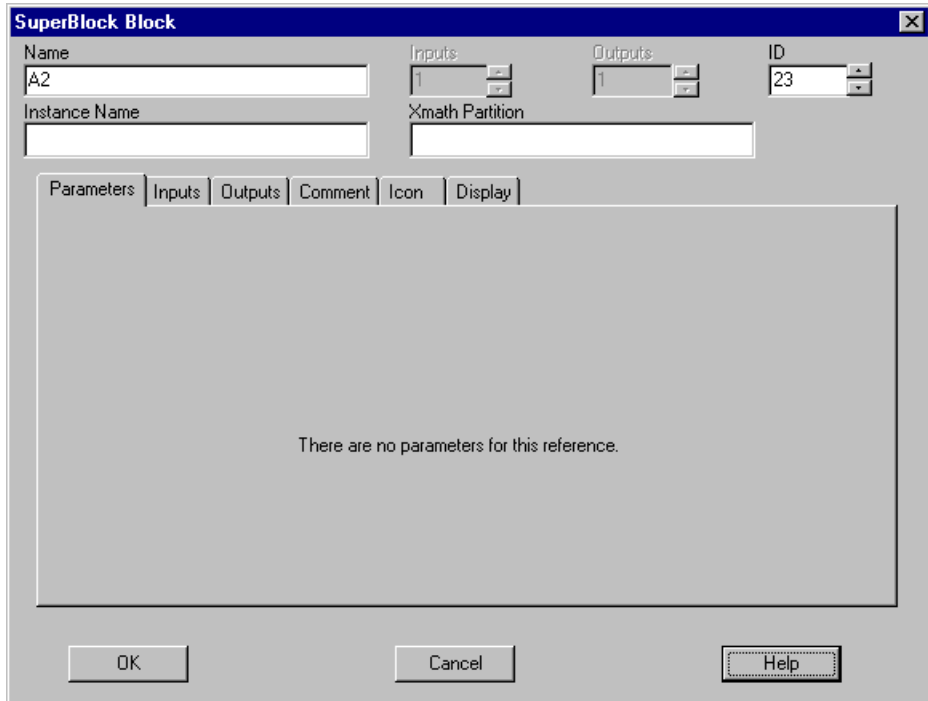
5. If you want to leave the SuperBlock reference undefined, put at least one primitive block in it as a placeholder block. (It doesn't have to be connected to anything.)

4.3.3 Defining the Reference SuperBlock's Properties

The SuperBlock Block dialog, also referred to as the SuperBlock Reference or Instance dialog, controls information specific to an instance of a SuperBlock within a block diagram.

To view the SuperBlock Block dialog (see [Figure 4-4](#)), select a SuperBlock icon and press Return (or Enter).

Figure 4-4 **SuperBlock Block Dialog that References an Existing SuperBlock**



The fields below appear on dialog at all times:

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>. You may use spaces or underscore characters as word separators in SuperBlock names. Use caution, however, in employing spaces because processes other than <code>MATRIX_x</code> might 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 appears 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 contains the default name 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	<p>The block ID of the SuperBlock block icon.</p>
Instance Name	<p>You can supply a local name for this reference. If an instance name is used, it is appended to the SuperBlock name field and surrounded by parentheses in the diagram. For example, if you give a reference to a SuperBlock named <code>System</code> the instance name <code>variation1</code>, the name above the SuperBlock reference is <code>System (variation1)</code>.</p>
Xmath Partition	<p>Use this field to specify the name of an Xmath partition that this SuperBlock instance is to use for loading and saving data.</p>

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 [5.5.2 Block Dialog Fields](#)).

The single exception is found on the Display tab. If enabled, the Propagate Label checkbox dictates that labels from the parent SuperBlock are propagated to child SuperBlocks in the subhierarchy and that signals passing through the SuperBlocks pass the reference's labels to the blocks that follow.

4.4 Renaming SuperBlocks

The procedures for renaming SuperBlocks are simple and intuitive. The consequences of renaming SuperBlocks is the primary subject of this section.

4.4.1 Replacing a SuperBlock with Catalog Browser Rename

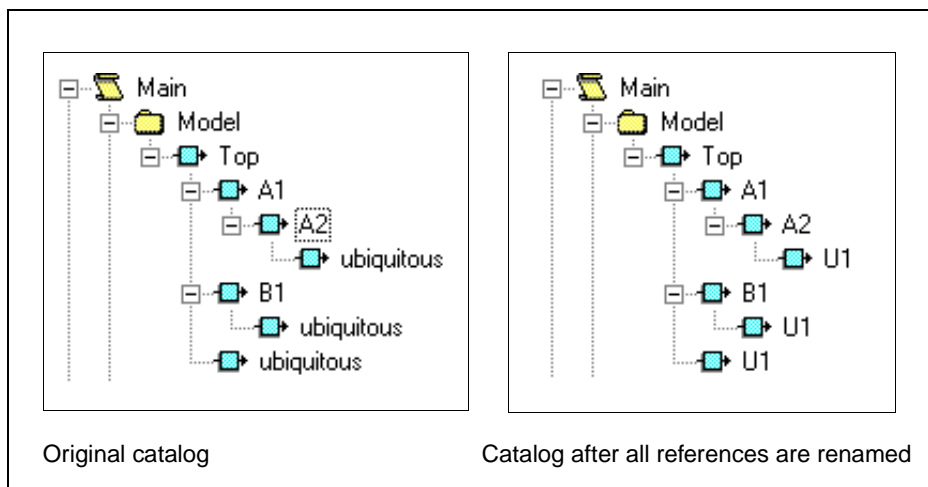
To rename a SuperBlock from the Catalog Browser:

1. Select the SuperBlock.
2. Select Edit→Rename, or right-click to raise the Shortcut menu, and select Rename.

Because the Catalog Browser operates on a global level, renaming a SuperBlock from the Catalog Browser results in the destruction of the original SuperBlock.

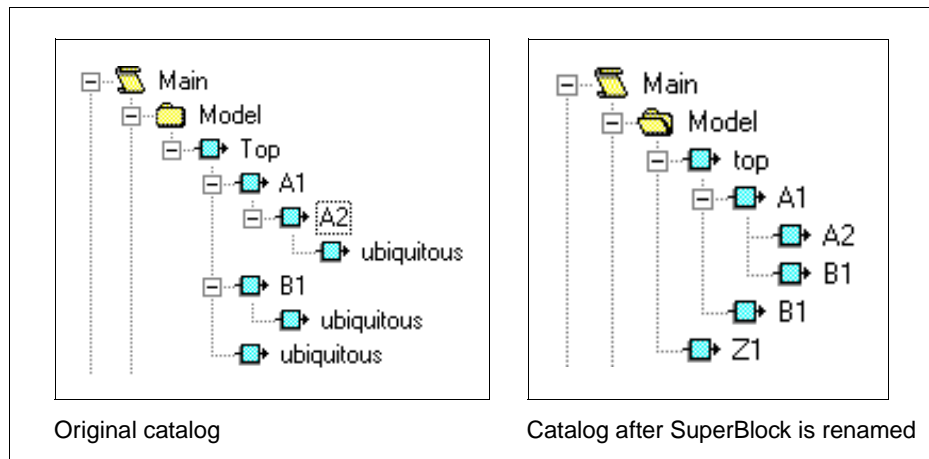
- If you rename a SuperBlock and enable Rename All References in the Rename dialog, SystemBuild gives the specified name to the SuperBlock definition and all references to it. The original SuperBlock is removed from the catalog. [Figure 4-5](#) shows this concept, where the SuperBlock ubiquitous has been renamed to U1.

Figure 4-5 Renaming a SuperBlock and All References



- 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 do not appear in the catalog. Figure 4-6 shows the effect of ubiquitous being renamed to Z1. Although the undefined SuperBlocks do not appear in the catalog, they remain in the block diagram, and they are still named ubiquitous.

Figure 4-6 Renaming a SuperBlock Without Renaming References



4.4.2 Renaming SuperBlocks in the Editor

The cases below deal with renaming a SuperBlock and renaming a SuperBlock reference from the editor.

- If you rename a SuperBlock from the SuperBlock Properties dialog (see Figure 2-3, p.24), you create a copy of the original SuperBlock with a new name (see *Creating a Copy by Modifying the SuperBlock Properties*, p.53). Assuming that the renamed SuperBlock is properly defined, when you update the catalog, it becomes a top-level SuperBlock. The original SuperBlock that you renamed and its position in the hierarchy are unaffected.



NOTE: This method of renaming provides no opportunity to rename all references to the SuperBlock.

- When you change the Name field from a reference's SuperBlock Block dialog (see Figure 4-4, p.62), only that block is renamed. If the renamed SuperBlock

exists in the catalog, the new name appears in the catalog hierarchy when you update the catalog. If the renamed SuperBlock does not exist in the catalog, the renamed SuperBlock does not appear in the catalog until you define it properly.

4.5 Using 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 File SuperBlock icons in the SystemBuild diagrams.

To create File SuperBlocks:

1. Identify parts of your system that can be logically grouped as standalone entities.

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 entity into a separate file.

Both binary and ASCII files are acceptable. Extending the concept from [Step 1](#), your model might contain the following files: **engine.cat**, **transmission.cat**, **drive_train.cat**, and **suspension.cat**.

To reference SuperBlocks from a file in other parts of the your model:

1. From the Xmath Commands window, enter an **setsbdefault** command using the **sblibs** keyword to list the files that you have created. Continuing with the example above, for example:

```
setsbdefault,{sblibs="engine.cat transmission.cat drive_train.cat  
suspension.cat"}
```

2. In the SystemBuild Editor, build the diagram using references to File SuperBlocks:
 - a. In the Catalog Browser, expand the Libraries folder.

You can see the list of filenames specified by the **sblibs** keyword in the Catalog view.



NOTE: The libraries are static, and therefore the Delete, Cut, and Copy options are not enabled for them.

- b. Select a file that contains a SuperBlock you want to reference.

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.)

The SuperBlock is then represented by a File SuperBlock icon in your diagram; you cannot open this SuperBlock in your model.

When you simulate your model, the simulator resolves the File SuperBlock references by sequentially searching each library in the order specified in the **sblibs** keyword. During the analysis phase, messages appear in the log area of the Xmath Commands window indicating which library supplies each SuperBlock where applicable.

A SuperBlock in a library may reference SuperBlocks within itself or another library, but SystemBuild does not resolve SuperBlock references by searching backward through the files listed with **sblibs**. 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 go to the SuperBlock in the library appearing *earliest* in the **sblibs** list ([Step 1](#)).



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

```
UNIX:          setenv SBLIBS "engine.cat transmission.cat"
Windows:      set SBLIBS="engine.cat transmission.cat"
```


5

Blocks

The SuperBlock Editor operates on a SuperBlock that has been loaded into the Catalog Browser (see [2.1 Loading Data](#)) or created using one of the methods in [4.2 Creating SuperBlocks](#). 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 ([4.3 Creating a SuperBlock Reference](#)), connections between blocks, external connections between blocks and the SuperBlock's inputs and outputs.

In this chapter, we show you how to create the contents of a SuperBlock. This is an iterative process that differs somewhat for every user on every diagram. Most people seem to drag all the blocks that they think they need off the palettes initially; perform some type of logical arrangement of those blocks; provide the basic properties of the blocks—name, inputs, outputs, states (if applicable), and ID; and then connect them. Finally, they go back and fill in all the details of the block definitions and perform some additional modifications to the diagram. This chapter follows this pattern with the major topics presented below:

- [Types of Blocks](#)
- [Creating a Model with Blocks](#)
- [Assigning the Basic Properties to Your Block](#)
- [Connecting Blocks](#)
- [Modifying Block Diagram Appearance](#)

There is nothing, however, that dictates that you work in the manner described, for SystemBuild is a flexible tool and accommodates many styles of working.

This chapter contains simple examples that demonstrate how to define, connect, and modify blocks and the block diagram.

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

5.1 Types of Blocks

SystemBuild contains many types of blocks. For convenience, however, we group these blocks into two categories: basic functional blocks and special blocks. The basic functional blocks perform dedicated computations, whereas the special blocks 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.

In the following paragraphs, we define basic functional blocks and the various types of special blocks.

5.1.1 Basic Functional Blocks

Most of the blocks are basic functional blocks, and you will work with them most of the time. Each of these blocks performs a dedicated computational function. One example is the Gain block, which multiplies the inputs by some fixed value and passes the result to the output channel.

5.1.2 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 SuperBlocks listed in the block dialog's Code tab are executed. See the Condition topic in online Help 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 a 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 to TRUE, the blocks in the IfThenElse block container are executed. See the IfThenElse topic in online Help for additional information.

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

5.1.3 Repetitive Execution (While, Break Blocks)

The While block has a similar structure to an IfThenElse block. 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.

5.1.4 Terminating Execution (Stop Block)

The Stop block stops execution of a model if any of its inputs are TRUE.

5.1.5 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 (WriteVariable, ReadVariable) 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 are also executed before blocks on the right of the bar if the child SuperBlock is in the same subsystem as the SuperBlock containing the Sequencer. See the Sequencer topic in online Help for more information and examples.

5.2 Creating a Model with Blocks

Once you create a SuperBlock, you can access the SuperBlock Editor (2.5.2 *Opening a SuperBlock or BetterState Chart in an Editor*). Then you can 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 7](#) and online Help.

A single Palette Browser supports all editor windows. You can access the Palette Browser from the editor by any of the following methods:

1. Double-click in an open area of the editor window.
2. Click the Palette toolbar button  .
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 Main palette; it can also display custom blocks and palettes, as described in [Chapter 20](#). For a full description of palette capabilities, including loading and deleting palettes, see the Palette Browser topic in online Help.

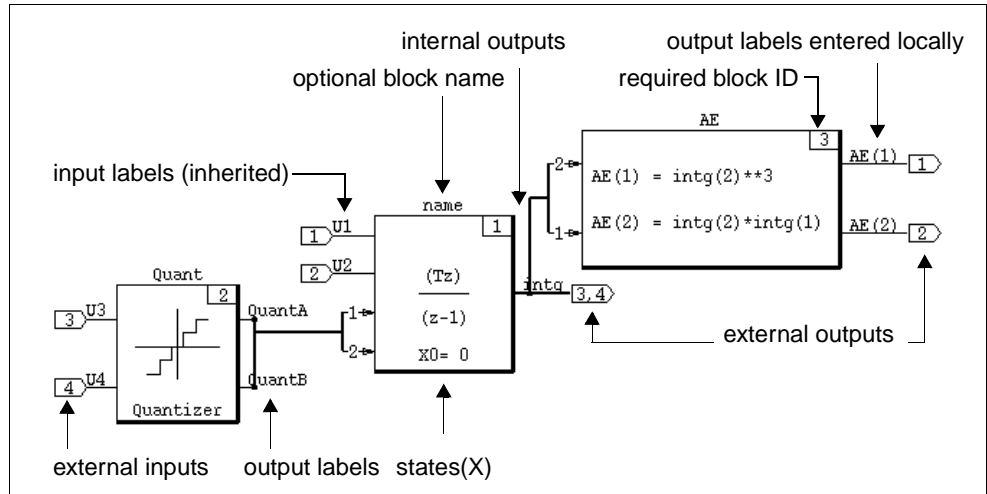
Under some circumstances you will not be able to drag a given block into the Editor. Typically this occurs when you try to instantiate a strictly continuous or strictly discrete block in a SuperBlock with conflicting timing. You must change the SuperBlock type or select another block.

5.3 Assigning the Basic Properties to Your Block

Once you instantiate a block, you define it from the block dialog. [Figure 5-1](#) illustrates some block properties that can be displayed in the editor. Typically, you define the basic properties and then connect the blocks (see [5.4 Connecting Blocks](#)) before you provide the details of the block definitions (see [Defining Your Block](#) on p.82).


In this section, we tell you how to raise a block dialog. Although you can have a number of SuperBlock Editors open at once, you can only have one block dialog open at any given time. We discuss the basic properties common to most block dialogs and then the buttons that are also common to the dialogs.

Figure 5-1 **Block Properties Visible in the Editor**



5.3.1 Raising the Block Dialog

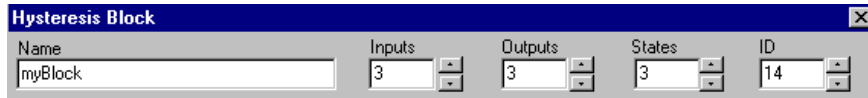
There are three ways to raise the block dialog:

- Position the cursor over a block, and then press Return (or Enter).
- Select a block, and click the Block Properties toolbar button .
- Select a block, and then select Edit→Block Properties.

5.3.2 Defining the Basic Properties

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 (see [Figure 5-2](#)).

Figure 5-2 Example of Basic Properties Display in Block Dialog



- 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 is given a default value by the system, but you can change it to any unused number in the range [1: 199].

Fields that do not apply may be missing or inactive (grayed out).

5.3.3 Using the Common Buttons

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

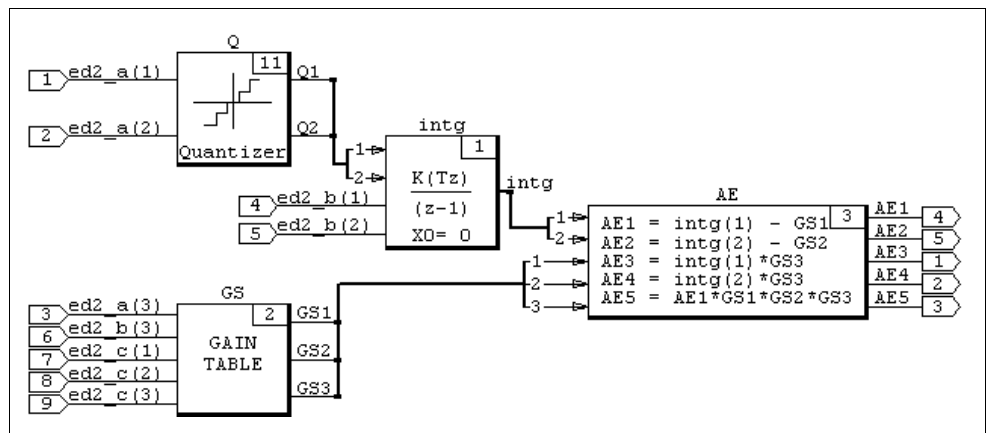
5.4 Connecting Blocks

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 that each pin is drawn separately. The number of pins and any related labels are displayed in the Editor.

In [Figure 5-3](#), the labels make it easy to follow the signals inherited from the SuperBlock ed2.

Figure 5-3 **Scalar Connections**



5.4.1 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 4-2](#), p.54).

- 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 produce a warning; unconnected inputs are assigned to zero.
- 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.

5.4.2 Creating Connections

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

Creating a Simple Connection

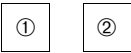
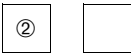
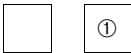
A *simple* connection connects the first available from block output signal to the first available to block input pin; it then connects the next available output signal to the next available pin until all pins are connected.

To perform a simple connection, middle-click (3-button mouse) or Ctrl-right-click (2-button mouse) as shown in [Table 5-1](#).



NOTE: If there is more than one available input pin, you must *hold* the mouse button down momentarily on the to block to immediately connect to the first available pin; otherwise, SystemBuild raises the Connection Editor.

Table 5-1 **Simple Connections**

Connection	From ①	To ②	Click
Block to block	Source	Destination	
External inputs to block	Open space to left of destination	Destination	① 
Block to external outputs	Source	Open space to right of source	 ②

Using the Connect Menu and the Toolbar Buttons

The connection process is selection-activated—that is, at least one block must be selected to activate Connect menu items.

The Connect menu items are described below; interspersed with the descriptions are alternative methods of performing the same actions using the toolbar buttons. The Connect menu is also described in online Help for the Editor window; click ? or Help→Topics to raise the Help window in the SuperBlock Editor.

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

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




Select one block; then click the external input toolbar button.

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

Mouse: Middle-click (3-button mouse) or Ctrl-right-click (2-button mouse) in from object, then in to object.



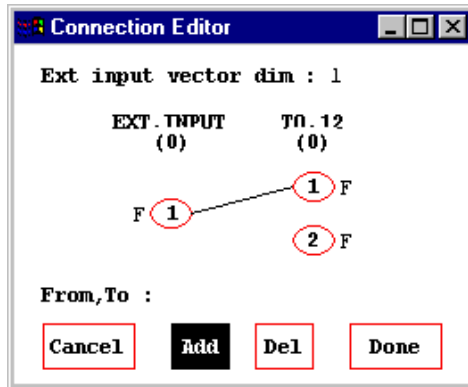
Select two blocks; then click the internal connection toolbar button.

- Outputs** Connect the selected block to an external output. This menu item is only active when one block is selected.
- Mouse:** Middle-click (3-button mouse) or Ctrl-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 Ctrl-right-click (2-button mouse), and change the routing by dragging the marker to a new location.
- Mouse:** Middle-click (3-button mouse) or Ctrl-right-click (2-button mouse) in an open area, holding down the button until transition markers appear.
- Automatic Routing** Automatically route connections to selected block(s). The SuperBlock Editor lays 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.

5.4.3 Using the Connection Editor

If you middle-click (3-button mouse) or Ctrl-right-click (2-button mouse) in a From block and then in a To block where the connections are not simple, the Connection Editor comes on view (see [Figure 5-4](#)).

Figure 5-4 Connection Editor



If external inputs or outputs are involved, the dimension of the External Input/Output vector is displayed at the top of the dialog; this field is editable. The FROM (or EXT. INPUT) and TO (or EXT. OUTPUT) fields show the block IDs of the blocks being connected. The buttons in the middle indicate the possible connections; the connecting lines indicate the connections, if any. The four buttons at the bottom of the Connection Editor—Cancel, Add, Del and Done—control the actions in the dialog.

Creating Connections

The following items provide the rules and the operations in this dialog:

- Only one Connection Editor is allowed at any given time, regardless of the number of SuperBlock Editors open.
- The Connection Editor interface is unique in SystemBuild in that you must select the action (Add or Del) *before* you choose the operand (the input signals).
- To add the maximum number of one-to-one connections in one operation, double-click Add, and the Connection Editor makes 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, SystemBuild connects them.

- 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 fields. 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 connects 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 the Del button to enable deletion. Click any pin on the source (left side) or destination (right side), and SystemBuild erases its connections.
- Double-click 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, and press Return (or Enter).
- If you add to the number of external inputs, the additional inputs are added to the end.
- If you are adding to the number of external outputs, you can add them to the end by typing the new number of outputs and pressing Return (or Enter).

- You can insert outputs into the existing output list as follows:
 - a. Type the new number of outputs (but do *not* press Return (or Enter)).
 - b. Click between any two output destination pins; the additional pins are inserted at that location. Any previous information is 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 block. By default, vectored labels are compressed. When you have a large number of pins shown in scalar mode, a scroll box appears 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 it right to display the bottom of the list.



NOTE: When you move connections out of alignment, they are visually truncated; however, the connection is not affected.

Exiting the Connection Editor

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

5.5 Defining Your Block

This section focuses on elements common to all blocks, for example, tabs, fields, and how to use them. The major topics are as follows:

- [Block Dialog Overview](#)
- [Block Dialog Fields](#)
- [Entering Matrix Data in Block Dialogs](#)
- [Specifying Labels and Names](#)
- [Specifying Data Types](#)

5.5.1 Block Dialog Overview

This section discusses graphical elements (controls) found on SuperBlock and block dialogs. All interactive dialogs have common graphical elements that provide clues about how to enter data. 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 calculate a value. You can also choose settings by selecting dialog-specific menu items or enabling or disabling checkboxes.

In SystemBuild, dialogs adapt to the block environment and settings whenever possible. For example, a visual indication lets you know if a field is not applicable in the current environment, as shown in Figure 5-5.

Figure 5-5 Motif and Windows Versions of the SuperBlock Dialog.

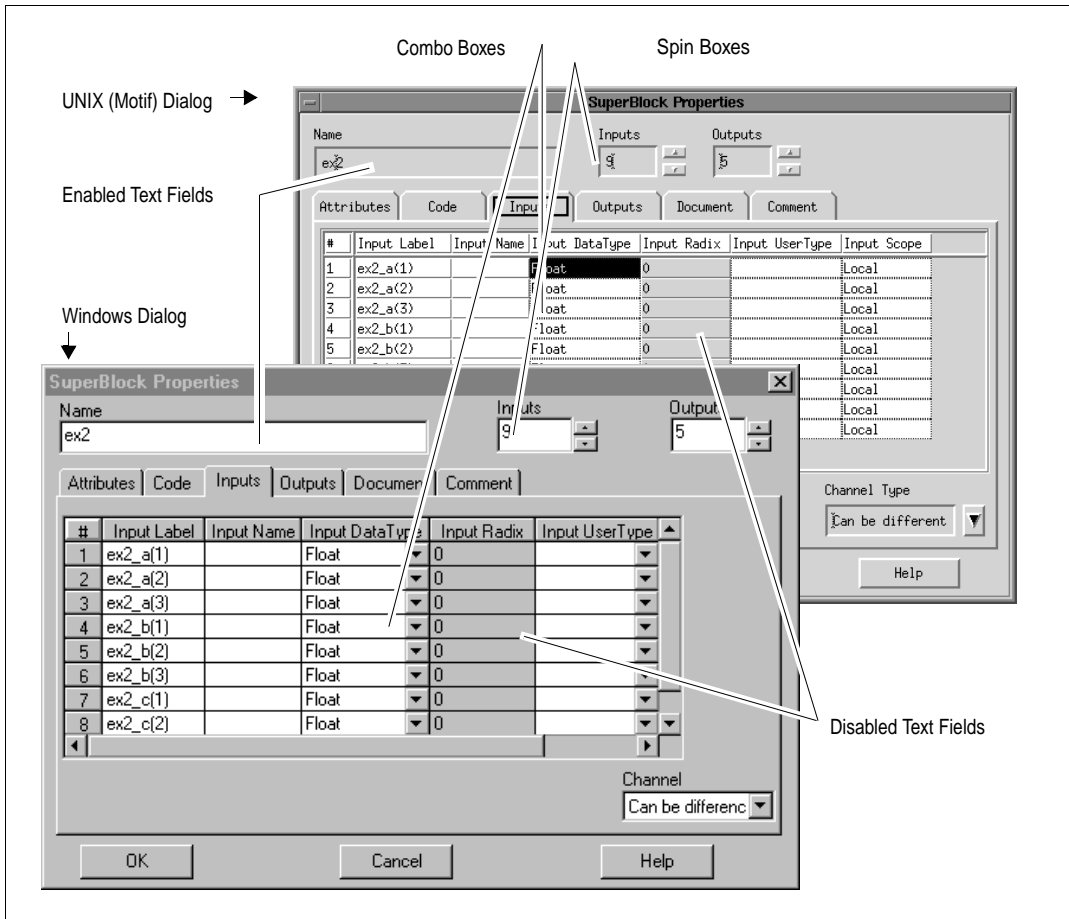


Figure 5-5 also demonstrates the principle of consistency between fields. Here, nine inputs have been specified, so the dialog supplies nine fields for each column on the Inputs tab. SystemBuild also synchronizes the Outputs tab fields and the Document tab fields to match the inputs and outputs specified. Each block has its own parameters and its own dependencies between fields; these are discussed in online Help for each individual block.

Differences Across Platforms

Dialog contents are the same across platforms, but there are small differences in the implementation. These differences are summarized in [Table 5-2](#).

Table 5-2 **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 the field to raise a menu, drag to highlight an item, and then release.	Click the down arrow to raise a menu. Click 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 it. To rapidly advance the counter, click the arrow and hold down the mouse button; release when the desired value is reached.

Dialog Navigation and Shortcuts

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.

5.5.2 Block Dialog Fields

In this section, we discuss the dialogs in more detail. Specifically, we provide information by tabs that is common to most of the block dialogs. The nine possible tabs—[Parameters](#), [Code](#), [Inputs](#), [Outputs](#), [States](#), [Document](#), [Comment](#), [Icon](#), and [Display](#)—are discussed in this section. The behavior of all the tabs is consistent among blocks with the exception of the Parameters tab.

You can find specifics for any given block in online Help with emphasis on the Parameters tab.

Parameters

The Parameters tab fields vary widely among blocks, but, in general, the settings and values used to operate on an input signal are found on the Parameters tab.

Data or settings information for the block's internal function is supplied here. Depending on the block, Parameters tab fields may require strings, vectors, or matrices. Numeric input may have specific data type requirements. It is common to have block-specific interdependencies between fields. For example, in [Figure 5-6](#), the number of elements in the Breakpoints field (three) determines the number of rows in the gain matrix.

Figure 5-6 Gain Scheduler Block Dialog, Parameter Tab View

Parameter	Value	% variable
Break Points	1.0, 3.0, 7.0	
Gain Matrices	(...)	

Gain Matrices			Rows: 3	Columns: 9	Value: -1.0			
	1	2	3	4	5	6	7	8
1	6.0	2.0	3.0	1.0	1.0	0	7.0	-1.0
2	3.0	4.0	4.0	7.0	6.0	6.0	0	-1.0
3	3.0	4.0	9.0	4.0	8.0	6.0	8.0	1.0

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, While, and FuzzyLogic. The syntax for instructions placed in the Code tab is block specific, so be sure to view online Help for each block before using the Code tab.

To change the text editor from the Code tab, see [18.1.3 Default Text Editor](#) on p.480.

Inputs

For primitive blocks, this tab displays the Input Name and Input Signal fields.

Input names are specified at the SuperBlock level as Input Labels (when Enter Local Label Names is selected for the Input Naming field on the Attributes tab) and then propagated to SuperBlocks or blocks further down the hierarchy. Output labels, by contrast, are specified within each elementary block.

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 appears in code generated by AutoCode for the block or SuperBlock only; it does not appear in block diagrams.

See the *AutoCode Reference* manual.

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.

Input Face Use this field to set the face for one or more inputs. Choices are Left, Bottom, Right, and Top; choices are relative to the default position of the block. The faces change with the block when you rotate it or change directions. This field is informed by and informs the Input Face field on the Display tab. If you use a different face for one or more inputs, the Input Face field changes to Mixed.

NOTE: If you want all inputs on the same face, you can set them all from the Input Face field on the Display tab. If you want to have different settings, then you must set them on the Inputs tab.

See [5.5.4 Specifying Labels and Names](#) for tips on entering labels or names.

Outputs

The Outputs tab is used to label and format block output data. You can display and modify several output-related fields, including Output Label, Output Name, and Data Type. 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 can be 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. You can use !"#\$%'+,-./>=<?@^ and European characters (ASCII codes up to 128). If Show Output Labels is enabled on the Display tab, the label is displayed at the corresponding output pin location in the block diagram. This label also appears in DocumentIt documentation.
Output Name	To increase traceability in generated code, AutoCode users can specify a name for the output signal. Do not use punctuation characters (!"#\$%&'*+,-./>=<?@^() ;[]\`{}~); invalid characters are mapped to underscores. This name never appears in the editor.
Output Data Type	This field allows you to assign a data type for each output. For more detail on data types, see 5.5.5 Specifying Data Types and 16. Fixed-Point Arithmetic .
Radix	If the Output data type is fixed point, this field may be enabled. See 16. Fixed-Point Arithmetic .
Output UserType	If you want to specify a user-defined type (UserType) for this output, specify it in this field. See 16.5 User-Defined Data Types (UserTypes) .
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 channel scoped as Global. This address has no simulation effect. See the <i>AutoCode Reference</i> manual for more information.
Overflow Protection	The overflow protection setting is ignored during simulation (forced to On) but, it significantly impacts code generation. If overflow protection is Off, AutoCode C does not generate overflow protection code for the current block (assuming the output data type is fixed point). This option is not supported for Ada.

States

The States tab is only present in dynamic blocks.

State Name	Optional. If entered, this name appears in the code generated by AutoCode.
State Comment	Optional text string.

Document

The Document tab is only present in blocks with outputs. Its fields are used to annotate generated code; DocumentIt also extracts these fields. 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 appear in the code generated by AutoCode.

Comment

The Comment tab is available on almost all blocks; it allows you to include a multiline 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 the Comments checkbox is enabled, 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.

Alternatively, you can create a comment in a supported document editor. To enter a comment in an editor, select an editor from the Editor combo box, and then press the Launch button to raise it. Enter the comment, and then close the editor when you are finished. The text you created is displayed in the Comment tab. If the editor is Word (Windows only), 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 are dropped.

To change the default text editor, see [18.1.3 Default Text Editor](#) on p.480. To add or remove applications from the Editor combo box, see [18.1.4 Comment Editor](#) on p.481.

Edit User Parameters

When User Parameters is enabled, the text area is used to display or change user parameters. You can create user parameters in several ways:

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

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

Icon

The Icon tab is present for all blocks. [19. Custom Icons](#), explains how you can define or reference an icon using this tab.

Display

The Display tab appears for every block.

Input Pins/Output Pins

These fields determine how connections are 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 [5.5.4 Specifying Labels and Names](#)), displaying one line per group. If Show Input Labels and/or Show Output 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 [5.4 Connecting Blocks](#). Note that you can also change the display modes via combo boxes in the SuperBlock Editor toolbar.

Input Face/Output Face

You can choose which face of the block that you want input pins connected and which face you want output pins connected: Left, Bottom, Right, or Top. You can also position the input pins on different faces; however, if you select Mixed from the Display tab, you get an error. If you want to use mixed faces, then set them on the Inputs tab; SystemBuild then sets this field to Mixed. The input and output faces are relative to the default position of the block (Rotation Normal and Direction Forward); when you rotate or change directions of the block, the input and output faces change with the block rotation and direction.

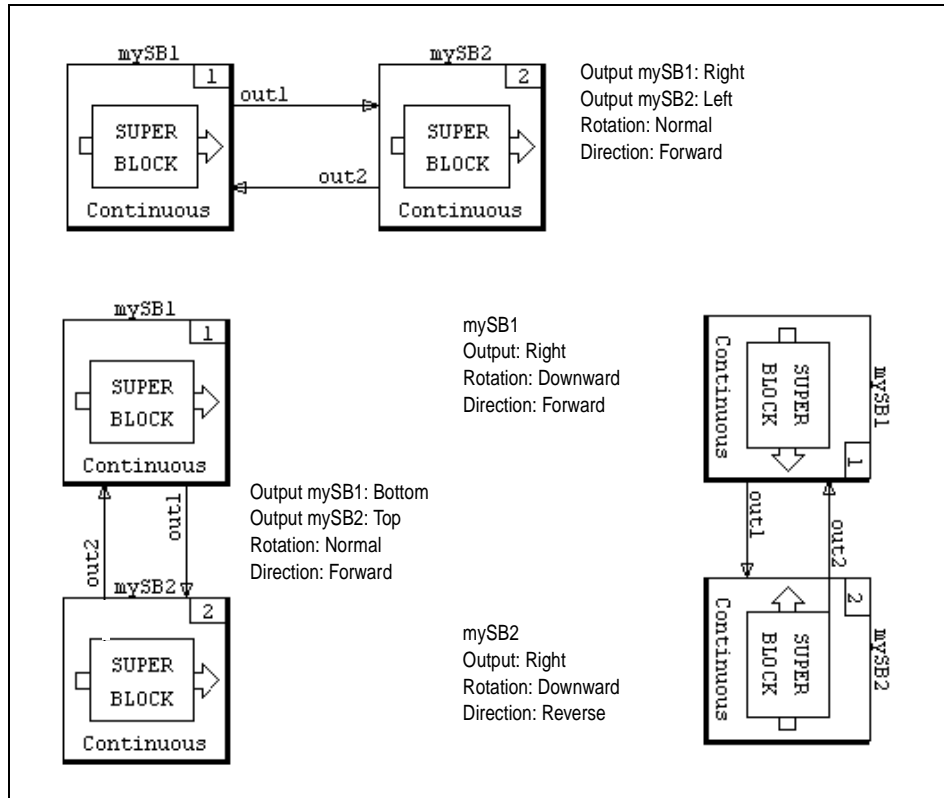
You can put both input and output pins on the same face. To minimize connection crossings, the following rules apply:

- Input pins are first on the left and bottom faces.
- Output pins are first on the right and top faces.

Figure 5-7 illustrates these points.

You can cycle through the input faces by positioning the cursor over the block and pressing CTRL+u repeatedly. Likewise, you can use CTRL+y to cycle through the output faces.

Figure 5-7 Inputs and Outputs on the Same Block Face



Show Input Signals

If Show Input Signals is enabled, input signal labels are shown in the diagram. This checkbox also enables the display of external input labels.

Any inherited input signal labels are 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.

Show Output Labels

If Show Output Labels is enabled, output signal labels are shown in the diagram. This checkbox also enables the display of external input labels.



NOTE: External input labels are not shown if *both* Show Input Signals and Show Output Labels are disabled.

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 enabled, 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 appear in the reference block's output labels. You cannot make modifications to reference block output labels when Propagate Labels is enabled.

When Propagate Labels is not enabled, 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 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 5-3](#).

Table 5-3 **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 [18.2](#), p.484).

Icon Type

The Icon Type field controls the icon appearance; you can select Special, Alternate, User, Simple, or Custom. Special is the default selection and usually displays a descriptive word or picture.



NOTE: Not all blocks have different views for each option.

You can also change the icon type at the block diagram level. When you select a block, its icon type is displayed on the SuperBlock Editor toolbar, and you can change it from the combo box.

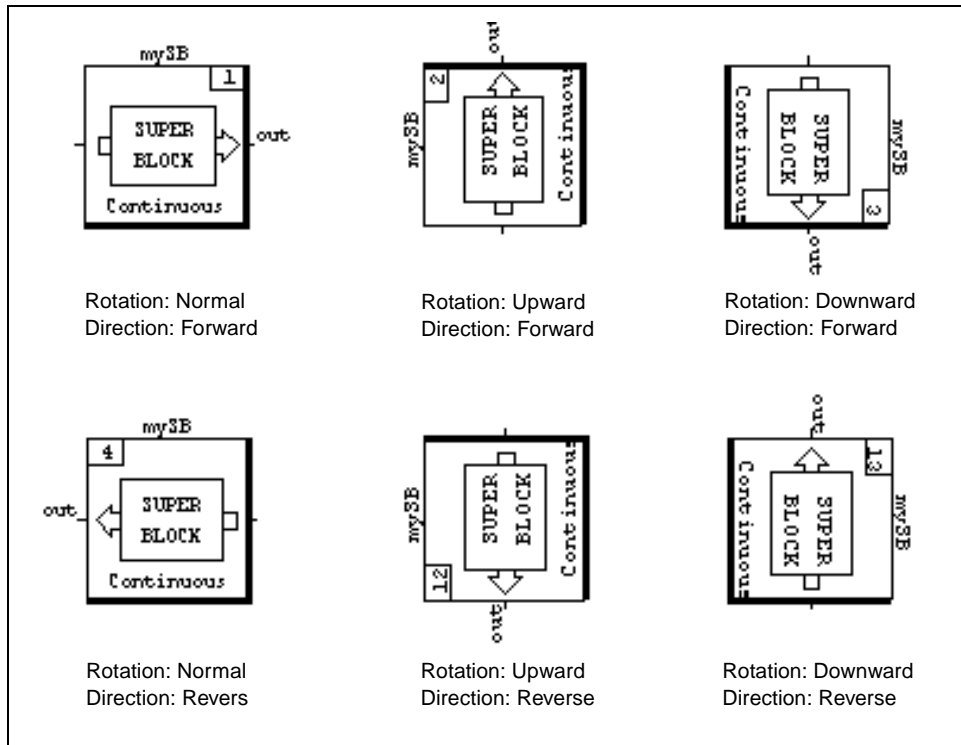
Alternatively, with the block selected, press **s** repeatedly to cycle through icon types.

Rotation and Direction

You can use a combination of the Rotation and Direction fields to configure a block (its icon, id, name, input pins, and output pins) in six ways. Rotation is Normal, Upward, or Downward; press **CTRL+r** repeatedly to rotate through the settings

counter-clockwise. Direction is Forward or Reverse; press CTRL+h or f to change direction. These settings are illustrated in [Figure 5-8](#).

Figure 5-8 **Rotation and Direction Settings for Blocks**



NOTE: This design prevents your positioning a block upside down.

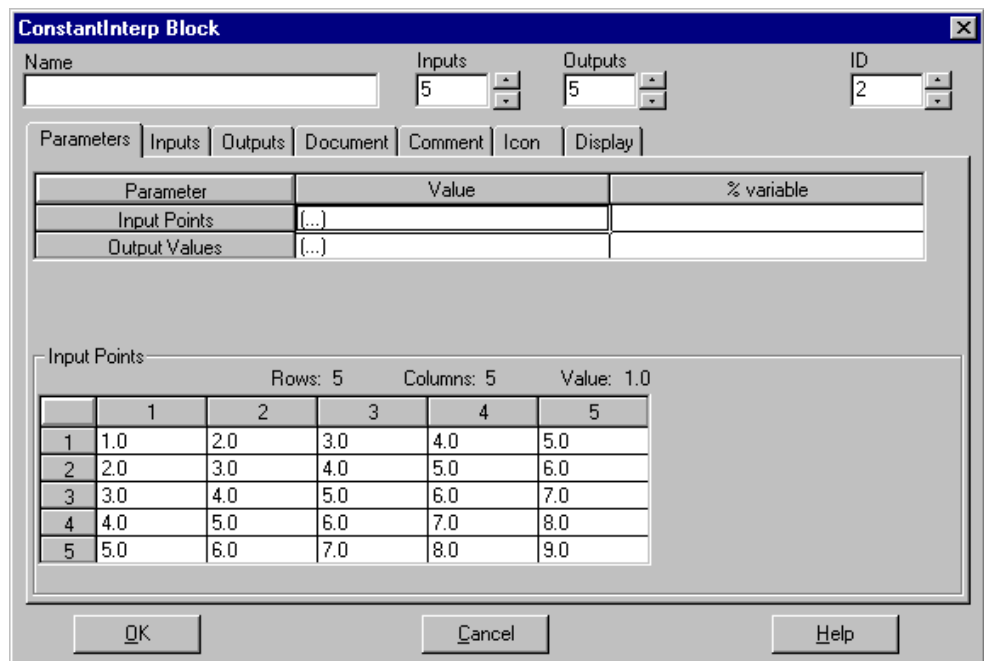
Name Location

You can place the name of a block at the Left, Bottom, Right, or Top of the block. The name location is relative to the default position of the block (Rotation Normal and Direction Forward); when you rotate or change directions of the block, the name location changes with the block rotation and direction.

5.5.3 Entering Matrix Data in Block Dialogs

For most blocks, numerical data is entered via a block's parameters tab; depending on the block, you can enter a scalar, vector, or a matrix. 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, states, or other block-specific considerations. Figure 5-9 shows a dialog with a two matrix fields: Input Points and Output Values. Note that both show the placeholder (...) in the input field.

Figure 5-9 Matrix Editor Field in Dialog



If applicable, the block dialog provides a simple matrix editor. You can enter data directly into the parameter fields, or you can use the matrix editor.

Invoking the Matrix Editor

To invoke the matrix editor, click the field name, or click in the field itself. The dimension of the matrix displayed below varies according to the selected field.

Entering a Matrix

To enter data, specify a matrix in the parameter field, or type the values into the matrix itself. When you are using the parameter field, you erase or overwrite the (...). If the matrix input is accepted, the placeholder reappears; if there are problems with the input, your expression remains in the field, giving you an opportunity to edit it.

You specify matrices in the same manner as Xmath data: enclosed in square brackets, with commas separating column elements, and semicolons separating rows; for example, [11,12;21,22].

In addition to vector/matrix notation, you can use Xmath expressions to supply matrix input. For example, given a ConstantInterp block with five inputs and outputs, as shown in [Figure 5-9](#), the following are acceptable inputs that you can type 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 the entry of illegal matrices. In the case above, a decreasing matrix is illegal for 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 might 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: krone(1:5,[1:5]')

5.5.4 Specifying Labels and Names

Blocks are connected such that the output of one block becomes the input of another. Exceptions are external inputs, which we cover below. Although SystemBuild allows you to show input signals, the default settings use the output labels from one block as the input of another block without repeating the labels.

Note the following definitions:

- A *label* is defined at the source of a signal and may be changed locally.
- A *signal* is assumed to originate elsewhere and can only be changed at the origin.


As shown [Figure 5-3](#), p.75, 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 display output labels:

1. Open the receiving block's block dialog, and bring the Display tab on view.
2. Enable the Show Outputs Label checkbox.

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.

Alternatively, you can use shortcuts:

1. Select the block.
2. Perform one of the following actions:
 - Press l (lower-case L).
 - Click the Output Labels On/Off toolbar button .

Specifying SuperBlock External Input Labels

External inputs to a SuperBlock represent the first place where a signal is visible and may be labeled in the SuperBlock.

By default, SuperBlocks have the Input Naming field set to Inherit Higher-Level Names; this disables the Input Label field on the Inputs tab, prohibiting local changes. You can replace the Input Label values with labels that are local to this SuperBlock and (optionally) the hierarchy below it.

To change the external labels on a SuperBlock:

1. Raise the SuperBlock Properties dialog.
2. In the Input Naming field on the Attributes tab, select Enter Local Label Names.
3. Go to the Inputs tab, and specify the local input labels.

Propagating Labels in a Hierarchy

To propagate the local signals down the hierarchy:

1. Double-click in the diagram to open each SuperBlock.



NOTE: SystemBuild has been intentionally designed so that you must double-click to open the SuperBlocks in the hierarchy for label propagation to work.

2. Raise the SuperBlock Block dialog.
3. Go to the Display tab, and enable Show Output Labels and Propagate Labels.

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

Creating Sequential Names for Vectors and 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* manual.

Vectoring Signal Labels or Names

You can use vectoring to generate unique labels or names for vectors. The syntax is:

```
string(ns:nf)
```

where *string* is the constant part of the name (must be entered first), *ns* is the starting number, and *nf* is the finishing number for the vector. The parentheses are required. For example, **lab(1:8)** produces lab1, lab2, ... lab8.

- Descending sequences are not supported.
- If a vector longer than the list is specified, a vector of labels is 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 5-1](#)).

Example 5-1 Vectoring Labels

1. In the Catalog Browser, select File→New→SuperBlock.
The SuperBlock Properties dialog appears.
2. Name the SuperBlock ex2, and make the Type Discrete.
3. Specify 9 inputs and 5 outputs.
4. In the Input Naming field, select Enter Local Label Names.
Because this is a top-level SuperBlock, it has no parent from which to inherit labels.
5. Go to the Inputs tab.
6. Click in the first row of the Input Label field, and type **ex2_a(1:3)**; then press Return (or Enter).

- Click in the fourth row of the Input Label field, and type **ex_2b(1:3)**. In the seventh row, type **ex_c(1:3)**. Press Return (or Enter).

Your dialog should now resemble one of the dialogs in [Figure 5-5](#), p.83.

Using Matrices for Signal Labels or Names

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

```
string(nsr:nfr, nsc:nfc)
```

where *string* is the constant part of the name (it must be entered first), *nsr* and *nsc* are the starting row and column number, and *nfr* and *nfc* are the finishing row and column numbers for the matrix. The parentheses are required, and the entries produced are emitted in row-major order. The total number of entries produced is $(nfr - nsr + 1) * (nfc - nsc + 1)$. For example, **foo(1:2,1:2)** produces the entries **foo(1,1)**, **foo(1,2)**, **foo(2,1)**, and **foo(2,2)**.

You can produce the same result with the following alternate syntax:

```
string[ROWSxCOLS]
```

In the above syntax, *string* is the root of the name and 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)**, **foo(2,2)**, and **foo(2,3)**. In this case, the matrix dimension **[2x3]** is displayed in the editor (assuming that labels are showing). You can view the individual element coordinates in the block dialog or from the Connection Editor.

Rules for Matrix Labelling

Either syntax is only reliable when $nsr = nsc = 1$ —that is, you must generate a matrix starting at the first row and column positions. If you start at a location other than (1,1), the automatically generated labels are “mangled.” You can 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 are not recognized as matrix elements. One exception exists: if the dimension of the matrix was properly specified at an earlier time, the matrix structure is recognized, but the labels are still mangled.

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 are mangled—they do not represent matrix elements. (You get foo_3_1, foo_3_2, foo_4_1, foo_4_2.) If you delete all the output labels (again) and go to the first output label box and type **foo(1:2,1:2)**, when you go to the fifth output label box and type **foo(3:4,1:2)**, you get the expected (true) matrix element entries.

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

Example

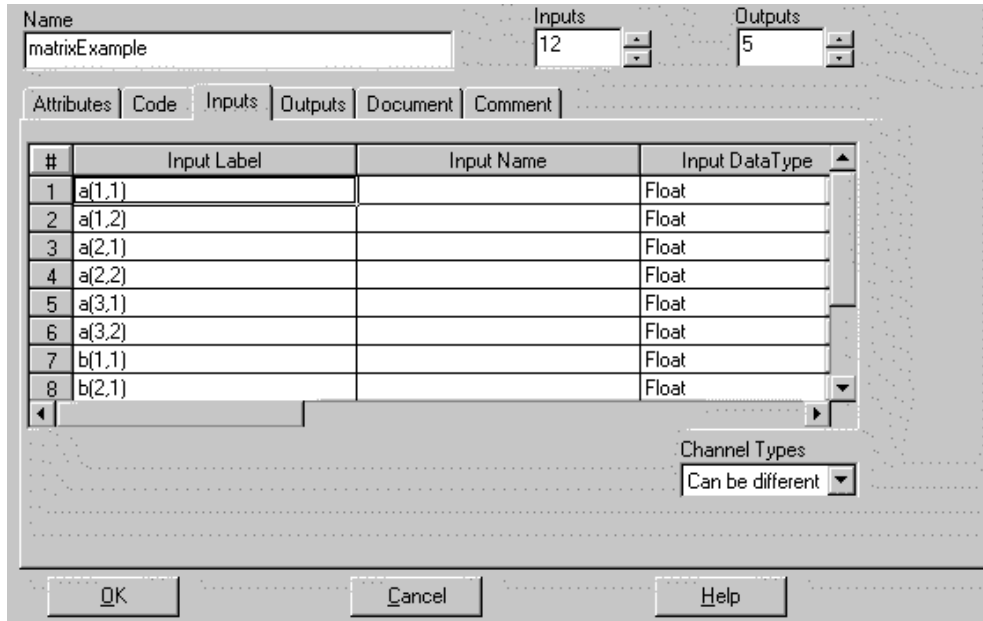
You can see how to use matrices by working through [Example 5-2](#).

Example 5-2 Using Matrices to Name Superblock Input Labels

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. In the Input Naming field, select Enter Local Label Names.
Because this is a top-level SuperBlock, it has no parent from which to inherit labels.
5. Go to the Inputs tab.
6. Click in the first row of the Input Label field, and type **a(1:3,1:2)**; then press Return (or Enter).
7. Click in the seventh row, and type **b(1:5,1:1)**.

The resulting Input tab appears in [Figure 5-10](#).

Figure 5-10 Inputs Tab with Labels Specified as Matrices



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 a vectoring or 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:

- | | |
|---------------------------------|--|
| <code>string(ns:nf)</code> | Starting from the cursor location, create unique names; existing names are overwritten. |
| <code>string(ns:nf)&</code> | Insert new labels starting at this location, and displace ensuing labels; some labels might drop off the bottom of the list. |

<code>string(nsr:nfr, nsc:nfc)</code>	Starting from the cursor location, create unique names: existing names are overwritten. If results are not part of a complete matrix, <i>string</i> is mangled and array notation is not used.
<code>string(nsr:nfr, nsc:nfc)&</code>	Insert new labels starting at this location, and displace ensuing labels; some labels might drop off the bottom of the list. If results are not part of a complete matrix, <i>string</i> is mangled and array notation is not used.
<code>(1:n)</code>	Starting at the cursor location, erase consecutive labels, where <i>n</i> is the number of fields you want erased.
<code>(1:n)&</code>	Starting at the cursor location, insert <i>n</i> empty fields; ensuing labels are displaced but preserved.
<code>(ns:nf)</code>	With your cursor in cell <i>ns</i> , erase all fields through cell <i>nf</i> .
<code>(ns:nf)&</code>	With your cursor in cell <i>ns</i> , erase all fields through cell <i>nf</i> and displace all ensuing labels.
<code>&d</code>	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.
<code>&dn</code>	<i>n</i> is a number indicating how many consecutive labels to remove. For example, <code>&d5</code> deletes five labels including the current one. The remaining labels are advanced to fill up the space.

5.5.5 Specifying Data Types

SystemBuild and AutoCode provide you with a rich collection of built-in data types for modeling, simulation, and code generation. All data type conflicts must be resolved before you can generate code.

The SystemBuild data type feature provided allows you to:

- Create models that may require a mix of different data types. You specify data types in the editor while defining individual blocks and SuperBlocks.

- Perform an automatic block-by-block check of the compatibility of output data types with input data types in a model (see the **typecheck** keyword in [Data Type Checking](#)).
- Simulate models using the appropriate arithmetic behavior when mixed data types are present (see the **fixpt** keyword in [Data Type Checking](#)).
- Automatically generate code with proper type declarations for the model.

The SystemBuild Editor sets data types for four classes of data:

- SuperBlock external inputs
- Block outputs
- Block states
- Block parameters

The data types for the SuperBlock external inputs are set explicitly in the SuperBlock Editor via the Inputs tab of the SuperBlock Properties dialog. The primitive block outputs are set explicitly on the Outputs tab of the primitive block dialog.

The block parameter and state data type rules usually depend on the data types of the inputs or outputs of the block and do not require your setting them explicitly. However, these data types may be affected by what you set on the Input tab of the SuperBlock Properties and the Outputs tab of the block dialogs. For example, the rule for the Quantization block requires that the data type of the Resolution(s) parameter be the same as the output. If you specify Integer as the Output Data Type on the Outputs tab of the block dialog, then the parameter Resolution(s) must also be an integer. This rule also applies to any parameter variables (%Variables) that are used for Resolution(s).

The default output and parameter data type for SuperBlocks and functional blocks is **Float**.

Traditional Data Types

The available set of data types is composed of the following traditional types:

- **Float**
- **Integer**
- **Logical**
- **Fixed-point**

The set of fixed-point data types includes more than 300 distinct members. Each fixed-point type is uniquely specified as a signed/unsigned type with two additional attributes: word length and radix position. Following the common microprocessor architectures, a fixed-point data type may have a word length of 8, 16, or 32 bits. The radix position is restricted to a value between -16 and 48. For a detailed description of the fixed-point arithmetic feature, see [16. Fixed-Point Arithmetic](#).

The rest of this chapter deals strictly with the non-fixed-point method of data type checking. Depending on what is specified in the SystemBuild model, AutoCode declares variables with one of the following data types:

Floating point — Data with real values is the default type for inputs and outputs. In Ada and C code, this type is referred to as **RT_FLOAT**.

Integer — Data specified by 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 — Data with two values is referred to as **BOOLEAN** or **RT_BOOLEAN** in Ada and C.

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

Data Type Checking

All data type conflicts must be resolved before you can generate code. To aid you in correctly matching the various data types, the Connection Editor shows data types for all inputs and outputs. There is also a complete list of the floating, integer, and Boolean data type rules for inputs, outputs, and states for each primitive block in [Table 5-5](#), p.109 and a list of blocks for which fixed-point arithmetic is supported in [Table 16-1](#), p.424, along with data type rules for these blocks. For parameter rules for each block, refer to online Help.

The TypeConversion block is provided to help with some of your data type mismatches. It accepts a vector of a given type and converts it to an identically dimensioned vector of the type you specify. 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 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 specifying a fixed-point data type on an external input causes the data on that input to be presented in the specified data type.

Data types 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 data types during the simulation analysis phase. You can use the **typecheck** keyword in the **analyze()**, **sim()**, **simout()**, **creatertf()**, **autocode()**, or **documentit()** functions.

Before the RTF is generated, type checking is performed for the entire model. Any inconsistencies detected in the model produce error messages pointing to the block or blocks that contain the conflicting data types.

- The **fixpt** keyword enables fixed-point arithmetic, which supports mixed data types. By combining 8, 16, and 32-bit types with signed and unsigned properties, more than 300 data types can be created. The powerful **fixpt** keyword propagates and performs checking on all data types encountered in the model. The keyword **fixpt** is provided as an option for the **sim()** function for this specific purpose.

[Table 5-2](#) summarizes the simulation behavior when different combinations of **typecheck** and **fixpt** are used. See [8.5.2 Showing and Setting Keyword Default Options](#) on [p.190](#) 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 data types as **Float**.

Table 5-4 **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

Example: How Typecheck Affects Simulation Results

[Example 5-3](#) shows how the `typecheck` keyword affects simulation results.

Example 5-3 How the Typecheck Keyword Affects Simulation Results

1. Go to the Xmath Commands window and type:

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

The catalog is 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 data types have been preserved.

4. Call the `sim()` function again without typechecking:

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

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

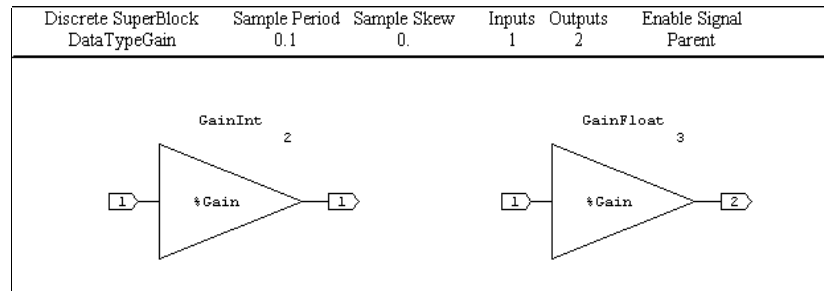
Example: Resolving Data Type Mismatching

[Example 5-4](#) demonstrates how to resolve an input/output data type mismatch.

Example 5-4 Data Type Mismatch with Gain Blocks

Consider the SuperBlock, `Data TypeGain`, shown in [Figure 5-11](#).

Figure 5-11 Data Type Mismatch Model



Check the model for errors using the **analyze()** function:

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

The keyword **typecheck** instructs **analyze()** to include data type consistency checking as part of the analysis. As a result of the data type checking, 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 in [Table 5-5](#), the input data type must be the same as the output data type for Gain blocks. The block GainFloat breaks this rule because its output data type is float, which does not match the input data type (inherited from the external input), which is integer.

To correct the problem, go to the Output tab of the GainFloat block dialog, and change the Output data type to integer. Run the **analyze()** command again, and the analysis completes with no errors.

Rules for Data Type Usage

The [Table 5-5](#) lists the blocks and their data type rules. Blocks that share common rules are grouped together. If not otherwise stated, all input and output channels must be the same data type. Footnotes for each group of blocks contain specific exceptions to the general rules; when a footnote marker appears, you can find the corresponding footnote on [p. 112](#).

Table 5-5 Legal Data Types for Each Block (Except Fixed-Point Data Types)

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 ^a	TRUE = 1, FALSE = 0.	NA
SpringMassDamper StateSpace NumDen Pole Zero ComplexPoleZero Integrator Hysteresis LimitedIntegrator PIDController UserCode Fuzzy Logic	Must be float	Must be float	Must be float

Table 5-5 **Legal Data Types for Each Block (Except Fixed-Point Data Types)** (Continued)

Block Type	Legal Inputs	Legal Outputs	Legal States
SquareRoot Logarithm Exponential SignedSquare Root Sin Cosin Atant2 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 ^b	Integer or float	NA
Waveform PulseTrain SquareWave Step	NA	Integer or float ^c	NA
SinWave UniformRandomGener ator NormalRandomGenera tor	NA	Must be float	NA
LogicalExpression	Integer or float ^a	TRUE = 1, FALSE = 0.	NA

Table 5-5 **Legal Data Types for Each Block (Except Fixed-Point Data Types)** (Continued)

Block Type	Legal Inputs	Legal Outputs	Legal States
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 Preload	Same as output	Integer or float	NA
WriteVariable	Must be float ^b	NA	NA
ReadVariable	NA	Must be float ^b	NA
Constant	N/A	Integer, float, or logical	N/A
ScalarGain, MatrixTranspose, MatrixMultiply, RightMultiply, LeftMultiply	Same as output	Integer or float	N/A
MatrixInverse, MatLeftDivide, MatRightDivide	Float only	Float only	N/A

- a. Each input and output channel can be a different data type, 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.

- b. Variable rules: The variables to which data is written may be float or integer. If bit addressing is used, the variable must be integer.
- c. The parameters related to the **Time Variable** (for simulation) are always floating point, while variables related to the **Output Variable** match the output data type in the generated code.

5.6 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, type **help shortcuts** from the Xmath command area.
- 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's viewing region.
 - Zoom changes the image size without changing the dimensions of block diagram elements
 - Normal restores the default Zoom (%100).

Some of these options are also available on the Edit toolbar, which appears below the menu bar. For a full description of the View menu items and the toolbar buttons, select Help→Topics in the SuperBlock Editor window.

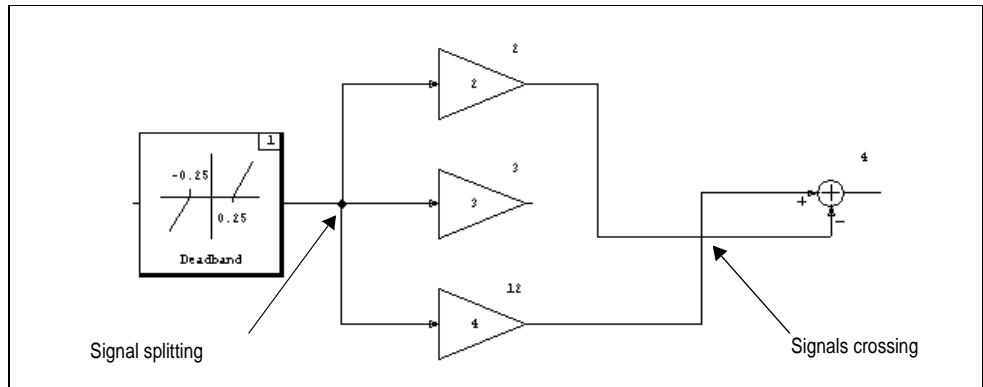
- Panning is a useful option for viewing large models that extend beyond the screen. Hold down the right mouse button in empty space, and drag the cursor in the direction that allows you to see the hidden portions of the model.
- The Display toolbar, which appears just above the work area, offers easy access to block attribute settings. See online Help for a description of each tool/toolbar button.
- Certain algebraic blocks have display icons that can appear with their input connections in any of several locations. These include summers, multiplications, dividers, and so forth. To toggle the icon display, select the

block and press *s* repeatedly, or change the type using the Icon Type combo box on the toolbar.

5.6.1 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, and so forth. The SuperBlock Editor puts a solid circle on the diagram where signals either come together or separate (see Figure 5-12). This eliminates confusion with signals crossing in a busy diagram.

Figure 5-12 Diagrammatic Difference in Signals Splitting and Crossing



The results of automatic routing are good for most diagrams, but you might need to use manual routing—in conjunction with block rotation, direction, and changing the face of input pins, output pins, and the name—to adjust the routing for complex diagrams.

There are two ways to enable manual routing:

- Select Connect→Manual Routing.
For this to work, no block or connection objects can be selected.
- Middle-click on a hidden handle on a connection.

Clicking in the middle of a horizontal connection is often successful.

When manual routing is enabled, square handles appear on the connection lines. Use the middle-mouse button to drag a handle to a new location. While changing the routing, you may have difficulty “lining up” connections. In limited

circumstances, it might be helpful to select Options→Snap to disable it. 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.

5.6.2 Improving the Appearance of a Cluttered Diagram

Figure 5-13 shows an example of an organized, but cluttered diagram. All elements are shown at normal size, and block 3 is too narrow to display the algebraic equations. Below are the steps used to improve the appearance of this diagram.

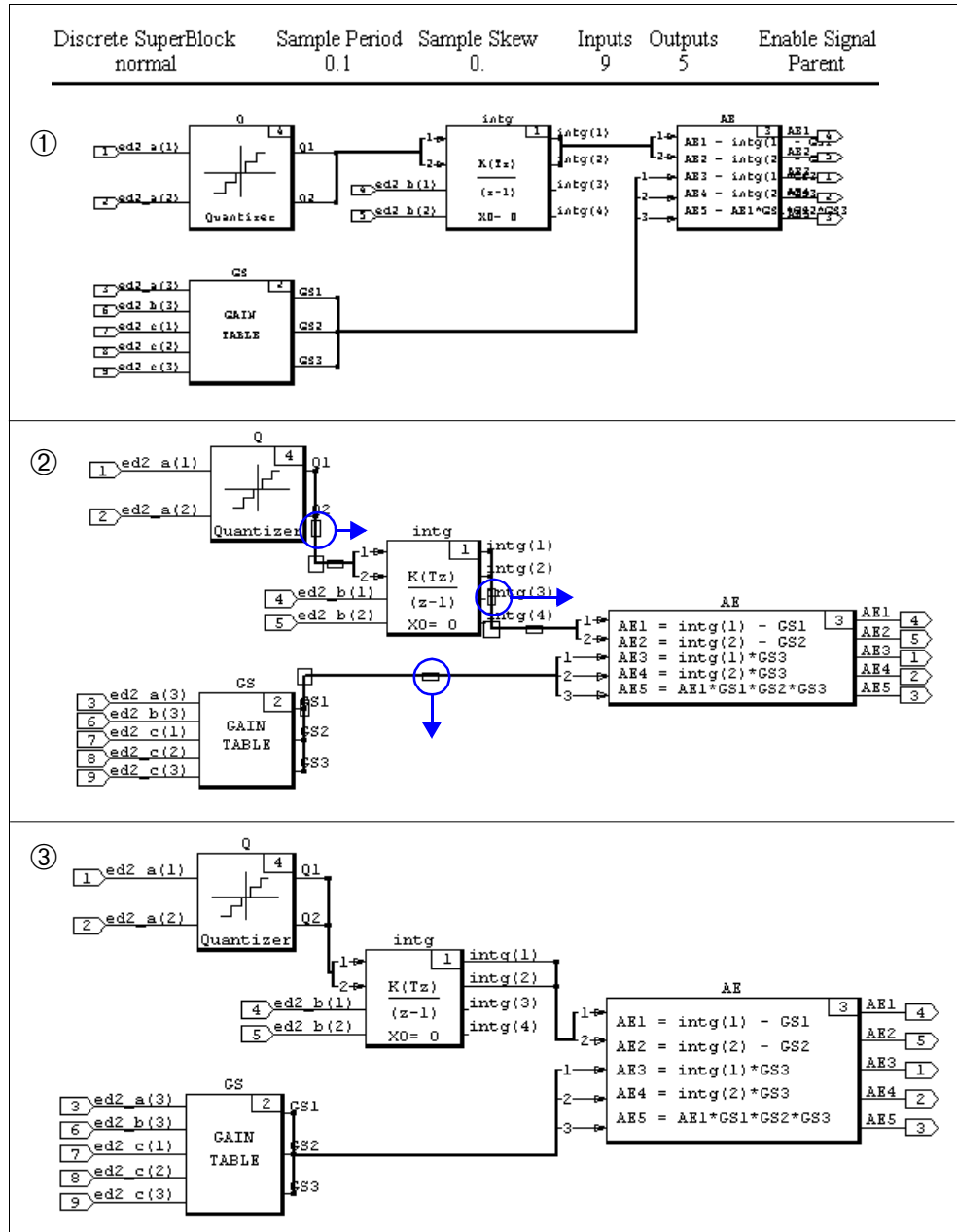
1. To make the labels more readable, increase the font size to 16 points. You can do this via the combo box on the Edit toolbar, or you can place the cursor in the window and type > until the font is the desired size. (Type < to reduce the font size.)
2. To widen block 3, place the cursor over the block 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 (see [5.6.1 Automatic and Manual Connection Routing](#)), 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.

3. 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.

Figure 5-13 Modifying a Block Diagram



5.7 Creating a SuperBlock That Uses the Connection Editor Extensively

[Example 5-5](#) showcases the Connection Editor. It demonstrates how block labels and settings affect the appearance of the block diagram.

Example 5-5 Creating SuperBlock Forty

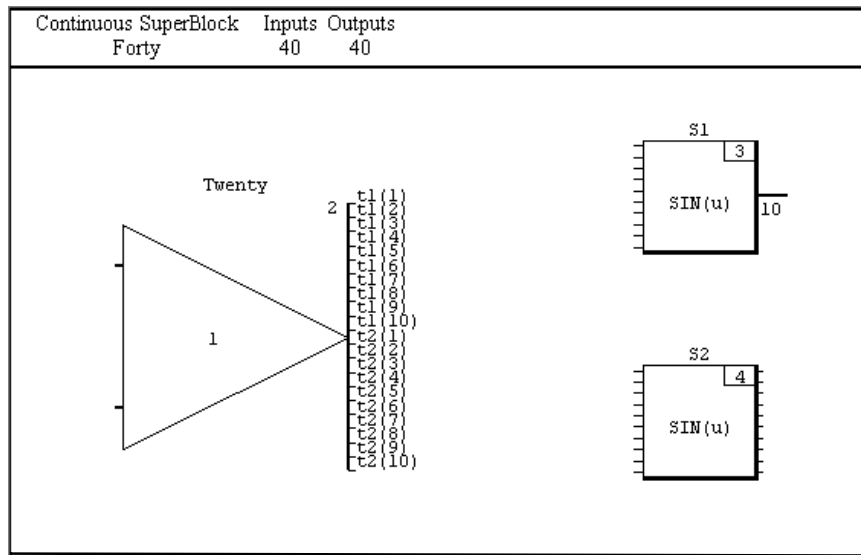
1. Create a new SuperBlock named Forty. Give it 40 inputs and 40 outputs. Set the Input Naming field to Enter Local Label Names. From the Inputs tab, specify labels as follows: Type **f1(1:10)** in cell 1, **f2(1:10)** in cell 11, **f3(1:10)** in cell 21, and **f4(1:10)** in cell 31. Click OK.
2. Create 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 Output Labels, and set the Input Pins field to Vector. Click OK.
3. Select SuperBlock Twenty, and drag the corner nearest the ID number to enlarge it so that all the pins are distinctly shown.

When the finished block is displayed, note that only two inputs pins are shown. This reflects the fact that two vectors with unique labels are used. There are 20 pins on the output, as is normal for scalar mode.

4. Create a SIN block with 10 inputs and 10 outputs. Duplicate the block by placing the cursor over it and typing d.
5. Edit the first SIN block. Name it S1. On the Outputs tab, specify **s1a(1:5)** in cell 1, and specify **s1b(1:5)** in cell 6. On the Display tab, enable Show Output labels, and set the Input Pins field to Scalar and the OutPut Pins field to Bundle. Click OK.
6. Name the duplicate SIN block S2.

Your diagram should now look something like [Figure 5-14](#). The outputs are displayed as a single thick pin; the number 10 indicates the number of pins in the bundle.

Figure 5-14 SuperBlock Forty Before Connections



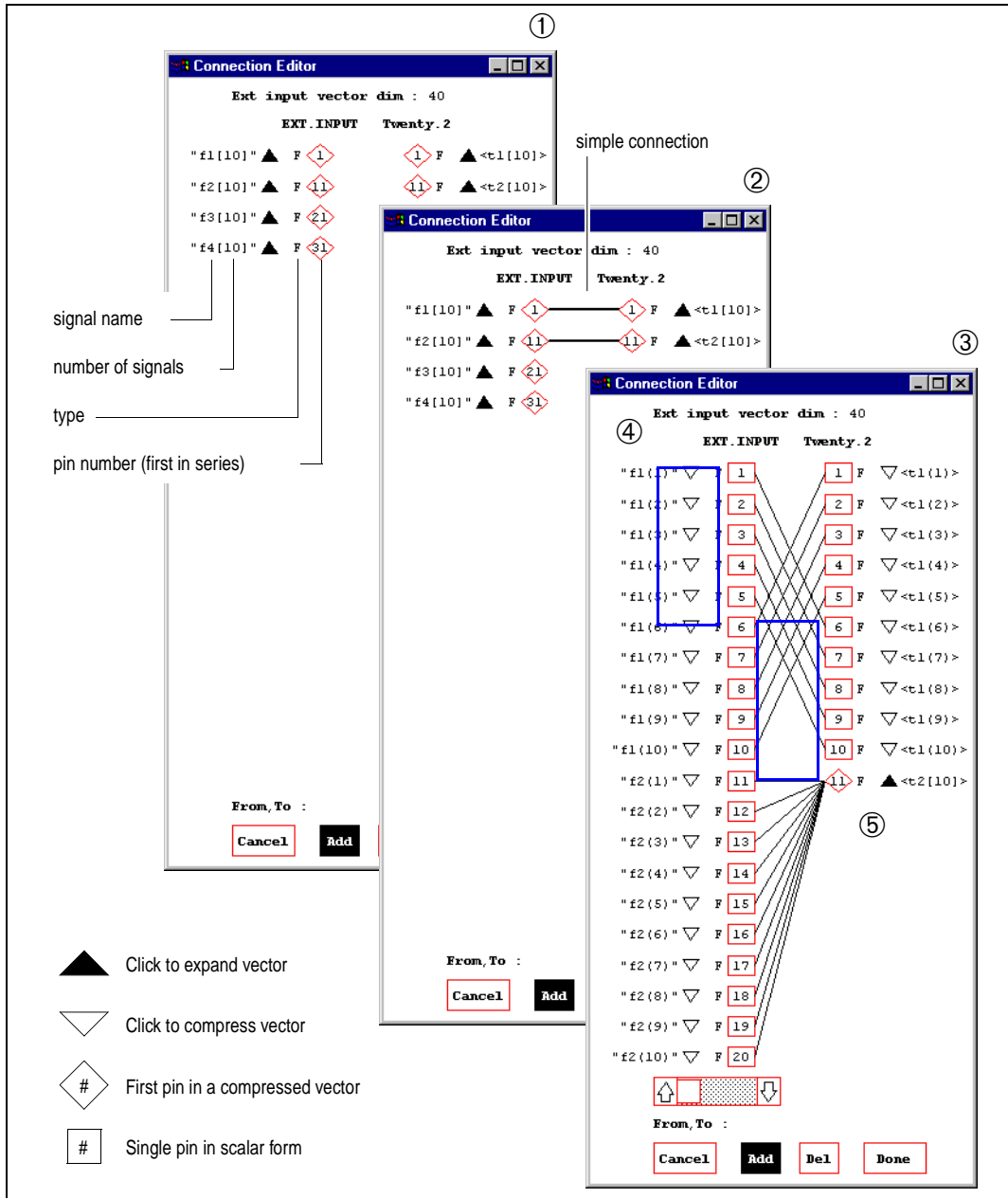
7. To connect external inputs to Twenty, middle-click in white space to the left of the block, and then middle-click the block.

The Connection Editor appears. The initial view, ①, in [Figure 5-15](#), demonstrates that vectored labels are grouped together by default.

- a. Expand a vector (to scalar view) by clicking a filled triangle. Then compress the vector by clicking a hollow triangle associated with any label in the vector.
- b. Double-click the Add button to create a simple connection from the external inputs to Twenty (see ②). Expand the *f1* and *f2* source vectors by clicking the filled triangle beside *f1*[10] and *f2*[10]; do the same for the *t1* destination.

Note that a scroll bar appears above the Cancel button (see ③). With the vector expanded we can create single connections or multiple connections.

Figure 5-15 Connecting External Inputs to Forty



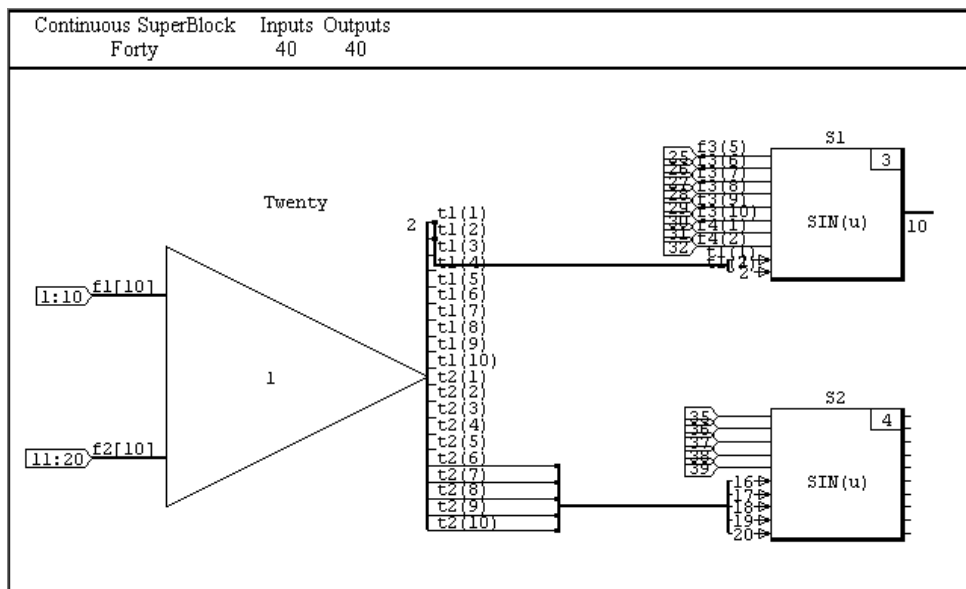
- c. With the Add button selected, lasso (drag a selection box) the first 5 pins from *f1* (④), and then select the last five pins of *t1* (⑤). Connect *f1*(6:10) to *t1*(1:5) in the same manner.
- d. Click Done.

Twenty external inputs are now connected to Twenty. Next we connect external inputs to S1.

- 8. Middle-click in empty space to the left of S1, and then middle-click S1. In the From, To Field, type 25:32; 1:8. Click Done.
- 9. Select Twenty and S1, and then raise the Connection Editor. Double-click Add. Click Done.
- 10. Create a connection from external inputs 35:39 to the first 5 inputs of S2.
- 11. Select Twenty and S2, and then raise the Connection Editor. Connect the last five pins of *t2* to the last five pins of S2. Click Done.

The diagram should now resemble [Figure 5-16](#).

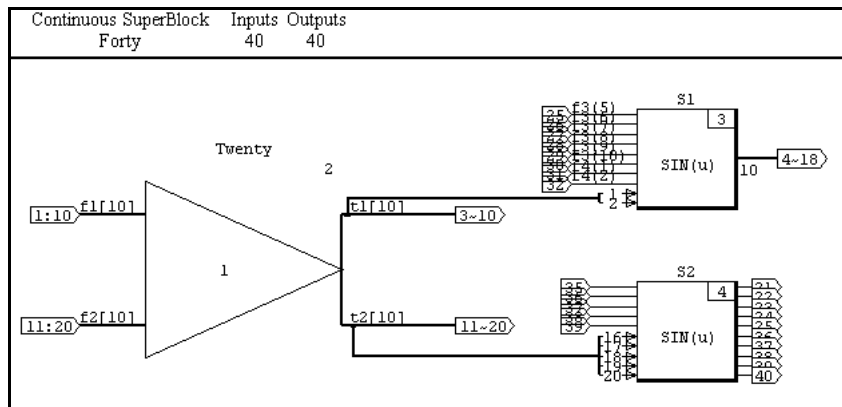
Figure 5-16 **Diagram Before External Output Connections**



12. Create external outputs as follows:
 - a. Create a simple connection between Twenty and the external outputs. To simplify the appearance, select Twenty; and set the OutPut Display Mode combo box on the toolbar to Vector.
 - b. Create the following connections from S1 to the external outputs: channels 9 and 10 to external outputs 1 and 2, channels 1:4 to 4:7, and 5:8 to 15:18.
 - c. Connect S2 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 (1:5) to (21:25) and from (6:10) to (36:40). Click Done.

Your connections are complete, but the diagram is disorderly (see [Figure 5-17](#)).

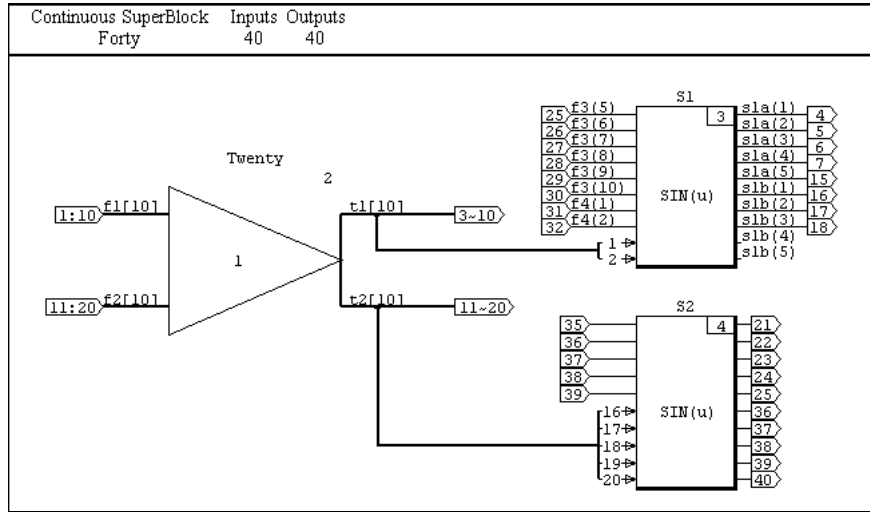
Figure 5-17 Finished Diagram Before Appearance Improvement



13. To improve the readability of the outputs:
 - a. Make the SIN blocks taller by placing the cursor over each block and pressing T. Move the blocks as necessary to prevent overlap.
 - b. Make block Twenty smaller by selecting it, placing the cursor in the selection box near the ID number, and dragging it to the desired size.
 - c. 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.

- d. Select S1, and change the Output Display Mode to Scalar on the toolbar.
The results are shown in [Figure 5-18](#).

Figure 5-18 **Diagram with Resized Blocks and Manual Routing with Vectorized Output from Twenty**



- Change the Input Display Modes and Output Display Modes on the toolbar for each block. Check out the Scalar, Vector, and Bundle settings.

6

SuperBlock Timing and Transformation

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.

6.1 Types of SuperBlocks

SystemBuild supports the following types of SuperBlocks, which are based on their timing methods:

- *Continuous SuperBlocks*
- *Discrete SuperBlocks*
- *Triggered SuperBlocks*
- *Procedure SuperBlocks*

6.1.1 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 [8.3.3 Selecting an Integration Algorithm](#) on p. 179.

6.1.2 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 interval (inverse of 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 is 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 executes 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 runs periodically, but only when it is enabled by its parent SuperBlock or the specified 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 executes 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.

6.1.3 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 negative to positive (≤ 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 second. 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 [6.3.2 Timing of Trigger Subsystems](#). Four choices are available:

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

Timing Requirement — When Output Posting is After Timing Req., 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 are available to other system elements. The lower the timing requirement, the higher the priority of the subsystem.

6.1.4 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 is generated for each procedure SuperBlock, and multiple references to the same procedure reuse the single function. Using procedure SuperBlocks can reduce the generated code size if SuperBlocks are referenced multiply.

There are two ways to use procedure SuperBlocks in a model:

- A direct reference
- A reference through a Condition block

For the most part, procedure SuperBlock behavior is consistent for both types of references, but some differences exist; these are detailed in the Condition block discussion in online Help.

There are six classes of procedure SuperBlocks: [Standard Procedure](#), [Macro Procedure](#), [Inline Procedure](#), [Background Procedure](#), [Startup Procedure](#), and [Interrupt Procedure](#); however, the last three classes are asynchronous SuperBlocks. 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 code generated by AutoCode.

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 standalone reusable functions. The exception to this is the inline procedure SuperBlock (see [Inline Procedure](#) on p.128).

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 inherits a 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.

Following is a list of limitations of procedure SuperBlocks:

- Procedure SuperBlocks cannot be used in any continuous SuperBlock or as the top-level SuperBlock.
- Procedure SuperBlocks are the only type of SuperBlock that can be nested within another procedure SuperBlock.
- The ReadVariable and WriteVariable blocks (see 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 might not occur in the order that you would assume.
- The requirement to treat procedures as standalone functions prohibits using DataStore blocks (see [6.2 Using DataStores](#)) in procedures.

Standard Procedure

The standard procedure SuperBlock is a generic utility that may be nested within any discrete, triggered, or procedure SuperBlock. A standard Procedure inherits the attributes of the parent SuperBlock. These are the only procedure SuperBlocks supported within the Condition block.

Macro Procedure

The macro procedure behaves the same as a standard procedure during SystemBuild simulations. However, AutoCode substitutes 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 6-1 **Macro Procedure SuperBlock**

Type the following in the Code tab:

```
Read_A0;  
Channel_1;
```

The above results in the following generated C code:

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

Input and output variables are listed individually following the optional arguments.

You must input the actual macro code 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 you 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.

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, using of inline procedures, versus standard procedures, influences 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 does 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 is treated like a standard procedure and an individual subsystem is created for that particular reference.

Asynchronous Procedure

Although there is no asynchronous procedure class, background, startup, and interrupt procedures are asynchronous procedures.

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. You can construct hierarchies of standard and macro procedures within the background SuperBlock. You cannot include dynamic blocks with states in a background SuperBlock, although state-like behavior may be achieved with variable blocks. UserCode blocks (UCBs) or BlockScript blocks in background procedure SuperBlocks cannot have states.

The background procedure is primarily a concept specific to AutoCode and is not supported in the SystemBuild simulator.

Startup Procedure

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

A startup SuperBlock cannot 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. You can construct hierarchies of standard and macro procedures within the background SuperBlock. You cannot include dynamic blocks with states in a startup procedure SuperBlock, although state-like behavior may be achieved using variable blocks. UserCode blocks (UCBs) and BlockScript blocks in startup procedure SuperBlocks cannot have states.

Interrupt Procedure

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 a concept specific to AutoCode and is not supported in the SystemBuild simulator.

The Interrupt 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; it must interact with other system elements through ReadVariable and WriteVariable blocks. You can construct hierarchies of standard and macro procedures within the interrupt SuperBlock. You cannot include dynamic blocks with states in an interrupt SuperBlock, although state-like behavior may be achieved using variable blocks. You can construct hierarchies of standard and macro procedures within the interrupt SuperBlock. UserCode blocks (UCBs) or BlockScript blocks in startup procedure SuperBlocks cannot have states.



NOTE: We encourage you to use asynchronous triggered SuperBlocks instead of interrupt procedures when possible because asynchronous triggers provide a more flexible, simulatable solution without most of the restrictions of interrupt procedure SuperBlocks. For example, they function correctly in simulation and in standalone AutoCode situations. They are fully supported by HyperBuild, and interrupt procedures are not. And, they share I/O normally with other SuperBlock types rather than communications being restricted to ReadVariable and WriteVariable blocks.

6.1.5 Effects of Nesting on Enabled and Trigger SuperBlocks

Enabled (discrete with either parent or pin number enable signal) and trigger SuperBlocks can inherit activation signals from their parent SuperBlocks. [Table 6-1](#) 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 is free-running, enabled, triggered, or idle (never executed).

Table 6-1 Relationship between Parent and Child Discrete and Trigger SuperBlocks

Parent→		Discrete			Trigger				
Child ↓	Parent	None	Pin #	Parent	None	Pin #	Cont	Top	
Discrete	Parent	P	F	E	F	F	E	F	F
	None	F	F	F	F	F	F	F	F
	Pin #	E	E	E	E	E	E	F	E
Trigger	Parent	N	N	N	N	N	T	N	N
	None	N	N	N	N	N	N	N	N
	Pin #	T	T	T	T	T	T	T	T
Key to table entries:		E = Enabled			P= E if first non-DP parent is E, else F				
		F = Free-running			T = Triggered				
		N = Never executed							

6.2 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 (see [6.5 AutoCode Timing Properties](#)).



NOTE: In most cases, using ReadVariable/WriteVariable blocks provides a simpler, though less deterministic, way to pass data between subsystems.

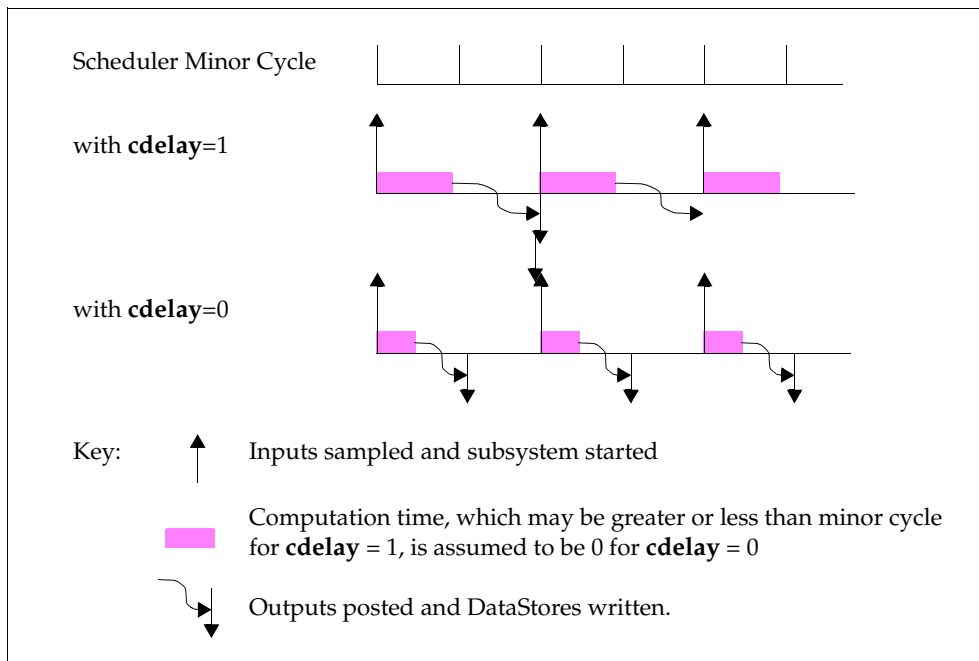
6.3 Simulation Timing Properties

This section discusses simulation timing properties, including subsystems (see [8.1 Dividing Your Model into Subsystems](#) on p.168) and DataStores (see [6.2 Using DataStores](#)). At the end of the section, we present an example that illustrates asynchronous timing.

6.3.1 Timing of Discrete Subsystems

Figure 6-1 illustrates the timing by which discrete subsystems are updated.

Figure 6-1 Simulation, Discrete Subsystem Output Timings



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.

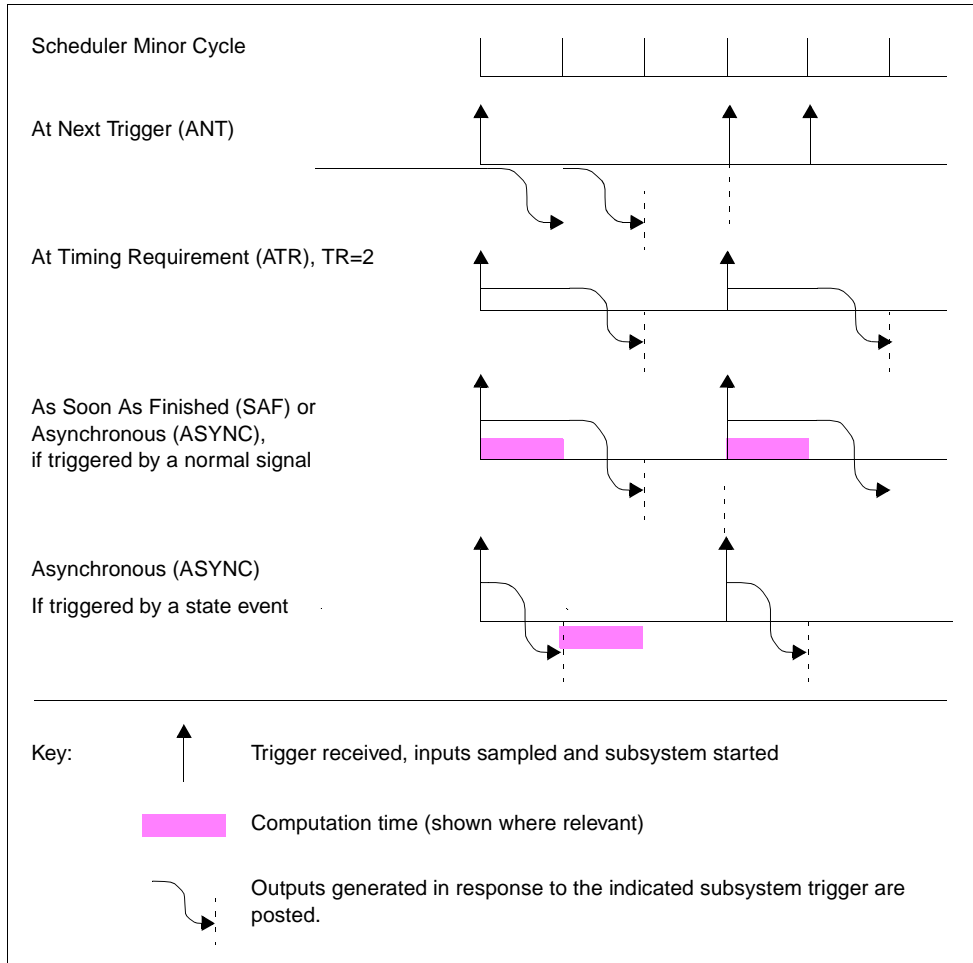
The situation differs according to whether the `cdelay` simulation keyword is specified to be `TRUE` (or `actiming`, which sets `cdelay` to `TRUE`). If `cdelay` is

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 to `FALSE`, no delay is added, the computation time is assumed to be instantaneous, and the outputs are posted (and DataStores written), immediately.

6.3.2 Timing of Trigger Subsystems

The timing of trigger subsystems is shown in Figure 6-2. The four types are defined in terms of their output posting requirements.

Figure 6-2 Trigger Subsystem Output Timings



At Next Trigger

The at next trigger (ANT) 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.

At Timing Requirement

For an at timing requirement (ATR) subsystem, you specify an amount of time to elapse from the beginning of execution to the time that the output is posted. Use this type of subsystem when determinacy is an issue and is more important than sheer performance.

As Soon As Finished

The outputs of an as soon as finished (SAF) subsystem are posted at the beginning of the next minor cycle after the subsystem finishes its computations. Use this type of subsystem when performance is more important than determinacy.

Asynchronous

An asynchronous (ASYNC) trigger subsystem differs from other trigger 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, but they can support model inputs and outputs, dynamic blocks, states (including states in UCBs) and so forth.

For simulation, the asynchronous subsystem is both leading and trailing edge triggered—that is, this subsystem is scheduled if the triggering signal transitions from negative to positive (≤ 0 to >0) and if it transitions from positive to negative (>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 is treated the same as a SAF triggered subsystem, except its priority is higher and the triggering is double-edged.

- If the triggering signal is a state event (see [13.6 State Events](#) on p.313), then the subsystem executes 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, the asynchronous trigger subsystem functions as an interrupt service routine in the simulator, where the simulated interrupts (from the ZeroCrossing block) occur asynchronously with 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 differs 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 is scheduled the same as a SAF triggered subsystem, except its priority is higher and the triggering is double-edged.
- 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 does not schedule this subsystem. AutoCode generates a wrapper for the asynchronous subsystem that is intended to be used as an interrupt procedure in your system.

6.3.3 Example: Using an Asynchronous Triggered SuperBlock

For this example, 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 $T_s = 0.4$ [sec] The controller is simple, and the overall system dynamics can be considered open loop. The controller's 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 detects the starting time at which we need to apply the step input. The starting time is the first zero crossing of a sinusoidal signal $sgn = \sin(2*\pi*f + phase)$, where $f = .5$ [Hz] and $phase = \pi/2$.

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.

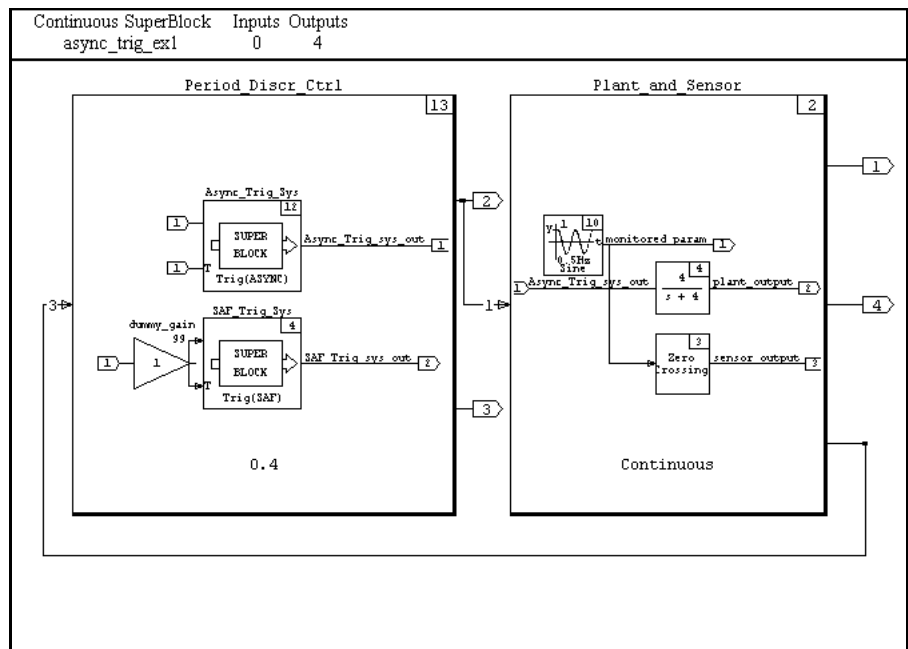
To load the file into the editor:

1. From the Xmath command area, copy the catalog file to your local system as follows:

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

2. From the Catalog Browser, load the file.
3. Open SuperBlock `async_trig_ex1` in the editor.

Figure 6-3 Asynchronous Trigger Example



As shown in [Figure 6-3](#), in order to demonstrate the difference between an SAF triggered system and an asynchronous triggered system, both triggered systems are built into the controller so that you can compare simulation outputs.

The catalog hierarchy is as follows:

- | | | |
|---|--------------------------------|---------------------|
| 0 | <code>async_trig_ex1</code> | Top-level SB |
| 1 | <code>Period_Discr_Ctrl</code> | Discrete controller |

2	Async_Trig_Sys	Asynchronous triggered system
3	SAF_Trig_Sys	Triggered system
0	Plant_and_Sensor	Continuous plant and sensor



NOTE: The numbers to the left above represent the subsystem number. The indent level represents the relative position of the SuperBlock in the SuperBlock hierarchy.

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 is 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 is an external input, which causes 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.

To simulate the model:

1. From the Xmath command area, 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});
```

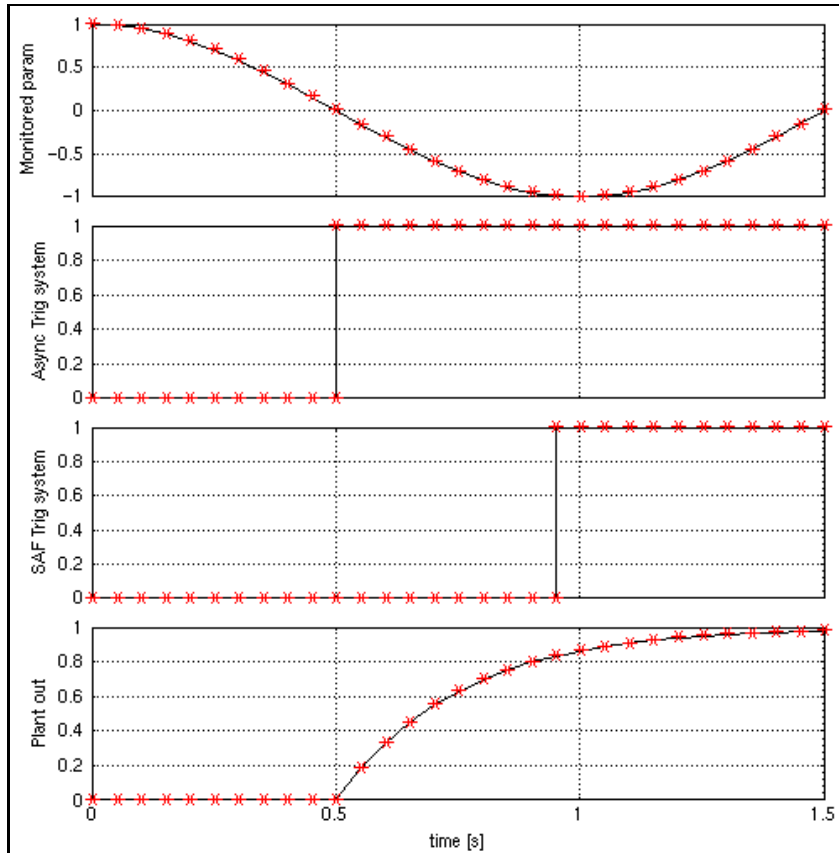
3. Plot the results:

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

Figure 6-4 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

Figure 6-4 **Asynchronous Triggered Timing vs. SAF**



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 system. 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.

If you want to know how AutoCode handles this situation, see [6.5.3 AutoCode and the Asynchronous Triggered System](#).

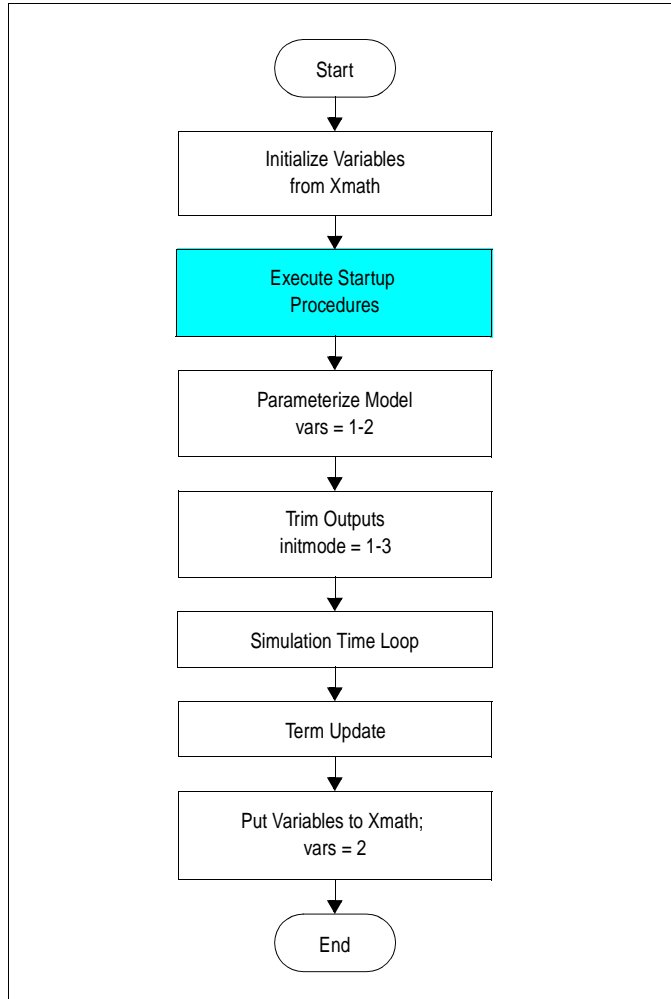
6.4 Execution of Procedures During Simulation

The simulation support of background and interrupt procedures cannot be accurately represented during a non-real-time simulation. As a result, these procedure SuperBlock references are not executed during simulation.

Typically, the background and interrupt procedures are referenced directly from within the part of your model that you use to generate code (with AutoCode), but this reference is not executed by the simulation.

Figure 6-5 illustrates the sequence of initializations that occur in a SystemBuild simulation. Note that only startup procedure SuperBlocks are supported during simulation.

Figure 6-5 Pre-Simulation Initialization Steps



6.5 AutoCode Timing Properties

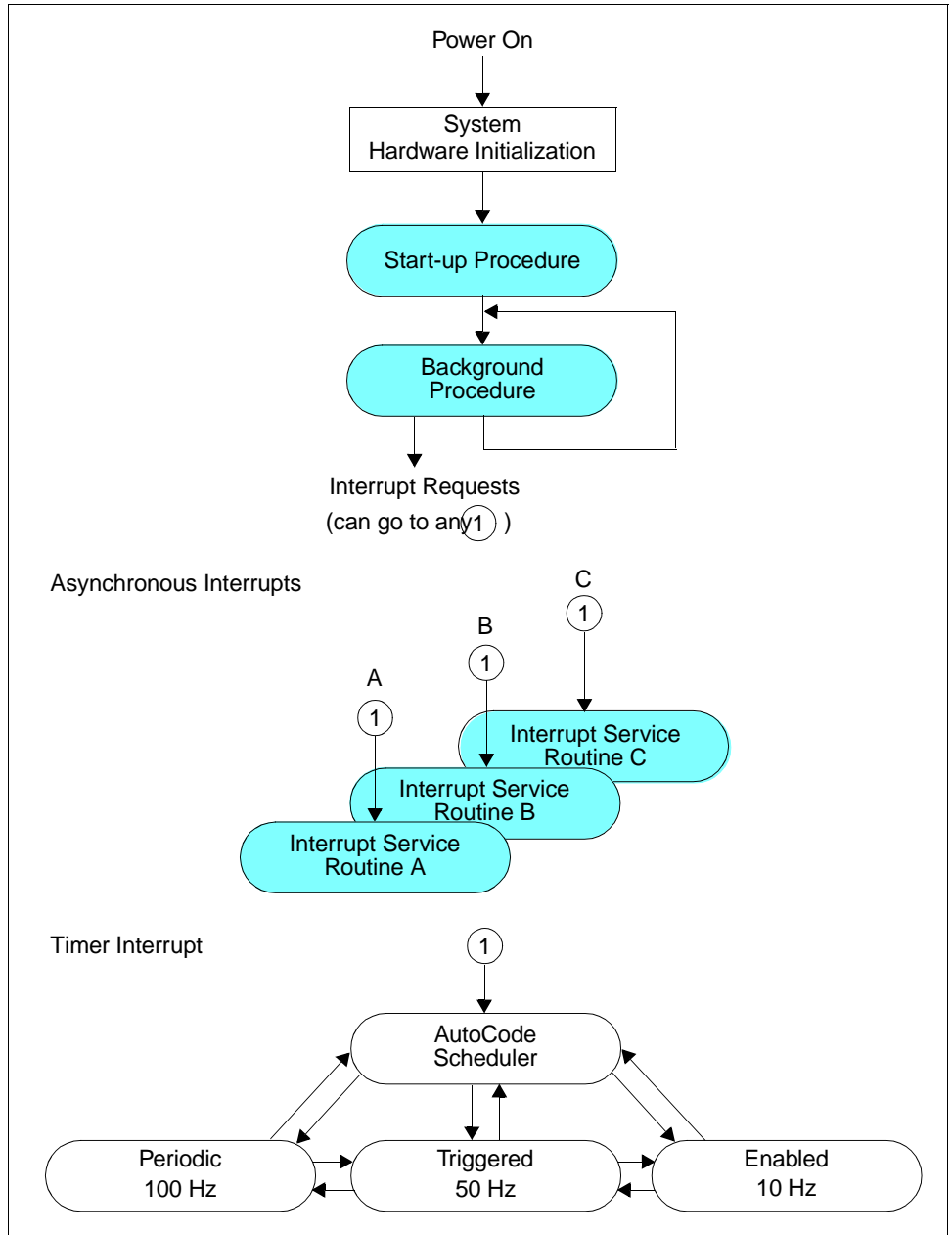
This section provides information on the following topics:

- [*AutoCode Real-time Application Execution Sequence*](#)
- [*AutoCode Timing Features Associated with Using DataStores*](#)
- [*AutoCode and the Asynchronous Triggered System*](#)

6.5.1 AutoCode Real-time Application Execution Sequence

[Figure 6-6](#) shows the AutoCode real-time execution sequence, including startup, background, and interrupt procedure SuperBlocks.

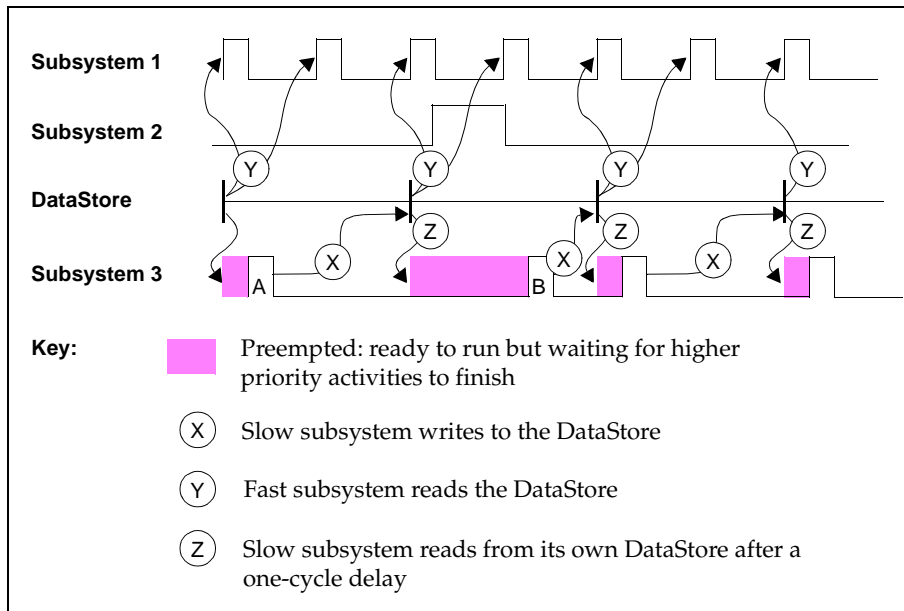
Figure 6-6 AutoCode Real-time Application Execution Sequence



6.5.2 AutoCode Timing Features Associated with Using DataStores

Timing features associated with using DataStores that are peculiar to AutoCode are illustrated in Figures 6-7 through 6-9. In this example, two different subsystems are writing to the same DataStore location, and the differences between the timings of the subsystems create 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 is no conflict (for example, by making it impossible to trigger or enable them at the same time).

Figure 6-7 DataStore Timings



In Figure 6-7, 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.

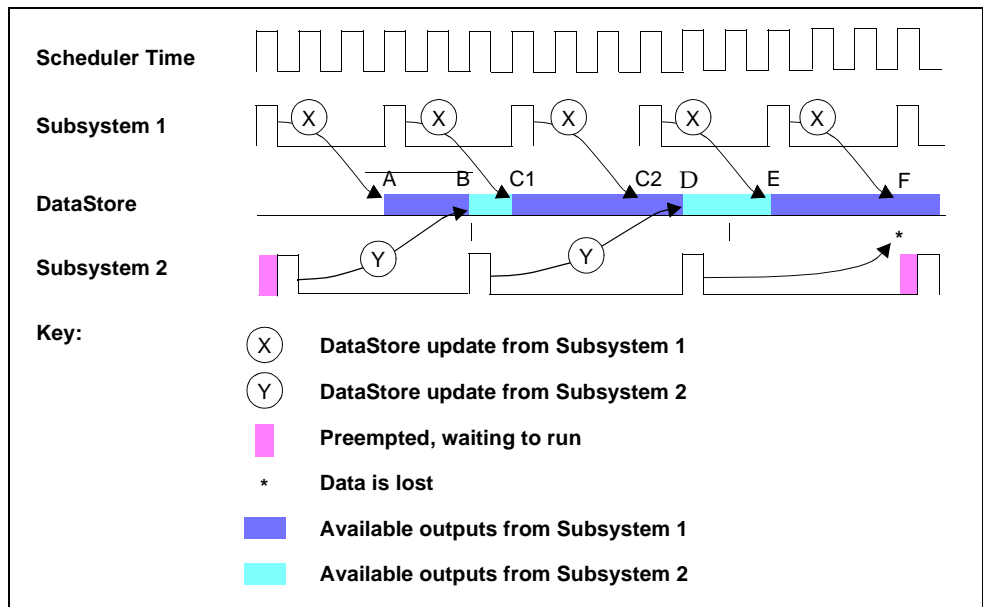
Whenever Subsystem 3 runs to completion, it posts its outputs in a DataStore for transmission to other subsystems and to itself. This is indicated by X in the diagram. Although the output of Subsystem 3 might become available before the next execution of Subsystem 1 (as at A), this cannot be guaranteed (as at B, when the occasionally-triggered Subsystem 2 also preempts Subsystem 3). Therefore, to

guarantee determinacy, the data from the DataStore is not made available to Subsystem 1 or other subsystems until the next time that Subsystem 3 is primed for execution. This same thing 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 because the outputs become available the next time that Subsystem 3 is primed for execution, as shown at location Z in Figure 6-7. Without the DataStore, the outputs of Subsystem 3 would be available within the subsystem on the same cycle as they were generated.

Figure 6-8 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).

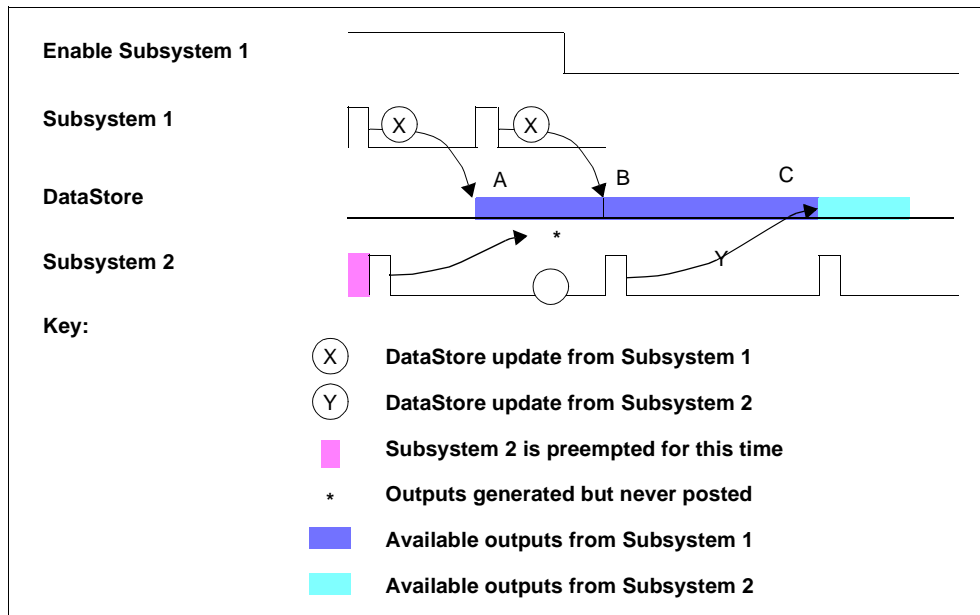
Figure 6-8 Writing into a DataStore Register from Two Different Subsystems



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 has a higher priority and prevails. The output of Subsystem 2 at this time never appears in the DataStore.

This illustrates the way that data in the DataStore may be visible for uneven time intervals, but the situation is always determinate. Figure 6-9 illustrates a situation where data from a slow subsystem might never show up in the DataStore and how to work with this situation.

Figure 6-9 Enabled Subsystem Writing into a DataStore



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.

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 posted (point C) in the usual manner.

6.5.3 AutoCode and the Asynchronous Triggered System

What does Autocode do in order to match the simulation results we obtained in [Example: Using an Asynchronous Triggered SuperBlock](#) on p.136?

While generating code for the discrete controller `Period_Discr_Ctrl`, the call to the `Async_Trig_Sys` subsystem is not included in the `SCHEDULER()` function but is left as an interrupt service request (ISR) call for the real-time controller. This way, the subsystem `Async_Trig_Sys` is executed almost immediately after 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 are present in many applications, it is important to recall that *the asynchronous triggered system is immediately available after servicing the ISR.*

6.6 SuperBlock Transformation

Each SuperBlock has computational timing attributes (continuous or discrete, triggered, and so forth). SystemBuild provides a powerful feature that enables you to transform SuperBlocks from continuous to discrete or from one discrete rate to another.

In this section, we discuss the following topics:

- [Transformation Limitations and Implications](#)
- [Transformation Methods](#)
- [Undoing a Transformation](#)

6.6.1 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:
 - STD (State TransitionDiagram), DataStore, and FuzzyLogic blocks
 - Signal TypeConversion 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:

$$s = \frac{2(z - 1)}{T(z + 1)} \quad (6-1)$$

SystemBuild performs transformations between discrete rates by first converting the current rate to continuous time and then converting from continuous time back to discrete with the new rate (see [Example 6-2](#)).



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

Example 6-2 Transforming a Block

Assume a block sampling at 0.1 seconds in Num/Den format with transfer function in the z domain = $1/z^2$; 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 allow SystemBuild to 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.

You can select a discrete integrator from the block's Parameters tab. Integrators for these blocks are:

- 1 Forward Euler
- 2 Backward Euler (default)
- 3 Tustin (trapezoidal)

6.6.2 Transformation Methods

You can transform SuperBlocks in several ways:

- Transform nonlinear dynamic systems from Xmath using the `lin()` and `discretize()` functions (see [10. Linearization](#) and online Help).
- Use the Transform option on the Tools menu in the Catalog Browser.
- Change the SuperBlock type from the SuperBlock Properties dialog.



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. From the editor, select File→Update, and then save the SystemBuild catalog before proceeding with your transformation.

Transformation from the Catalog Browser

To make a transformation from the Catalog Browser:

1. In the Catalog Browser, 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 are transformed to the new settings; the only restriction is that the hierarchy cannot contain any state transition diagrams or DataStores.

Additionally, if any of the SuperBlocks being transformed are trigger SuperBlocks, they become enabled SuperBlocks during the transformation.

2. Select Tools→Transform.

The Transform SuperBlock dialog appears.

3. Change the block type and other fields as appropriate:
 - a. Discrete to Discrete Transformation: Rate only or Rate and Block coefficients. Coefficients apply for NumDen, StateSpace, and PoleZero blocks in a discrete-to-discrete transformation. Frequency-normalized systems require a rate-only transformation.
 - b. Transform Initial Conditions (of StateSpace blocks). Enable or disable. See [Initial Condition Transformations](#).
 - c. Sample Period (0 = continuous).
 - d. Sample Skew (the first period).

This dialog also allows you to provide attributes for trigger and procedure SuperBlocks. For additional help, press the Help button.

Transformation from the SuperBlock Properties Dialog

You can also transform a SuperBlock from the SuperBlock Properties dialog.

To use the SuperBlock Properties method:

1. Raise the SuperBlock Properties dialog.
2. Click the Type field combo box, and select the SuperBlock type.
3. Type in Sample Period and Sample Skew.

While the methods for changing the type is similar in either dialog, the results can be somewhat different. Using the SuperBlock Properties dialog, only the current SuperBlock being edited is transformed. However, using the Catalog Browser provides the option of transforming multiple SuperBlocks, including lower-level SuperBlocks. Note that with both methods, the transformed SuperBlocks overwrite the original SuperBlocks.

If you change the rate from the SuperBlock Properties dialog, however, 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 nonzero), 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 SuperBlock types in this dialog may not be possible due to insufficient information for the transformation. You'll be notified when this condition exists and instructed on how to perform an alternative method of the desired transformation.

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) 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.

6.6.3 Undoing a Transformation

- You cannot undo a transformation that you initiate from the Catalog Browser.
- Transformations initiated from the SuperBlock Properties dialog can be undone with the general function Edit→Undo All. As the name implies, the SuperBlock reverts to its state at the time the display was last updated to the catalog. All changes to the SuperBlock and its displayed children disappear.

7

SystemBuild Access

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.

The main topics of this chapter are as follows:

- *Overview*
- *SBA Syntax*
- *Basic SBA Tasks*
- *Using SBA*
- *Tutorial*

7.1 Overview

You can enter SBA commands and functions directly into the Xmath command area, execute them from within MathScripts, or call them from an Xmath user callable interface (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 (see online Help), even custom blocks you create yourself (see 20.3.1, p.534). However, IA icon blocks are not supported.

Figure 7-1 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 7-1 SBA MathScript Blocks in the MATRIX_x Context

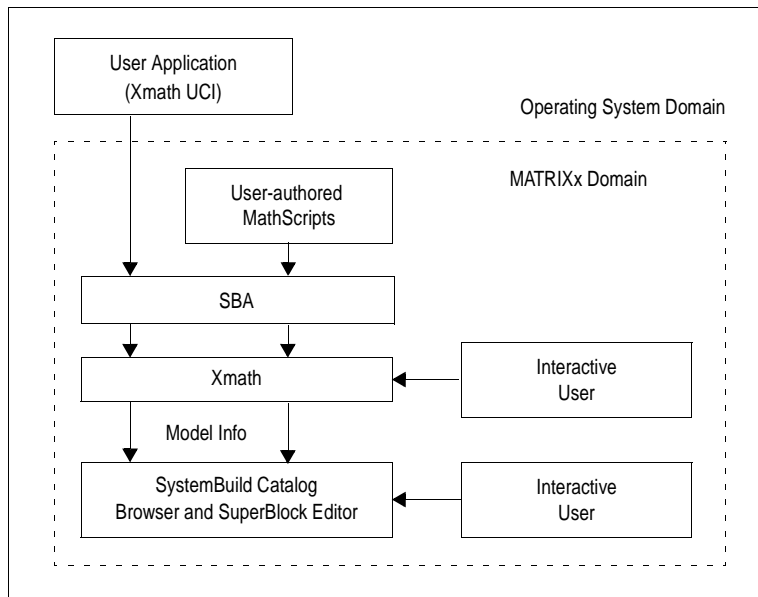
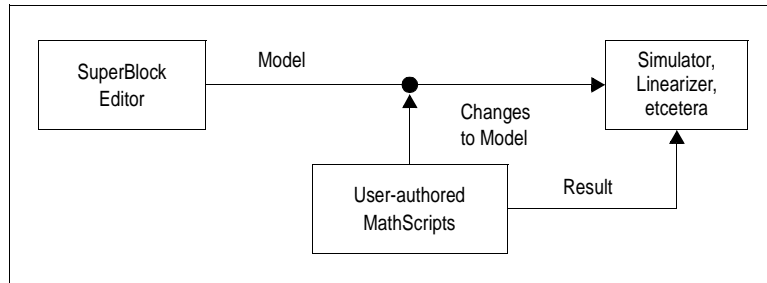


Figure 7-2 shows how you can use SBA to extend the SystemBuild paradigm to allow the results of a simulation to change the model. In this scenario, a MathScript executes a simulation. The MathScript can evaluate `sim()` results and then execute SBA code that modifies the model before the next simulation is started.

Figure 7-2 Typical SBA Program Used in a SystemBuild Application



7.2 SBA Syntax

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 User's Guide*). The basic syntax, and the behavior of inputs, keywords, and outputs is the same.

7.2.1 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 SuperBlock Editor. The syntax is as follows:

```
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 SuperBlock Editor.

7.2.2 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 might 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 [7.3.2 Query](#).

7.2.3 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 and 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 is used if the keyword is not called. Keyword order is not important unless it violates the logical rules of the editor. See 7.4 Using SBA . |

7.3 Basic SBA Tasks

This section shows some basic examples of SBA syntax. Any object you can create and modify in the SuperBlock Editor can be reproduced with SBA. This section shows SBA commands to create a simple model, which is then referred to throughout the section.

7.3.1 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]
```

7.3.2 Query

You can query an existing block to determine what its settings are; you can also 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
```

7.3.3 Modify

These commands modify the model created above.

```
modifysuperblock "cmd_createconnection",
  {inputlabel=["U1";"U2";"U3"]}

modifyblock 1,{labels="on"}
```

7.3.4 Display

The **SBADISPLAY** command allows you to refresh the editor window after changes and to 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}
```

7.3.5 Delete

You can delete any of the elements created so far. For example,

```
deleteconnection 0,1  
deleteblock 2  
deleteSuperBlock "cmd_createconnection"
```

7.3.6 Sample Scripts

For examples of some sophisticated SBA scripts, see *SYSBLD/examples/export*, where *SYSBLD* is the root SystemBuild directory in your distribution. These unsupported SBA commands and functions read all or part of an existing model and create an SBA script that reproduces the model. *Exportsuperblock.msc* is able to do this for an entire SuperBlock hierarchy. You can find other examples of SBA scripts 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.

7.4 Using SBA

This section provides a variety of tips that will help you get the results you want when using SBA.

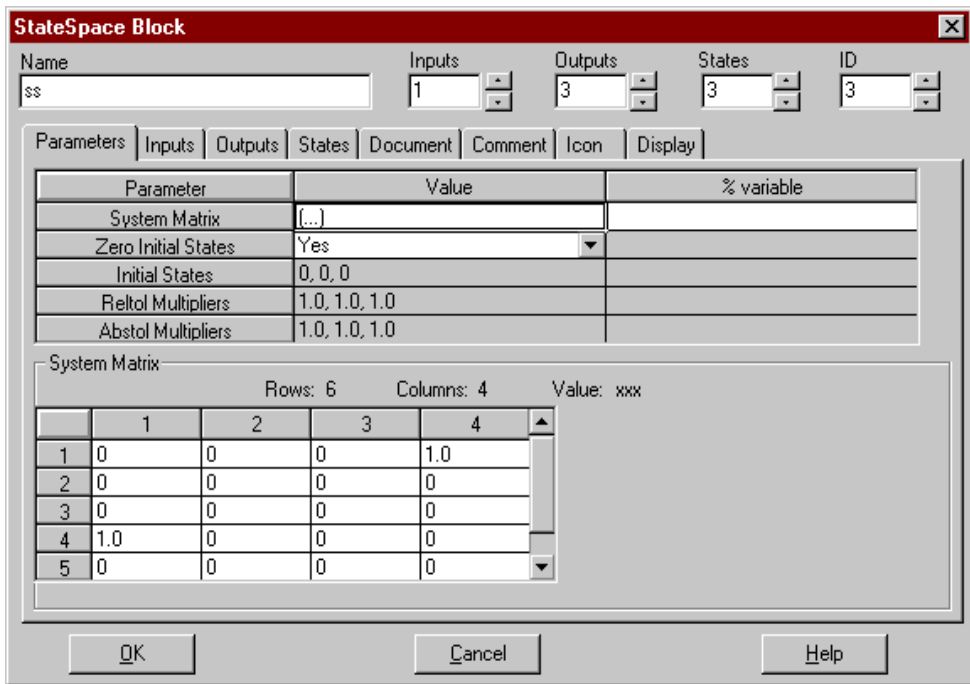
7.4.1 Keyword Ordering

In general, you can specify Xmath keywords in any order because they have defaults. 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 you click OK. If inputs are compatible, the editor provides defaults, even for fields you did not edit; for example, if you specify three outputs, the editor automatically gives you the opportunity to specify three output names, three output labels, and so forth.

When in doubt, look at the block dialog. If you plan to define any of the values shown above the tabs, define them first and in the order shown from left to right. Define inputs on the Parameter tab later. Let's apply this principle to the block dialog in [Figure 7-3](#).

Figure 7-3 **StateSpace Block Dialog**



The name and ID are optional, so it doesn't matter when you define them. Of the remaining items, define inputs, outputs, and states in that order. The dimension of the system matrix is determined from these values.

7.4.2 Block Parameters

Many blocks have parameter dependencies, or special parameters. The `queryblockoptions()` function returns 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 online Help for the specific block before creating your script.

7.4.3 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 passes a call to the editor; if the call does not make sense, the editor attempts to create compatible settings. This behavior is consistent with the editor's interactive behavior, but it makes script debugging difficult because no error messages are returned to Xmath when SystemBuild encounters the improper values. For example, the following is legal command that creates a block in SystemBuild; unfortunately you might 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 the command is interpreted, `outputlabel` is encountered first; therefore, 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 [7.4.1 Keyword Ordering](#), you should always define inputs and outputs first.

7.4.4 Input Formats

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 can see that the input format (integer, matrix of strings, and so forth) is specified for each parameter.

Typical Input Formats

The following table shows samples of possible input formats.

Table 7-1 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. You can specify these inputs as an array of values, or you can specify each value 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"]}
```

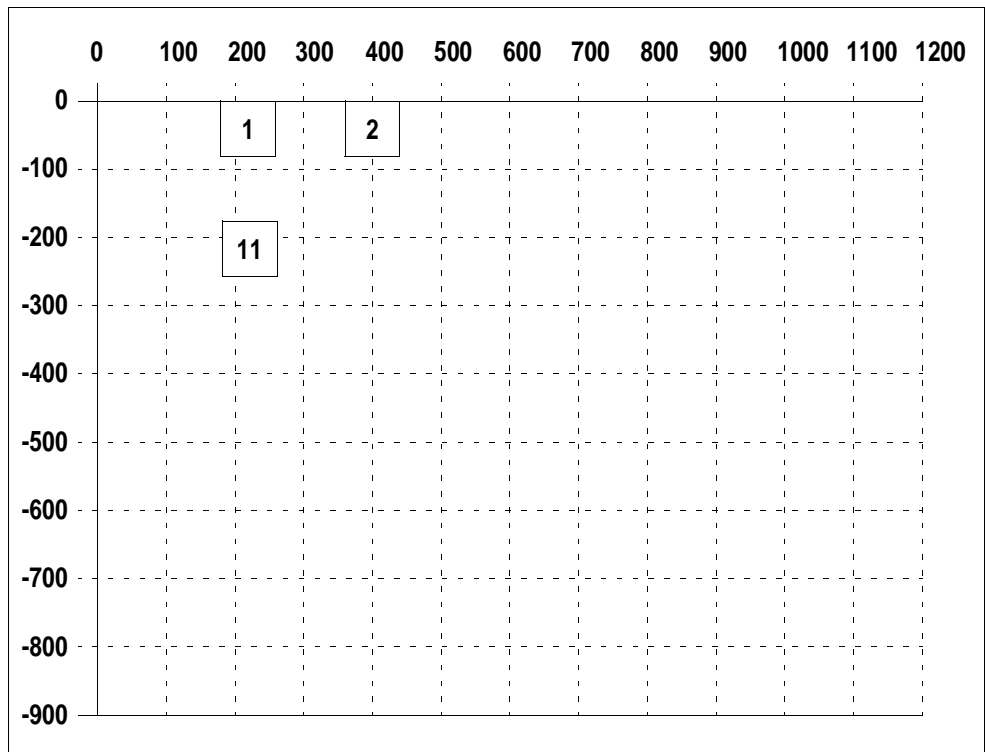
To change one name after the fact, append the index number to the keyword:

```
modifyblock 13, {outputname2="changed_gain_B"}
```

7.4.5 SuperBlock Editor Coordinate System

As shown in [Figure 7-4](#), 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 7-4 SystemBuild Coordinate System



When SBA places new blocks in an empty diagram, the default location of the first block 1 is (180,0); if no ID is specified, the default ID is 1. If the default is assumed for subsequent blocks they are placed from left to right and numbered

sequentially; if the number of blocks is greater than 10, block 11 starts a new row at (180,-180). You can position blocks in the diagram by specifying a block ID. For example, assigning a block ID of 14 places a block at (720,-180) as long as that position is unoccupied.

You should not specify a Y value greater than 0; although SystemBuild places your block according to your direction, unpredictable numbering can result.

7.5 Tutorial

As discussed in previous sections, 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.

7.5.1 Building the Predator-Prey Model

[Example 7-1](#) is a model from the field of population demographics, illustrating the interaction of predator and a prey species in a competitive environment. You can copy it from *SYSBLD/examples/pred_pre/predprey.ms*.

Example 7-1 **Commands to Build the Predator Prey Model**

```
# 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 SuperBlock Editor may differ sometimes from the obvious way of doing things in SBA. When working interactively, you typically create a few blocks and then perhaps duplicate blocks in order to build the model quickly and efficiently; you might connect a few blocks, create additional blocks 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 and then form the connections; we then create the second loop and form its connections. We create the blocks that work between the two loops, and then connect them; finally, we add the external inputs and outputs. You can organize your script as you see fit.

7.5.2 Simulating the Predator-Prey Model

This MathScript file is available on your system as *SYSBLD/examples/pred_prey/predprey_driver.ms*, which in turn calls (executes) the previous file, *SYSBLD/examples/pred_prey/predprey.ms*. In [Example 7-2](#), we create the time- and input data vectors, run the simulation, and plot the output.

Example 7-2 Creating the Data Vectors, Running the Simulation, and Plotting 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_prey",t,u);
# plot the results
plot(t,y)?
#
# modify the model
modifyblock 9, {gain=.2}
# rerun the simulation
y2 = sim("predator_prey",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}
```

8

Simulator Basics

Once you have built a model in the SystemBuild Editor you're almost ready to analyze and simulate your system using the SystemBuild simulator. Prior to the simulation itself, you need to set up parameters and might want to perform an analysis prior to the simulation. During the simulation, you can perform experiments by varying the values of variables.

This chapter explains how to use the simulator and customize your simulations with simulation keywords. The major topics are as follows:

- *Dividing Your Model into Subsystems*
- *Scheduling Subsystems*
- *Setting Options and Parameters for Your Model*
- *Analyzing Your Model Prior to Simulation*
- *Some Additional Tools*
- *Simulating Your Model*
- *Terminating Your Simulation*
- *Simulation Errors*

We also discuss the following related functions: **analyze()**, **creatertf()**, and **simout()**.

8.1 Dividing Your Model into Subsystems

By default, subsystems are determined internally by SystemBuild and require no intervention by the user. However, you might want to make some changes to SystemBuild's natural divisions. This topic helps you to understand how SystemBuild creates subsystems and then tells you why and how to make changes.

8.1.1 How SystemBuild Divides Your Model Into Subsystems

Subsystems 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.

An understanding of how subsystems function is helpful in the interpretation of simulation results.¹

Example 8-1 shows how a hybrid (continuous and discrete) model is divided into subsystems.

Example 8-1 Dividing a Model into Subsystems

Consider a model for the following SuperBlocks:

- Two continuous SuperBlocks
- A discrete SuperBlock at a sample rate of 0.1
- Two discrete SuperBlocks with a rate of 0.05

1. When you generate code, AutoCode also uses subsystems in the same way, although the timing might be different (see the *AutoCode User's Guide*).

The subsystems making up this model include:

- One continuous subsystem (made up of two SuperBlocks)
- A free-running periodic subsystem (made up of one SuperBlock) sampled at 0.1
- Another free-running periodic subsystem (made up of two SuperBlocks) sampled at 0.05.

Each of these subsystems then accept inputs and post outputs under the control of the scheduler at scheduler-specified times.

8.1.2 Assigning SuperBlocks to Additional Subsystems

Consider the model described in [Example 8-1](#), but increase the number of SuperBlocks with a sample rate of 0.05 to 200. By default, SystemBuild assigns all 200 SuperBlocks 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 multiprocessor applications it might be desirable to break up the generated subsystem code and divide its execution among several processors.

SystemBuild provides a solution for these problems. You can assign discrete and trigger SuperBlocks to subsystems using the processor Group ID field located on the Attributes tab of the SuperBlock Properties dialog. For this field 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 are assigned to different subsystems.

Using this field, it is possible to divide a set of SuperBlocks with identical timing characteristics into different subsystems. Using our example, the model might 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.

8.2 Scheduling Subsystems

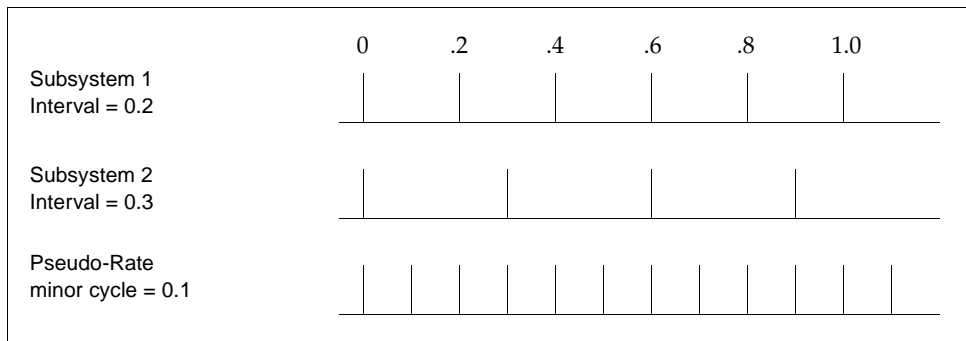
Different scheduler programs are used for simulation and for the execution of generated code. The generated code scheduler is optimized for real-time operations, whereas the simulation scheduler is optimized for performance in simulation. The simulator scheduler controls the overall flow of data between subsystems, scheduling of the subsystems and posting their outputs. If you want the simulator scheduler to match the code scheduling of discrete models, you can use the [actiming](#) (AutoCode timing) keyword.

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. (See [8.2.2 Scheduling Discrete Subsystems.](#))

Whenever a system with one or more discrete subsystems is analyzed for simulation or any other purpose, 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, *including any trigger subsystem timing requirements*. A minor cycle time 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 8-1](#) 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; the pseudo-rate of .1 is generated.

Figure 8-1 Derivation of a Pseudo-Rate



8.2.1 Scheduling Continuous Subsystems

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.

8.2.2 Scheduling Discrete Subsystems

For discrete subsystems, the scheduling 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.

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 (see [DataStores](#) on p.12) at the same time.

- 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 asynchronous, as soon as finished, at timing requirement, and at next trigger (see [Triggered Subsystems](#)).

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 you can modify the posting of outputs with certain keywords.

- Posting of outputs for free-running and enabled periodic subsystems.
 - Modify 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.
- Posting of outputs for triggered subsystems is controlled on a SuperBlock-by-SuperBlock basis. A description of each of these posting options is given under [Triggered Subsystems](#).

Free-running Periodic Subsystems

A free-running subsystem is always enabled and gets executed when its sample time arrives.

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 [13.6 State Events](#) on p.313). 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).

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.

Matching the Timing of AutoCode for Discrete Systems

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 match AutoCode:

Keyword and Value	Description
cdelay = 1	The output posting is always delayed one minor cycle.
initmode = 0	This keyword setting disables the initialization that is normally performed at simulation time.

8.3 Setting Options and Parameters for Your Model

In this section, we continue the discussion of scheduling with a discussion of time lines and how to set them up, tell you how to parameterize your variables so that you can change their values for repeated simulations ([Changing Parameters for Repeated Simulations](#)), and provide some assistance in selecting an integration algorithm for your model ([Selecting an Integration Algorithm](#)).

8.3.1 Simulation Time Lines, Inputs, and Outputs

Input Time Line

The input time line is formed from the time vector entered via the Time Vector/ Variable field on the Parameters tab of the SystemBuild Simulation Parameters dialog 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 are synchronized with input data points (see [Computing External Input Values](#)).
- In continuous and hybrid systems that use variable-step integration algorithms, you are guaranteed that the integration algorithm will converge on each of these time points.



NOTE: There might be other time points for which the algorithm converges.

- Outputs from the simulation are saved for each point of the input time line—that is, each point of the input time line is also a point on the output time line (see *Input Time Line*).

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.

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 SystemBuild Simulation Properties dialog, you enter this data 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 is used for input values.

Whenever the simulation requires external inputs, it first compares the current simulation time (from the internal time line) versus 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).

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 you specify a reporting period for `sim()` with the `dtout` keyword, every integral multiple of this value is a time point on the output time line.
- If you specify extending the output calculations for `sim()` with the keyword `extend`, every discrete subsystem transition time point and every discrete event time point is added to the output time line.

8.3.2 Changing Parameters for Repeated Simulations

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 can 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. You can enter the variable name with a partition specified (*partition.variable_name*) or without (*variable_name*). If you specify a partition name, SystemBuild looks there to resolve variable names rather than in the default partition.

You may assign a default value to 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 (or Enter) 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 you can change 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 (or Enter).

The Simulation vars Keyword

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

If you use the `vars` keyword, the simulator picks up %Variables specified in the model and uses 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 uses 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

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 is 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 Xmath Partition field of the SuperBlock Block dialog. The simulator then looks up the SuperBlock hierarchy for the existence of a partition name. If a 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.

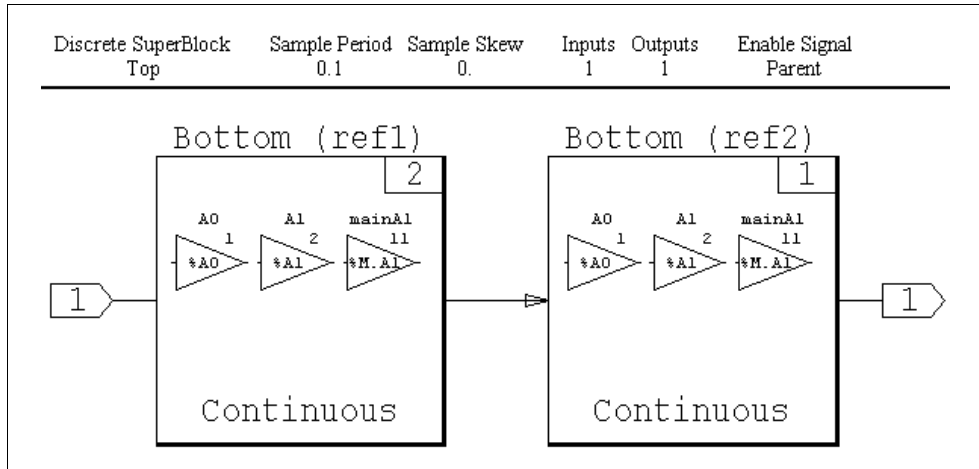
If no partition is specified in the SuperBlock Block dialog for this instance, the simulator continues its search for the next SuperBlock up the hierarchy. If the search through the SuperBlocks is exhausted, the simulator finally looks for the variable in your current working partition. See [Example 8-2](#).

Example 8-2 Demonstrating Parameter Scoping

This example demonstrates the flexibility of parameter scoping; see [Figure 8-2](#). 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 8-2 Example of %Variable Scoping



The model has two SuperBlocks, Top and Bottom. The Top SuperBlock contains two different SB references to SuperBlock Bottom. 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 has 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, the simulator locates A0 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.

8.3.3 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 (a combination of continuous and discrete subsystems)

Given the user-defined initial conditions and input vector, the problem of simulating the SystemBuild model, or obtaining a sequence of solutions to the system equations, 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.

Integration Algorithms

Table 8-1 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-step Kutta-Merson). You can set the algorithm globally using the command:

```
SETSBDEFAULT,{ialg=algnumber}
```

where *algnumber* is a member of the list above. You can also determine the current default number using the command:

```
SHOWSBDEFAULT
```

You can set the integration algorithm for a given simulation on the Parameters tab of the SystemBuild Simulation Parameters dialog, or you can set the **ialg** keyword in the **sim()** function call:

```
y = sim("model",t,u,{ialg = algnumber})
```

Integration Algorithm Recommendations

[Table 8-1](#) lists recommendations for choosing an integration algorithm. Note that a great variety of systems 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 [13. Advanced Simulation](#).

When algebraic loops are present in a model, all methods other than the stiff system solvers, DASSL and ODASSL, introduce a delay into the system even though they might integrate the equations successfully.

Table 8-1 Selecting an Integration Algorithm

Problem Type	Euler	RK2	RK4	FKM	VKM	DASSL, ODAS, GEAR	ABM	QuickSim
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:								
+	Marginally suitable.							
++	Very suitable.							
+++	Best for this problem type.							
*	Only with state events modeling the discontinuities.							
**	Recommended with state events/appropriate dtmin option.							

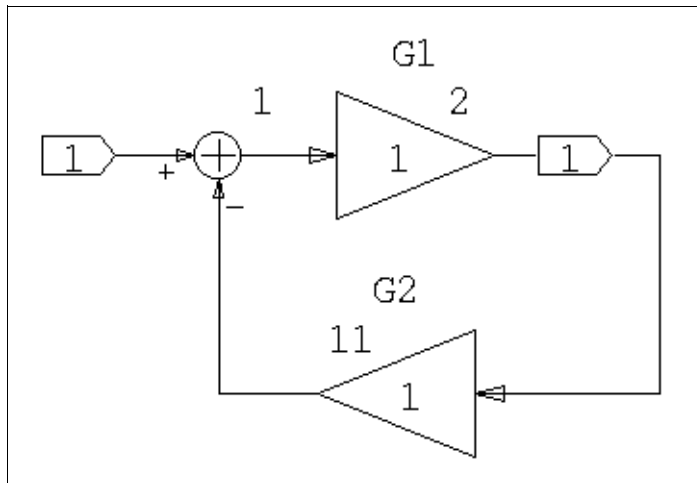
8.4 Analyzing Your Model Prior to Simulation

In this section, we provide you some insights into [Working with Algebraic Loops](#) and [Using the analyze\(\) Function](#) to debug your model prior to simulation. If you want to save your model for simulation or creating code, see [Saving Your Model with the CREATERTF Command](#).

8.4.1 Working with 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 8-3](#), where the input to the gain of the 2 block is dependent on its output. This results in a situation where the simulator cannot readily decide which block to evaluate first.

Figure 8-3 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 8-3](#), illustrates some of these problems. The 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 we 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)

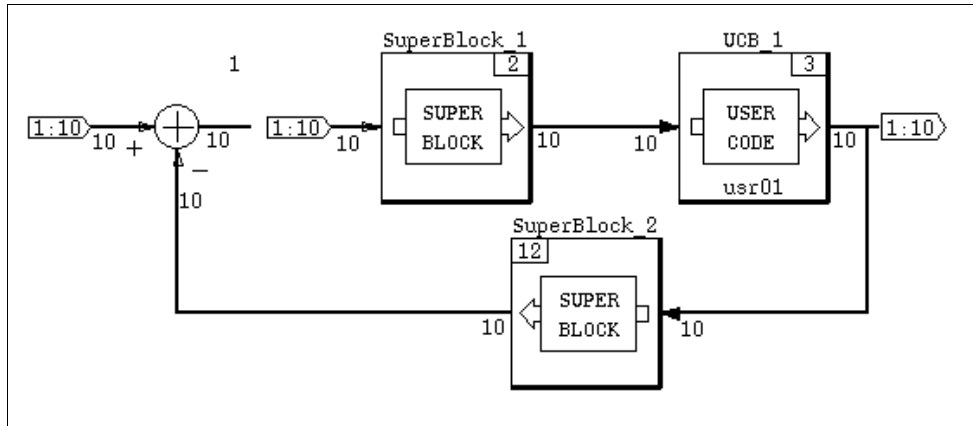
The numbers just get bigger until the ---FIXUP OVERFLOW--- message occurs. The starting and finishing values for C will never agree, and the system runs away.

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. A UserCode block (UCB) is a block that accepts a number of inputs and generates corresponding outputs (see [Figure 8-4](#)). On initialization, however, the situation becomes more problematic. For example, if UCB_1 has no direct (or feedthrough) terms, then it might be necessary for the system to evaluate the outputs of the UCB before it evaluates either SuperBlock_1 or SuperBlock_2. If so, you probably need to furnish initial states for the UCB.

On the other hand, if there *are* direct terms in the UCB, SuperBlock_1 might 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 startup.

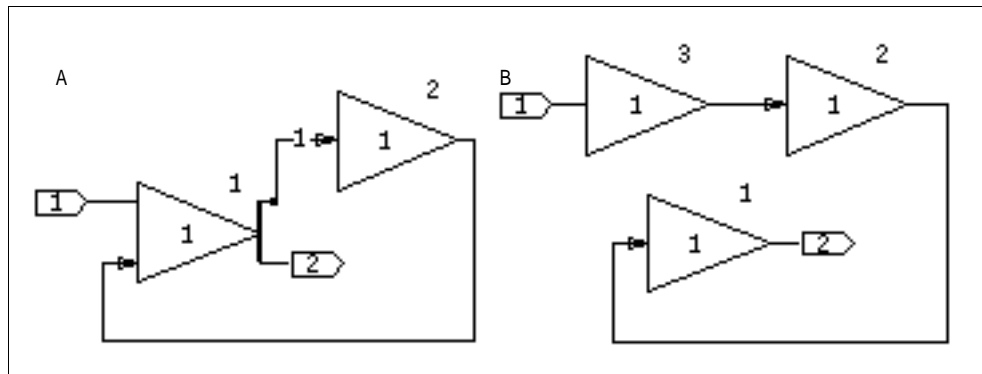
Figure 8-4 Algebraic Loops Initialization Problem



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 might 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 [Stiff System Solvers \(DASSL and ODASSL\)](#) on [p.303](#) for more on DASSL.

Connecting one of the outputs of a SIMO block to one of its inputs² results in an algebraic loop being detected. See Figure 8-5 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 delay is inserted in the loop and the output of the blocks at A lags the outputs at B by one cycle.

Figure 8-5 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 your responsibility to provide consistent values for **yimp0**.

8.4.2 Using the **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, it is an essential tool for debugging your models before simulation. The **analyze()** function returns

2. They have to be connected through other blocks because the SuperBlock 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.

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 simulator automatically invokes the **analyze()** function on newly edited models; it 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, you must explicitly execute the **analyze()** function from the Xmath command area or from the SuperBlock Editor.

Invoking 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, **analyze()** returns an Xmath list object containing the following information:

SBInfo(1)	Number of inputs
SBInfo(2)	Number of outputs
SBInfo(3)	Number of implicit outputs
SBInfo(4)	Number of states
SBInfo(5)	Number of implicit states
SBInfo(6)	Names of inputs
SBInfo(7)	Names of outputs
SBInfo(8)	Names of implicit outputs
SBInfo(9)	Names of states
SBInfo(10)	Names of implicit states
SBInfo(11)	System attributes: continuous, discrete, hybrid, or multirate
SBInfo(12)	Rates of subsystems, ordered from slowest to fastest

When you use the **subsystem** keyword, **analyze()** returns a list object with the following information:

- subsysInfo(1)** RateArray. The rate array lists the subsystem to which the SuperBlock has been assigned. All continuous SuperBlocks are given the value 0 and all DataStores are given the value -1.
- subsysInfo(2)** Parent Index. The parent index contains the index for the name of the parent SuperBlock in the SuperBlock name array (**subsysInfo(4)**).
- subsysInfo(3)** Block IDs. The block ID array contains the block ID of the SuperBlock reference within the parent SuperBlock.
- subsysInfo(4)** 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 Commands window message area by default:

- The SuperBlock reference map, including:
 - Subsystem number (continuous = 0, 1 = fastest discrete, 2 = second fastest discrete, and so forth)
 - 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 you use the **analyze()** keyword **silent**, **analyze()** generates the output list, but it does not display system inputs, outputs, states, and block names.

The **analyze()** keywords **delaybuf**, **vars**, and **typecheck** are the same as corresponding **sim()** keywords (see 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:

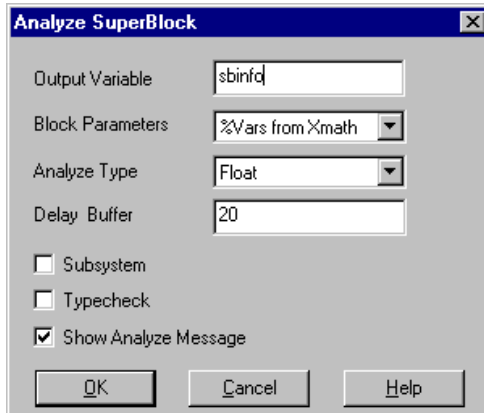
```
SETSBDEFAULT,{typecheck = 1}
```

Invoking analyze from the SuperBlock Editor

To run analyze from the SuperBlock Editor:

Select Tools→Analyze.

The Analyze SuperBlock dialog comes on view. For a complete explanation of the analyze tool, open the dialog and click Help.



8.4.3 Saving Your Model with the 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; it makes a file of the output of the **analyze()** function.

The syntax of **CREATERTF** is as follows:

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

where *model* is the name of a SuperBlock in the SystemBuild catalog. For a full description of **CREATERTF** syntax, type **help creatertf** from the Xmath command area.

8.5 Some Additional Tools

This section discusses several additional tools that you might find helpful: the **simout()** function, and the commands to show and set SystemBuild default values for keywords.

8.5.1 Extracting Dynamic State Values with the **simout()** Function

simout() can extract the dynamic state values (*x*), rates (*xd*), outputs (*y*), and continuous implicit outputs (*yimp*) 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:

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

where *model* 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, type **help simout** from the Xmath command area.

Keywords Specific to **simout()** 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

You can request one, two, three, or four outputs. If you request multiple outputs, 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, they are appended to the *x* and *xd* vectors; thus, you cannot use *x* and *xd* directly in the next simulation.

For the impact of fixed-point arithmetic on *simout*, see [16.6.2 Simout Function](#) on p.446.

8.5.2 Showing and Setting Keyword Default Options

SystemBuild default values are provided for each keyword used by **sim()** and related functions (**lin()**, **simout()**, and so forth). 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 SystemBuild Simulation Parameters 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 is in effect until you change it or exit SystemBuild. For example,

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

sets 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 8-3](#).

Example 8-3 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 = ""}
```

8.6 Simulating Your Model

You can access the SystemBuild simulator from:

- SuperBlock Editor
- Xmath command area
- Operating system command line

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

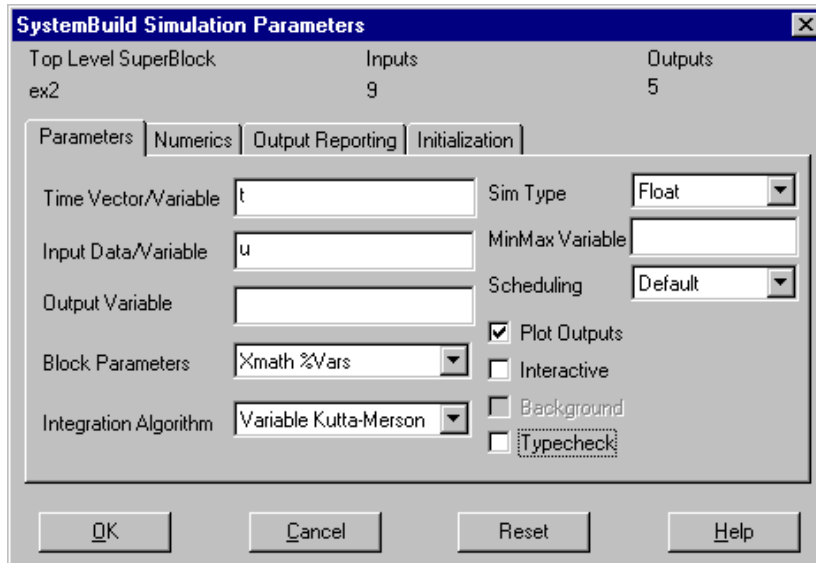
8.6.1 SuperBlock Editor Simulation Interface

To invoke the simulator from the SuperBlock Editor:

Select Tools→Simulate.

The SystemBuild Simulation Parameters dialog box comes on view (see [Figure 8-6](#)). In this dialog, you can enter the time and input variables for your simulation, specify selected simulation keyword options (including fixed-point arithmetic), and enter the output variable name. Click the **Help** button in this dialog for more information about how to select simulation options.

Figure 8-6 SystemBuild Simulation Parameters Dialog



This is the preferred way to invoke the simulator.

8.6.2 Xmath Command Area Simulation Interface

Using the `sim()` function in the Xmath command area 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. You can incorporate the `sim()` function 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 you can run many simulations unsupervised.

Once you invoke the simulator from Xmath, it remains memory-resident so that additional simulations run much faster than the first.

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 area.

To invoke the simulator from the Xmath command line, use one of the following formats:

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

where *model* is a text string, which is the name of the top-level SuperBlock in the SuperBlock Editor.

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)
```

`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 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 32 characters; first character not numeric). For example,

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

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 and 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);
```

You can use existing t and u variables in the simulation dialog.

Given a SuperBlock with three inputs, the following lines 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 don'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. (For example, if you issue three simulation commands and the output variable is y for each one, the ones that finish first are 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—that is, if the simulation is still running—the variable is followed by **busy** and the job number of the related simulation process. See [Example 8-4](#).

Example 8-4 Using who

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 = job_number** (see [Example 8-5](#)).

Example 8-5 **Using stop job**

```
stop job = 1262
Sim is stopped.
```

8.6.3 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 keywords 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.

The syntax for the `sbsim` executable is as follows:

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

The input file must be in MATRIX_X 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 . The default format for the output file is MATRIX_X binary.

You can see how to use the command in [Example 8-6](#).

Example 8-6 **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 is as follows:

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

To output your data in ASCII format, invoke the simulation as follows:

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

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 that 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.

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.

8.7 Terminating Your Simulation

The simulator normally terminates 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, and so forth)
- Error reported by UserCode block
- Stop block encountered with input signal greater than zero

For interactive simulation (ISIM) (see [9. Interactive Simulation](#)), each of these conditions terminates the current simulation; however, you can restart the simulation in the graphical environment. Selecting File→Exit terminates ISIM.

In addition to the above, you can decide to abort a simulation at any time by pressing Ctrl-Break (Windows) or Ctrl-c (UNIX) with your cursor in the Xmath command area as long as the simulator is running in foreground mode (the **bg** keyword is not being used). These key combinations immediately terminate the currently executing simulator; however, saved output data may be lost, and the termination section of UserCode blocks is not invoked.



NOTE: If for some reason you have an active `simexe()` process that does not close properly, you can use `undefine simexe` to terminate it.

8.8 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; the type of block may be indicated.

8.8.1 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 Asin or Acos 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.

8.8.2 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 proceeds, 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 ---

8.8.3 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 Wind River representative.

```
--- OPEN OR DEVICE ERROR ---  
--- INTERNAL IO ERROR ---  
--- REWIND ERROR ---  
--- RESERVED OPERAND ERROR ---
```


9

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.

Using ISIM, you can quickly build up IA diagrams that resemble control panel displays 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 palette, drag it into place, define attributes via an on-screen dialog, 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.

You can execute ISIM as a background task under Xmath, which allows you to enter commands in the Xmath Commands window as the interactive simulation is running.

The primary topics in this chapter are:

- *Interactive Simulation Versus Interactive Animation*
- *Constructing an ISIM Model*

- [Running ISIM](#)
- [Using the Run-Time Variable Editor](#)

9.1 Interactive Simulation Versus Interactive Animation

Interactive simulation contrasts with Interactive Animation (IA), an optional SystemBuild package that lets you build separate standalone control panel displays for monitoring and controlling SystemBuild simulations and the RealSim™ hardware interface. See [Table 9-1](#) for an illustration of the differences between ISIM and IA.

Table 9-1 **ISIM and IA Compared**

Interactive Simulation (ISIM)	Interactive Animation (IA)
Operates inside SystemBuild only. Invoked by calling sim() with the interact keyword, or enabling Interactive on the SystemBuild Simulation Parameters dialog.	Operates standalone only.
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 and stored with the SystemBuild model file.	Icons are in separate .pic files, linked to SystemBuild diagrams. An IA translator converts .pic files to SuperBlocks for RealSim use.
Provided as part of SystemBuild.	Optional.

For a complete treatment of IA, see the *Interactive Animation User's Guide*.

9.2 Constructing an ISIM Model

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.

9.2.1 Using the IA Palettes

The IA icons are invoked from a set of palettes that is available from the SystemBuild toolbar. The IA palettes are discussed in the *Interactive Animation User's Guide*.



NOTE: It is not necessary to include IA icons in your model to take advantage of the other ISIM features (data display, block and time stepping, RVE, and so forth). However, IA icons are the most convenient way to observe or set internal simulation values while the simulation is executing.

To add an IA icon to your SystemBuild model:

1. Click the IA button to raise the palette.
Because SystemBuild is in a modal state when the IA palette is open, you can display only one IA palette at a time.
2. Click one of the boxes at the bottom of the Animation Palette to see a different set of icons.

You can select a specific set of icons by clicking its field directly—for example, clicking MI bring up the Monitor Animation Icons. Some of the icon sets appear on two pages; to change pages, click 1 or 2. Alternatively, you can rotate through the sets of icons by clicking - Icon Set - or + Icon Set +; the latter method does not change pages within a set of icons.

The current set of icons contains asterisks on either side of its field mnemonic, and the name of the set appears in the bottom center field of the palette.

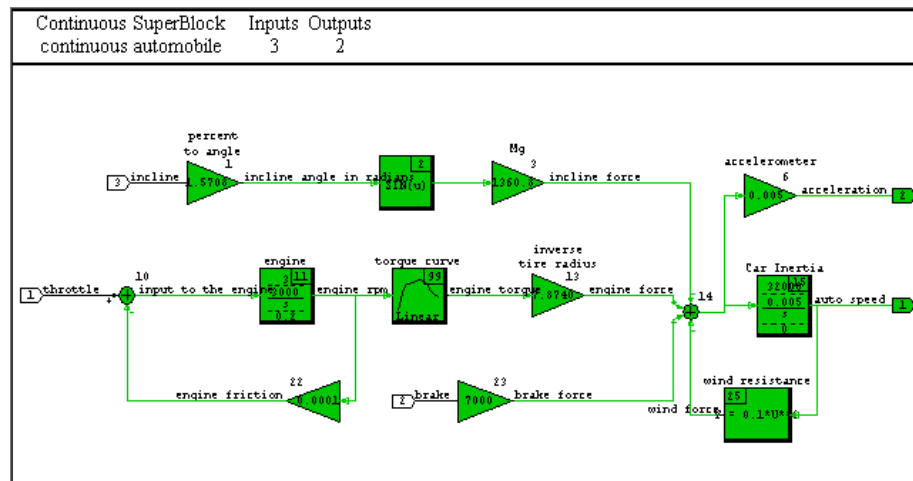
3. Drag an icon into the editor.
4. Connect the icon as though it were a SystemBuild block.
5. To modify its parameters, select the block, and press Return (or Enter) to raise the IA icon dialog.

9.2.2 Building an ISIM Car Model

In this section, we load an existing model. It has 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. In this example, you can manually change the throttle setting, brake position, and road incline while the simulation is running.

To load the model and view it in the editor:

1. Load the file named `$$SYSBLD/examples/auto/cruise_d.cat`.
2. Edit the SuperBlock, `continuous_automobile`, shown below.



To add a output icons to the model:

1. Click the IA button in the SystemBuild toolbar.
The palette of IA icons appears.
2. Select the Single Line strip chart icon from the Monitor Animation Icons, and drag it toward the right side of the screen.

Do not worry about exact placement of the icon: you can move it later.

3. Raise the icon dialog, and change the following fields:

Icon Title to **Speedometer**

Y-axis Label to **Speed**

Minimum Value to 0
Maximum Value at 100 (no change)

Change the color, and then click DONE.

4. When the Speedometer icon appears on the screen, move it to the right side of the display. If you need 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 and then in the Speedometer icon.

Observe that the connection is made.

6. Add an Accelerometer icon. Follow the steps for the Speedometer icon, but give it the name **Accelerometer**, change the Y-axis 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. Change the color, and then click DONE.
7. Move the Accelerometer to a convenient location near the top of the display. Connect the Accelerometer Gain block to the Accelerometer display icon.

To add input icons to the model:

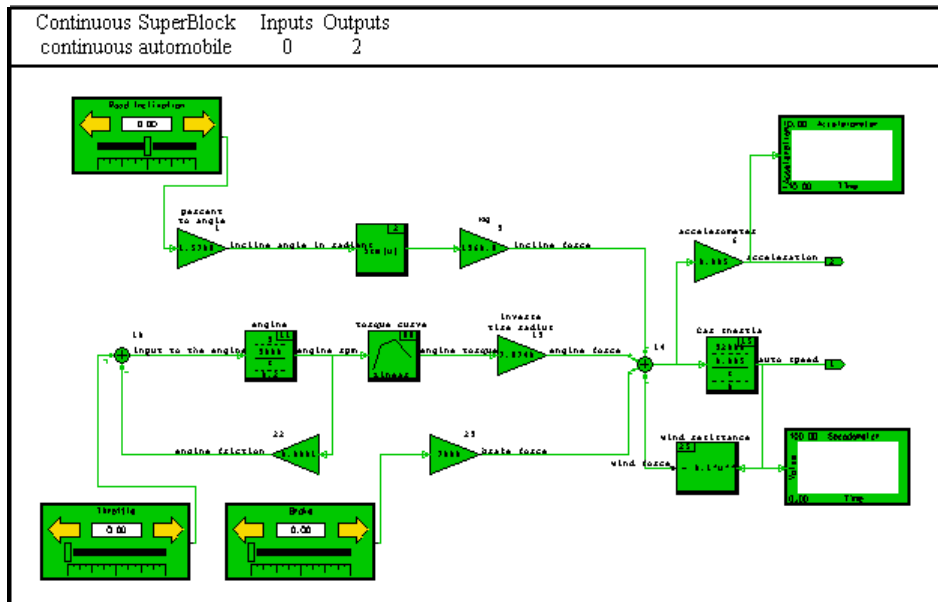
1. Put in a Slide icon to furnish a road inclination. Select the icon from page 1 of the Controller Animation Icons, and give it the name, **Road Inclination**. Change the limits to -10 and +10, and change the Negative Decrement and the Positive Increment to 0.1. Change the color, and then click DONE.
2. Connect the slide switch to the input side of the Percent to Angle Gain block. Click the middle mouse button in the slide switch and then in the Gain block. When the Connections Editor appears, click to connect these blocks, just like any other. Click DONE to release the Connection Editor.

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.

3. Add a brake control by placing another slide switch somewhere near the bottom middle of the screen. Give it a name, **Brake**. Make the limits 0 and 1. Change the Positive Increment to 0.01, which gives a delicate range of brake controls. Change the color, and then click DONE.
4. Connect the brake switch to the brake input Gain block.
Observe that the **brake** external input vanishes.

5. Add a throttle control by placing another slide switch somewhere near the lower-left side of the display. Name it **Throttle**. Change the limits to **0** and **1** and change the Positive Increment to **0.01**. Change the color, and then click **DONE**.
6. Connect the icon to the summing junction at the left side of the engine part of the model.
7. Set the number of external inputs to 0 in the SuperBlock Properties dialog since we are providing inputs through the IA icons.
8. Compare your completed model to [Figure 9-1](#).

Figure 9-1 **Diagram with Input and Output Icons**



9.3 Running ISIM

In this section, we tell you how to run ISIM, what components you can expect in the ISIM window, provide some general information about ISIM, and finally have you run the car model that you completed above.

9.3.1 Keywords and Syntax for Running ISIM

Invoking ISIM

To invoke ISIM from Xmath:

```
sim("model", t_vector, ..., {interact, ...});
```

where the model name and time vector are required. In the above syntax, ... represents any optional input, such as the u vector, and additional simulation keywords.

To invoke ISIM from the SuperBlock Editor:

1. Click Tools→Simulate.
2. Enable the Interactive checkbox, supply a t vector, and any other optional input in this dialog.
3. Click OK.

Invoking ISIM for a Specific SuperBlock

Use the **sbview** keyword to specify a specific SuperBlock. For example, if top is the SuperBlock name, specify:

```
sim("model", t_vector, ..., {interact, sbview="top"});
```

The **sbview** keyword is only available from the **sim()** command in Xmath; it might be discontinued in future versions.

Invoking 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 defaults to non-interactive simulation mode.

```
sim(...,{!interact}) #or  
sim(...,{interact = 0})
```

To make ISIM the default, execute the command:

```
setsbdefault {interact=1}
```

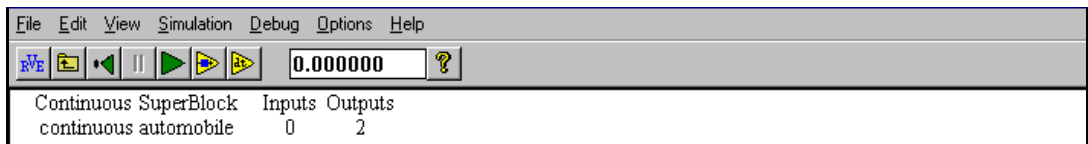
Pausing ISIM at a Non-Zero Time

```
sim(...,{interact, iahold = pausetm})
```

where *pausetm* is in the same units as the time vector of the simulation and must be less than the simulation duration established by the time vector.

9.3.2 ISIM Window

When you start the interactive simulation, the ISIM window comes on view with the SuperBlock that you are simulating. At first glance, it is very much like the SuperBuild Editor; note, however, that the toolbar immediately below the menu bar is different.



You can select ISIM options from the toolbar or from a pulldown menu:

- RVE** Invokes the Run-Time Variable Editor (RVE) that allows you to change the value of a %Variable at a point in an interactive simulation. See [9.4 Using the Run-Time Variable Editor](#) 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	Pause (“hold”) the simulation while it is running, clicking this button temporarily halts (“holds”) the simulation. See also the Hold Time field.
Resume	To start a simulation, or restart a paused simulation, click the Resume button. Once the simulation reaches the end of the time line, all buttons are disabled, with the exception of the Reset button.
Block Step	Show the 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 is highlighted. If you step through an interactive simulation all the way to the end, all buttons are disabled until you click Reset (or you exit the simulation).
Local Block Stepping	If enabled, limits block stepping to the currently displayed SuperBlocks; the simulation does not stop on blocks outside this SuperBlock. 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 runs 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 again. For a way of using hold time, see Example 9-1 .

9.3.3 Special Notes on ISIM

- A principal use of ISIM is for debugging SystemBuild designs. Attach a numeric output icon to any signals of interest (for example, 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 by entering a value in the Sampling Interval field.
- You can click 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 are displayed while you are running the simulation. 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 corner of the ISIM window.
- You can choose global or local block step mode on the Debug menu (option enabled or disabled). If global, the Block Step button causes 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 resumes the simulation and pauses 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.
- 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 (or Enter). 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 and then repeatedly press the ' (apostrophe) key to cycle through the colors.
- You can add user-written icons and palettes of icons to the IA palettes. For more on this subject, see the *Interactive Animation User's Guide*.

9.3.4 Simulating the Car Model

To simulate the car model using ISIM:

1. In the Xmath command area, create a time vector large enough to give you a little time at the wheel:




```
t = [0:0.1:1000]';
```

Depending on the speed of your system, the number of time points specified might not be enough to exercise the model adequately; change this parameter as necessary.

2. Run the simulation on the modified model using the `sim()` command:

```
y = sim("continuous automobile", t, {interact});
```

You can observe a display of your ISIM model in the Interactive Simulator window.

3. 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 Resume button. Move the throttle slide forward gingerly and learn to drive all over again.
6. When you are ready to stop, click the Pause button. After you pause, select File→Exit to exit simulation and return to Xmath.

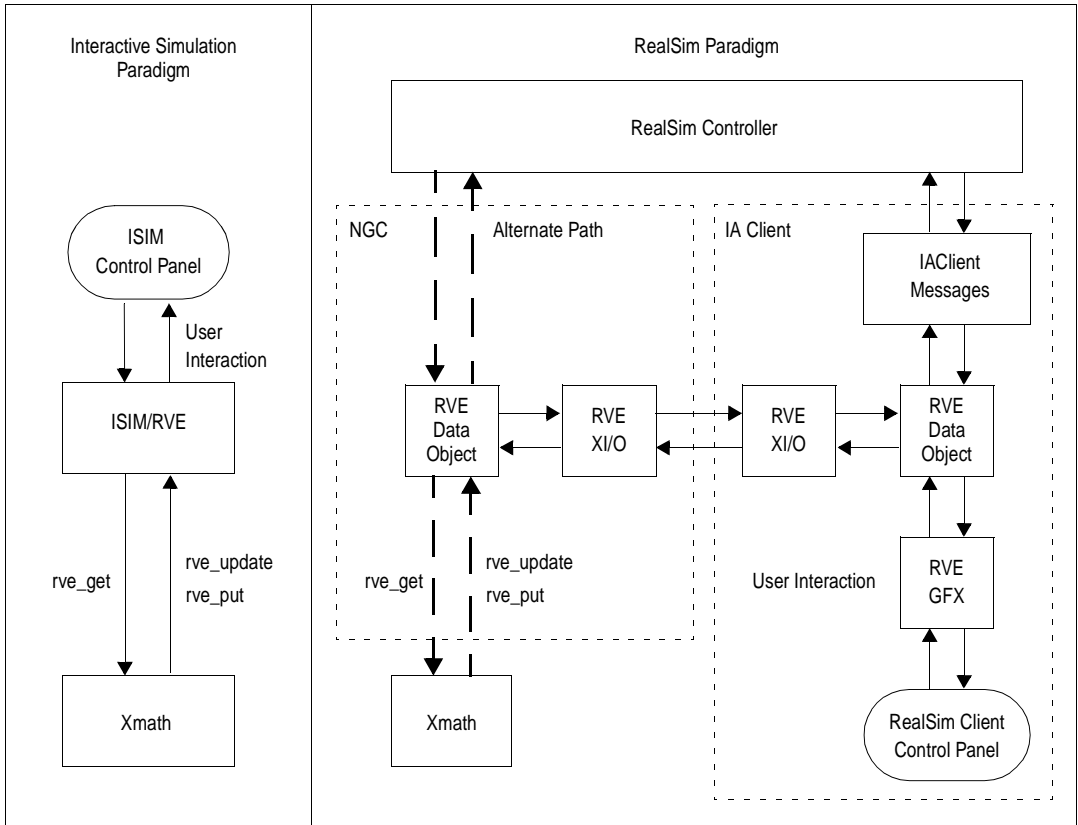
9.4 Using the Run-Time Variable Editor

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 RVE 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 [Changing Parameters for Repeated Simulations](#) on p.176.

[Figure 9-2](#) shows the connections of RVE in ISIM and RealSim systems. In the ISIM context within SystemBuild, as shown in the left side of [Figure 9-2](#), the RVE software is part of the ISIM program, and you interact with both ISIM and RVE through the ISIM toolbar. As shown in the right side of [Figure 9-2](#), 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 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.

Figure 9-2 RVE in ISIM and RealSim Contexts



9.4.1 RVE and ISIM

This procedure explains using RVE from ISIM. The term *run-time 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 9-2,) 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.

To use the Run-Time Variable Editor:

1. Prepare your model for simulation.

You must have one or more run-time variables in the model, although having ISIM icons in the model is optional.

2. Copy your run-time variable into the Xmath data area and give it an initial value.

If you are working with %Variables, use one of the following ways to accomplish this:

- a. When you enter the %Variable, press Ctrl-p on the keyboard, and SystemBuild enters the variable automatically.
- b. Enter it explicitly by typing the variable name in the Xmath command area and setting it equal to the initial value for your simulation.

Either way, the %Variable acquires an initial value as required by RVE.

If you are using Variable block variables, SystemBuild initializes them; if you have given them no value, the default is 0.

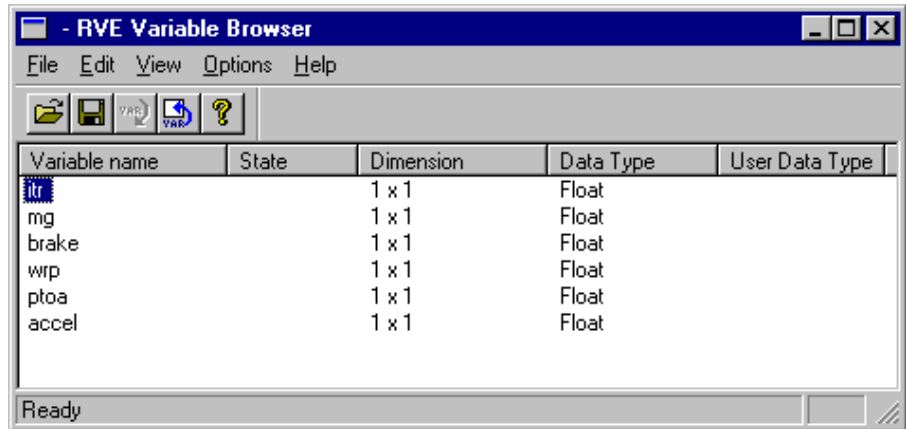
3. When you run the simulation, make sure that the **interact** keyword is set to TRUE.

You can enable Interactive on the Parameters tab of the SystemBuild Simulation Parameters dialog or by including the **interact** keyword in a **sim()** command issued from Xmath. Because you are simulating interactively, the Interactive Simulator window appears when you execute the **sim()** command.

4. Invoke RVE selecting Edit→Variables or by clicking the RVE button on the ISIM toolbar.

The simulation may be executing during the editing process, or it may be paused. [Figure 9-3](#) shows a typical Run-Time Variable Editor.

Figure 9-3 Run-time Variable Editor



5. To change a column width, click a divider and drag it left or right. To edit a variable, double-click its name, or select the name, and then select Edit→Open.

You may add new values into the variable spreadsheet or define them from the Xmath command area.

6. After you have edited the variable, click OK.

This causes the RVE to be displayed again. You may now select and change another run-time variable; there is no limit to the number of run-time variables you can change.

7. When you are finished adjusting the variable values, use Edit→Select Modified to select the changed variables. Then select Edit→Download, or click the Download button on the browser toolbar to complete the edit.

The new run-time variable values are available immediately. If the simulation is running while the edit is taking place, the edit applies to the next **sim** time step.

[Example 9-1](#) gives a step-by-step example of using RVE.

Example 9-1 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 and then stop and change the value of that parameter.

To use RVE with the **pred-prey** model:

1. To copy the data to your current working directory, type:

```
copyfile "$SYSBLD/examples/pred_pre/pred_pre.cat"
```

2. Load the file. From the Catalog Browser, open Predator_Prey.
The Efficiency_factor Gain block is parameterized via the %k parameter.
3. To initialize this gain, define it in the Xmath command area:

```
k = .5;
```

4. Enter values for t (time) and u (input):

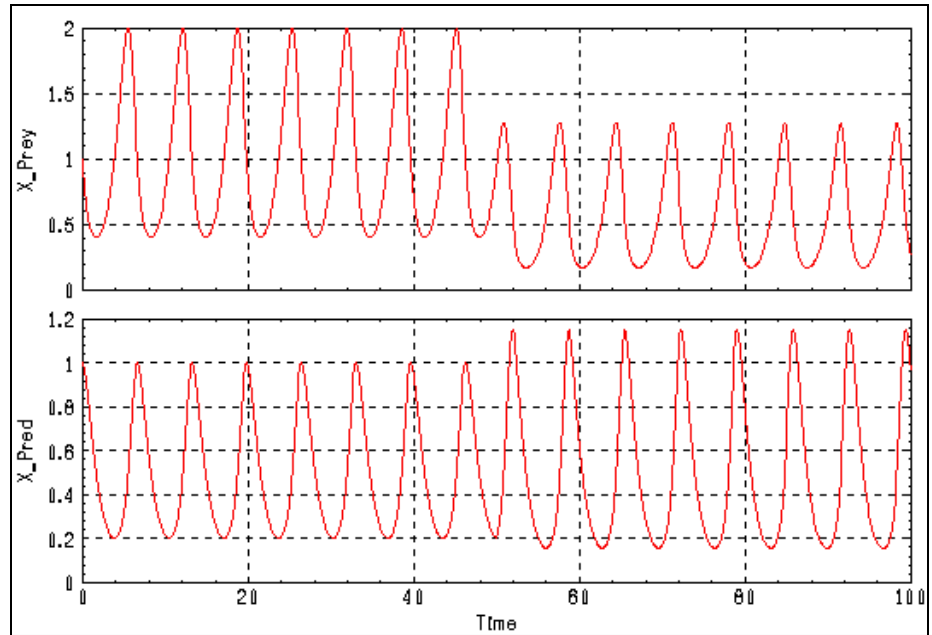
```
t = [0:.1:100]';  
u = [ones(t)];
```



NOTE: Select Tools→Simulate to raise the SystemBuild Simulation Parameters dialog. Type t in the Time Vector/Variable field and u in the Input Data/Variable 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), and then click the Resume button (the green triangle).
Observe that the Time field in the lower-right corner of the ISIM window advances until it reaches 50 and then stops.
6. Click the RVE button.
7. In the Run-Time Variable Editor, double-click k . Change the value of k from .5 to .9. Press OK.
8. Back in the Run-Time Variable Editor, select k , and then press the Download button.
9. Back in the ISIM main window, click Resume.
Observe that the Time indicator advances to 100 and then stops.
10. Select File→Exit to return to the SuperBlock Editor.
The plot produced should look like [Figure 9-4](#).

Figure 9-4 Plot with Run-time Variable Editing Illustrated



An examination of [Figure 9-4](#) 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. Increasing the **Efficiency_factor**, k , at 50 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 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.

9.4.2 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 run ISIM in the background, specify one of the following:

```
sim(...{bg})  
setsbdefault, {bg = 1}
```

These commands are treated in more detail in online Help. To see a complete listing of the RVE commands in online Help, type **help rve** in the Xmath command area, or select the RVE topic from the SystemBuild Help area. 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 a 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 a run-time variable from the RVE workspace.

```
rve_put("var_name", var_val);
```

Assign the value *var_value* to run-time 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 a run-time variable. If a variable has been modified via the **rve_put** command and has not been updated yet, the **rve_reset** command resets the working copy of the run-time 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 run-time variables are specified, then only those variables are updated.

- rve_info** Retrieves and displays the names of all the run-time 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 run-time variable *output* is locked while you are attached to RealSim.

9.4.3 RVE-Compatible Blocks

Table 9-2 shows the blocks that you can use with RVE. Most %Variables on most blocks are compatible with this feature.

Table 9-2 **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 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	

Table 9-2 **Blocks Supported in RVE** (Continued)

Block Type	Supported Parameters	Unsupported Parameters
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)	
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	

In most cases, the following blocks are not supported in RVE 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	ZeroCrossing

10

Linearization

The `lin()` function is used to linearize a continuous, single rate or multirate discrete, 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 attributes¹, 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()` function are:

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

The required input *model* is the name of the SystemBuild model to be linearized. Keywords are optional, except in the implicit case, where the keyword **implicit** must be present. To see a full description of each keyword, type **help lin** from the Xmath command area.

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.

1. Computational attributes are defined as the timing attributes and requirements of the SuperBlock.

The implicit form of linearization is:

$$\begin{aligned} E\dot{x} &= Ax + Bu \\ y &= F\dot{x} + Cx + Du \end{aligned}$$

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) =
      (9) =
     (10) =
     (11) =
```


10.1 Linearizing Single-Rate Systems About an Initial Operating Point

10.1.1 Continuous Systems

Continuous systems are represented by the following non-linear differential and output equations:

$$\begin{aligned} \dot{x} &= f(x, u) \\ y &= g(x, u) \end{aligned} \quad (10-1)$$

where 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:

$$\begin{aligned} (0 &= f(\dot{x}, x, u)) \\ (y &= g(\dot{x}, x, u)) \end{aligned}$$

and the linearization becomes:

$$\begin{aligned} [E\dot{x} &= Ax + Bu] \\ (y &= F\dot{x} + Cx + Du) \end{aligned}$$

where

$$E = -\left. \frac{\partial}{\partial \dot{x}}(f) \right|_{\dot{x}_0, x_0, u_0} ; A = \left. \frac{\partial f}{\partial x} \right|_{\dot{x}_0, x_0, u_0} ; B = \left. \frac{\partial f}{\partial u} \right|_{\dot{x}_0, x_0, u_0} \quad (10-2)$$

$$F = \left. \frac{\partial g}{\partial \dot{x}} \right|_{\dot{x}_0, x_0, u_0} ; C = \left. \frac{\partial g}{\partial x} \right|_{\dot{x}_0, x_0, u_0} ; D = \left. \frac{\partial g}{\partial u} \right|_{\dot{x}_0, x_0, u_0} \quad (10-3)$$

- Time is assumed to be zero.
- For continuous systems, algebraic loops are resolved—that is, **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. If you do not want algebraic loops to be resolved, specify **algoalp = 0**. Note that algebraic loop delays appear as additional states when this is done.

10.1.2 Discrete Systems

Each discrete subsystem is represented by the following difference equations:

$$\begin{aligned}x_{k+1} &= f(x_k, u_k, z_{k+1}) & X_k &= f(X_k, U_k, Z_k) \\z_{k+1} &= h(x_k, u_k, z_{k+1}) & Z_k &= h(X_k, U_k, Z_k) \\y_k &= g(x_k, u_k, z_{k+1}) & Y_k &= g(X_k, U_k, Z_k)\end{aligned}\tag{10-4}$$

where z represents the algebraic loop variables (if any) in the model. When **algoalp = 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 **algoalp = 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, as before, the equations become 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.

10.2 Exact Versus 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 are specified via the keywords **du**, **dx**, **dxdot**, the linearization defaults to a central finite-difference linearization.

10.3 Special Linear Models

10.3.1 Continuous Time Delay

For **lin()**, the continuous time delay is modeled by a state-space representation using a Padé approximation:

$$H(s) = \frac{N(s)}{D(s)}$$

with the 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 of the time delay block.

The initial conditions for the states in the continuous time delay model are assumed to be zero; however, you can change them from the dialog. Use the **analyze()** function to determine the names of the states; **analyze()** returns a list object in which the state names are the ninth element:

```
SBinfo = analyze("model", {keywords});  
snames = SBinfo(9)?
```

10.3.2 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.*

10.3.3 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.

10.3.4 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.

10.3.5 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 [14. UserCode Blocks](#)). If the template linearization is not in `usr01`, the default linearization is by finite differences.

10.3.6 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.

10.4 Linearizing Single-Rate Systems 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 is 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 **resumefrom=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})
```

10.5 Multirate Linearization

MATRIX_x multirate linearization is based on one of two methods, both of which return a single-rate discrete linear system:

Kalman-Bertram method — is based on simulations over the basic time period (BTP), which is defined as the least common multiple (LCM) of all subsystem sampling intervals. For multirate systems with rates that are integer multiples, the basic time period is the slowest rate in the system. The BTP is also the sampling interval of the resulting model. The highest frequency in the model is 0.5/BTP, which is less than or equal to the highest frequency of all subsystems. The states of the new single-rate system correspond to the continuous states and all the discrete states appended together.

Subsystem method — is based on linearization and rate transformation of individual subsystems. The resulting model generally has more states than the original one as a consequence of the methods used for rate transformation and the appending of delay filters. The highest frequency in the model is at least as large as the highest frequency of all subsystems.

For additional information on multirate linearization, see [A80] and [G83].

10.5.1 Kalman-Bertram Method

In the Kalman-Bertram method, 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 $dx = 0.001 * (1 + abs(x_0))$, 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**, and **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 because signals in discrete systems are not defined between samples. **btptol** is a **lin()** keyword with a default value of 0.001. In this implementation, the basic time period starts at zero and all skews must be less than $1e-6$. However, the sample rates may be asynchronous. **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})?
```

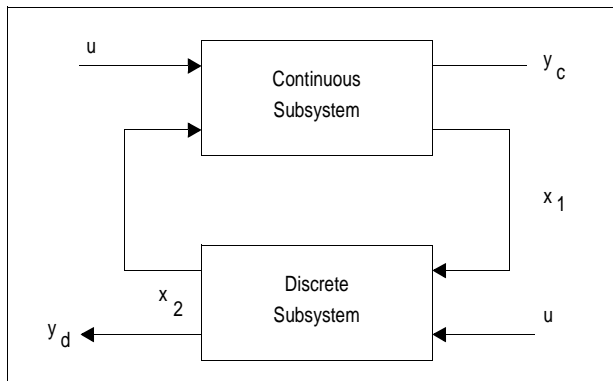
The simulation must end at t , which should be a multiple of the basic time period, to obtain a correct multirate linearization result.

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, you can ask questions 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 10-1](#).

Figure 10-1 **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:

1. Compute the state transition matrix from $t = 0$ to $t = T^-$ for the continuous system.
2. Compute the state transition matrix for the discrete system from $t = T^-$ to $t = T^+$. The continuous state equation is computed as follows:

$$\begin{matrix} & & & \mathbf{F \ Matrix} & & \\ & & & \overbrace{\hspace{10em}} & & \\ \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{u} \end{bmatrix} & = & \begin{bmatrix} A_c & B_c B_{11} & B_c B_{12} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} x_1 \\ x_2 \\ u \end{bmatrix} \end{matrix}$$

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:

$$\begin{matrix} & & & \mathbf{G \ Matrix} & & \\ & & & \overbrace{\hspace{10em}} & & \\ \begin{bmatrix} x_1(T^+) \\ x_2(T^+) \\ u(T^+) \end{bmatrix} & = & \begin{bmatrix} I & 0 & 0 \\ (B_d B_{21}) & A_d & B_d B_{22} \\ 0 & 0 & I \end{bmatrix} & \begin{bmatrix} x_1(T^-) \\ x_2(T^-) \\ u(T^-) \end{bmatrix} \end{matrix}$$

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:

$$\Phi = Ge^{FT}$$

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, you can observe that for linear systems, the Kalman-Bertram method gives 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 responds to the pulse. However, a single-rate system can miss the pulse if it occurs between samples. Following this reasoning, the Kalman-Bertram method is not 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 is not well matched. For additional resources, see [A80] and [DoFrTa92].

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)
```

More generally, one can use any piece-wise constant input sequence (where the interval length is equal to the BTP) for verification.

10.5.2 Subsystem Method

This method is based on linearization and rate transformation of the subsystems individually. In order to match the phase shift related to the sample-and-hold characteristics of the individual subsystem models, filters are appended; these filters cause additional states to be added. This type of model is useful for analysis purposes over a bandwidth that includes all subsystem bandwidths.

There are two choices of rate transformation that are selectable by the **subsys_trf** keyword:

- Constant delay rate transformation {**subsys_trf=0**}

Given a transfer function $H_1(z_1)$, where z_1 corresponds to the forward shift operator for sampling interval Δt_1 , a delay preserving transformation to a sampling interval $\Delta t_2 = \Delta t_1/N$ can be found by defining $H_2(z_2) = H_1(z_2^N)$. The transformed model has N times as many states as the original one. This leads to frequency folding in the transfer function. Fortunately, that effect is generally eliminated by the delay filters that are appended to the transformed subsystem model. The original subsystem model outputs have an average delay of $(\Delta t_1 - \Delta t_2)/2$ compared to the new one. An important advantage of this method is that this delay is independent of the frequency content of the input signal and of the model characteristics.

- Impulse invariant rate transformation {**subsys_trf=1**}

This transformation is a generalization of the well known impulse invariant transformation from continuous to discrete time. The transformation of a discrete system to a faster rate can cause problems for systems with poles on the negative real axis. Such poles are split into a pair using a perturbed modal decomposition model. The number of states added is generally much smaller than with the constant delay method. Unfortunately the average delay compared with the original subsystem model depends on the system dynamics, which makes this method less reliable than the constant delay method. Therefore, you should always validate the result with the original SystemBuild model or the constant delay method.

There is also a choice of two types of filters (keyword **subsys_fil**) that are appended to the subsystems to emulate sample-and-hold delays of the original subsystems:

- Moving average (MA) {**subsys_fil=0**}.

The number of filter states required per subsystem output is $2 * delay$, where *delay* is the required delay specified in number of sampling intervals.

- Auto-regressive (AR) {**subsys_fil=1**}.

Only one filter state required per subsystem output.

AR is much more efficient than MA but less exact for the purpose of introducing a constant delay.

The constant delay rate transformation with the MA filter is the default method; it is the most reliable but can easily lead to an excessive number of states, whereas

the impulse invariant rate transformation with the AR filter is the most efficient but also the least reliable.

The sampling interval of the resulting model with the constant delay method is always the greatest common divisor (GCD) of all sampling intervals, whereas the sampling interval with the impulse invariant method is the smallest sampling interval of all discrete subsystems. For the impulse invariant method, however, you can override the sampling interval by specifying the **subsys_dt** keyword.

The keywords **actiming** and **cdelay** are optional and result in higher orders of the filters that emulate the subsystem sample-and-hold effects.

With the subsystem method, models containing subsystems that cannot be linearized in explicit form are not allowed. Enabled subsystems are assumed to be in the enabled state. Triggered subsystems are ignored.

10.5.3 Linearizing Fixed-Point Blocks

Fixed-point block linearization is performed the same way, 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 [16. Fixed-Point Arithmetic](#)).

11

Operating Point Computation

The **trim()** function 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. The system to be trimmed must have at least one state, but no external inputs are required.

trim() 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.

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 **xdot** 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:

$$\dot{x}_t = \frac{x(k+1) - x(k)}{T}$$

11.1 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, trimtol})
```

where *model* specifies the model to be processed and is required.

The following are optional keyword inputs

- | | |
|------------------|---|
| xdt | State derivative vector at which the system is to be trimmed of dimension n_x by 1. Default = all 0. |
| xdt_float | List of integers specifying the floating (unconstrained) derivatives. The system is trimmed at zero equilibrium by default. The constraints on derivatives corresponding to these are removed from the trim() equations to be solved. trim() usually detects "free integrators" and informs the user. In some cases, the trim() solution may improve if indices of these free integrators are included in xdt_float . Default = null. |
| u0 | Nominal input vector of dimension n_u by 1 or n_u by 2. Default = all 0. |
| u_frz | List of integers specifying inputs to be frozen. Default = null. |
| x0 | Nominal initial state vector; defaults are taken from the SystemBuild catalog. |

x_frz	List of integers specifying states to be frozen. Default = null.
y0	Nominal output vector.
y_frz	List of integers specifying outputs to be frozen. Default = null.
yimp0	Nominal implicit output vector for systems with algebraic loops.
dx	Real (default 0.001). State perturbation vector for linearization.
du	Real (default 0.001). Input perturbation vector for linearization.
iter	Integer. Number of trim() iterations. trim() continues improving the solution until either the tolerance criterion is satisfied (see trimtoll below) or the maximum number of iterations specified with iter is exceeded.
trimtoll	Real (default 100*eps; eps=machine epsilon). Convergence criterion for trim() iterations. Convergence is assumed when the Euclidean norm of the constraint vector xdt (or [xdt ; y] if some outputs are frozen) is less than eps . In some cases, this value may be too strict, and it may have to be relaxed.

The outputs are as follows:

xt	Trimmed state vector
ut	Trimmed input vector
yt	Trimmed output vector
yimpt	Trimmed implicit output vector

If **u0** is not specified as an input, it is initialized to zero. If **y0** is not specified, then **simout()** is used to compute it from **x0** and **u0**. If **x0** is not specified as an input, it is initialized from the SystemBuild catalog values.

u_frz, **x_frz**, and **y_frz** designate components of the external input, state, and output vectors that are to be held constant. **y_frz** can be used for **trim()** problems where operating point states and their derivatives are known but the inputs are unknown. Constraining outputs is preferable to constraining states because the mapping from states to outputs is often not known to the user (for example, in transfer function blocks). Note, however, that for some overly constrained problems, it may not be possible to satisfy **y_frz** exactly because output values at the **trim()** point are constrained by $y = g(x_t, u_t)$.

The **trim()** function uses the **lin()** function to linearize a system. Therefore the necessary initial conditions and inputs for the operating point must be specified with the **x0** and **u0** keywords.

Once the trimmed state values are computed, they may be inserted as initial conditions for a subsequent simulation or linearization. If the trimmed point is indeed a steady-state operating point, the output shows no transients.

11.2 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:

$$\begin{aligned}\dot{x} &= \dot{x}_t \\ y &= y_f\end{aligned}$$

y_f represents the constrained ("frozen") components of the output vector. To describe the algorithm, we use a continuous nonlinear system:

$$\dot{x} = f(x, u)$$

We wish to satisfy the following at the steady-state operating point

$$\begin{aligned}\dot{x}_t - f(x, u) &= 0 \\ y_f - g_f(x, u) &= 0\end{aligned}$$

subject to x_f, u_f as specified by the user.

Linearization of the system and output equations yields:

$$\begin{aligned}\frac{\partial f}{\partial x} \Delta x + \frac{\partial f}{\partial u} \Delta u &= 0 \\ \frac{\partial g}{\partial x} \Delta x + \frac{\partial g}{\partial u} \Delta u &= 0\end{aligned}\tag{11-1}$$

At the operating point, if we set

$$\begin{aligned} A &= \frac{\partial f}{\partial x} \\ B &= \frac{\partial f}{\partial u} \\ C &= \frac{\partial g}{\partial x} \\ D &= \frac{\partial g}{\partial u} \end{aligned}$$

the result is:

$$\begin{bmatrix} \tilde{A} & \tilde{B} \\ \tilde{C} & \tilde{D} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta u \end{bmatrix} = 0 \quad (11-2)$$

The matrices \tilde{A} , \tilde{B} , \tilde{C} , and \tilde{D} are found by linearizing the system with the `lin()` function. \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 (11-1) and (11-2) as:

$$\begin{bmatrix} \tilde{A} & \tilde{B} \\ \tilde{C} & \tilde{D} \end{bmatrix} \begin{bmatrix} x_{i+1} - x_i \\ u_{i+1} - u_i \end{bmatrix} = \begin{bmatrix} \dot{x}_i - f(x_i, u_i) \\ y_{fi} - g_f(x_i, u_i) \end{bmatrix} \quad (11-3)$$

The iterations continue until convergence is achieved:

$$\begin{bmatrix} x_{i+1} \\ u_{i+1} \end{bmatrix} \cong \begin{bmatrix} x_i \\ u_i \end{bmatrix}$$

11.3 *trim()* Behavior

11.3.1 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 diverges from such trimmed points. To check for stability, inspect the eigenvalues of the system obtained by linearizing the system at the trimmed operating point.

11.3.2 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`.

11.3.3 Algebraic Loops

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, yimpt] = trim("model", {xdt, xdt_float, u0, u_frz, x0,  
    x_frz, y0, y_frz, yimp0, dx, du, iter, trimt01, message})
```

where *model* is the name of a SuperBlock in the SystemBuild Editor and

yimp0 Nominal implicit output vector for the algebraic loop outputs.

yimpt 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=yimpt})
```

11.4 *trim()* Examples

Example 11-1 Simple Linear Model

In this example, we trim a simple linear model called Flat Model. This system has eleven states, three inputs, and four outputs.

To trim Flat Model:

1. Copy the model to your working directory:

```
copyfile "$SYSBLD/examples/f14/f14_2.cat"
```

2. Load the file.
3. Use the following `trim()` function:

```
[xt,ut,yt]=trim("Flat Model",{u0=[2; 2; 2], u_frz=[1, 2, 3]})
```

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 (11-3), and so the trim problem is well-posed:

You can verify the results of the `trim()` operation 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 11-2 Over-Constrained Simple Linear Model

The following example is over constrained. The `trim()` function gives eleven unknowns (solving only for states; inputs are frozen), and $11 + 4 = 15$ constraints (four outputs are 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 gives a least-squares solution but cannot attain zero error.

Example 11-3 **Nonlinear Model**

The Predator_Prey model provides an example of a nonlinear `trim()` problem.

To trim `pred_prey`:

1. Copy the model file to your local directory:

```
copyfile "$SYSBLD/examples/pred_prey/pred_prey.cat"
```

2. Load the model.
3. Call the `trim()` function:

```
[xt,ut,yt]=trim("Predator_Prey",{u0=1, x0=[2;2], u_frz=1, iter=5})
```

You can verify the steady-state operating point 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 $xt = [0;0], [1;0.5]$.

12

Classical Analysis Tools

This chapter describes the user interface to the analysis tools that you can use from the SuperBlock Editor. These tools provide an easy-to-use graphical interface for analysis of the current model using the **analyze()** function.

After discussing how to use the tools generally (*Using the Tools*) and what you can expect to see after you invoke the tool (*How SystemBuild Proceeds to Analyze Your Model*), this chapter focuses on the classical analysis tools, which are:

- *Open-Loop Frequency Response*
- *Time 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 multirate), 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 are automatically linearized using the multirate linearization method. The syntax of the **lin()** function is explained in detail in online Help and in *10. Linearization*.

12.1 Using the Tools

If a system contains a hierarchy of SuperBlocks, you must perform the analysis with the top-level SuperBlock open in the SuperBlock Editor window.

To use a tool:

1. If you want to use the Time Response, Open-Loop Frequency Response, Point-to-Point Frequency Response or Root Locus tool, select an input block and an output block. To perform a multiple selection, select the input block, and then hold down the Ctrl key and select the output block.

If you want to use the Parameter Root Locus tool, you do not have to select anything in the model.

Depending upon what you select in the diagram, the appropriate tools are enabled on the Tools menu.

2. Select the desired tool from the Tools menu.

A dialog appears. You supply the appropriate parameters, as explained in the sections for each of the functions below. Press the Help button on the dialog for additional information on using the tool.



NOTE: Whenever you enter a new value in a dialog field you must press Return (or Enter) in that field to ensure that the value is read.

12.2 How SystemBuild Proceeds to Analyze Your Model

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`.¹ This system is modified depending on the input and output blocks that you selected and whether the mode of analysis is open-loop or point-to-point.

You can edit the modified SuperBlock, `_Analysis_System`, and use it like any other SuperBlock except that you cannot use analysis tools on it. `_Analysis_System` is displayed in the catalog listing of SuperBlocks.

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. Although the area of interest may be single rate, if a subsystem with a different rate is part of the original model, multirate linearization is invoked and may give unexpected results. If this presents 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 dialogs and by the external input entered in the analysis dialog.



NOTE: The external input has the reserved name `sb_uext`. This name should not be used by any user-defined Xmath variables, or they are overwritten.

4. Finally, the appropriate Xmath function is invoked, and the results are plotted.

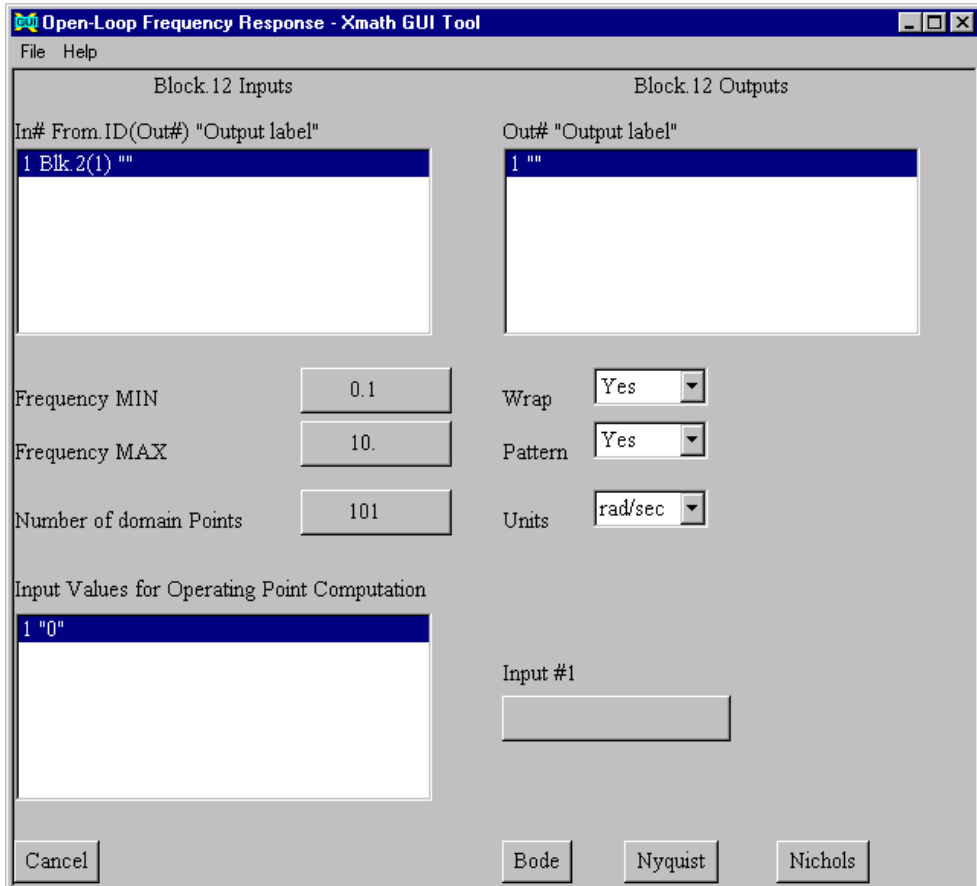
You can follow the progress of the function execution by checking for diagnostic or error messages in the Xmath Commands window.

1. 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.

12.3 Open-Loop Frequency Response

As soon as you have selected one or two blocks 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 appears (see [Figure 12-1](#)).

Figure 12-1 Open-Loop Frequency Response Dialog



NOTE: The input block is the first block selected and the second block is the output block. If you select only one block, the block is used for both the input and output block.

The inputs area at the top of the dialog lists the input channels of the selected input block. Correspondingly, the outputs area at the top of the dialog shows output channels of the selected output block. You can select one channel each for the open-loop system's input and output. This defines 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.



NOTE: Remember to press Return (or Enter) every time you edit a value.

- The Number of domain points field defines the number of frequency values at which the plot is calculated.
- The Input Values for Operating Point Computation region is a table listing the values of the external inputs with respect to which the linearization is performed (default values are zero). Use the edit fields to the right of the list to change the value.

Setting the Pattern option causes SystemBuild to draw constant M and N contours on the plot. Setting Wrap limits the phase to $\pm 180^\circ$. Refer to the *Control Design Module User's Guide* for further details.

You can perform one of three frequency analyses: Bode, Nyquist, or Nichols. As soon as you select one of the analysis types in the dialog, SystemBuild creates the reserved SuperBlock `_Analysis_System`, which contains 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 appears with the desired analysis result.

You can apply the open-loop frequency response analysis 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.

[Example 12-1](#) provides an opportunity for you to use the tool.

Example 12-1 **Getting a Bode Plot of classical.cat**

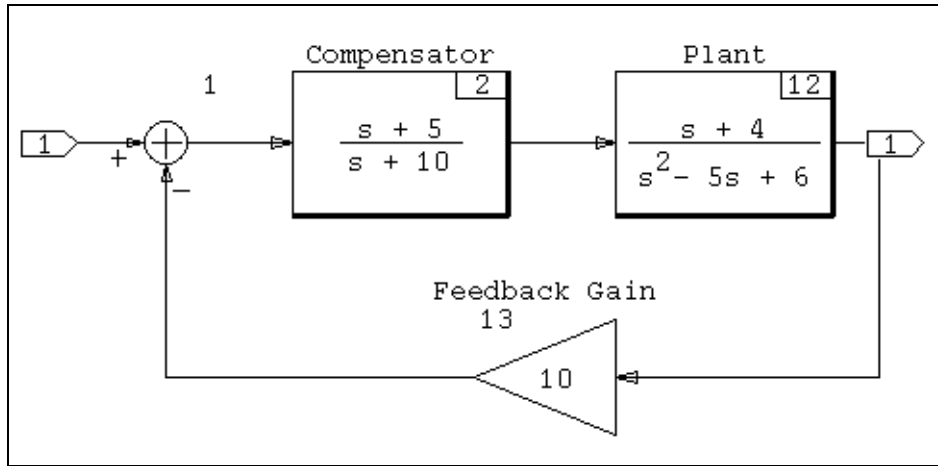
To perform an open-loop frequency analysis on **classical.cat**:

1. Copy this file to your local directory:

```
copyfile "$SYSBLD/examples/classical_example/classical.cat"
```

2. Load the file into SystemBuild, and then display classical 1 in a SuperBlock Editor (see [Figure 12-2](#)).

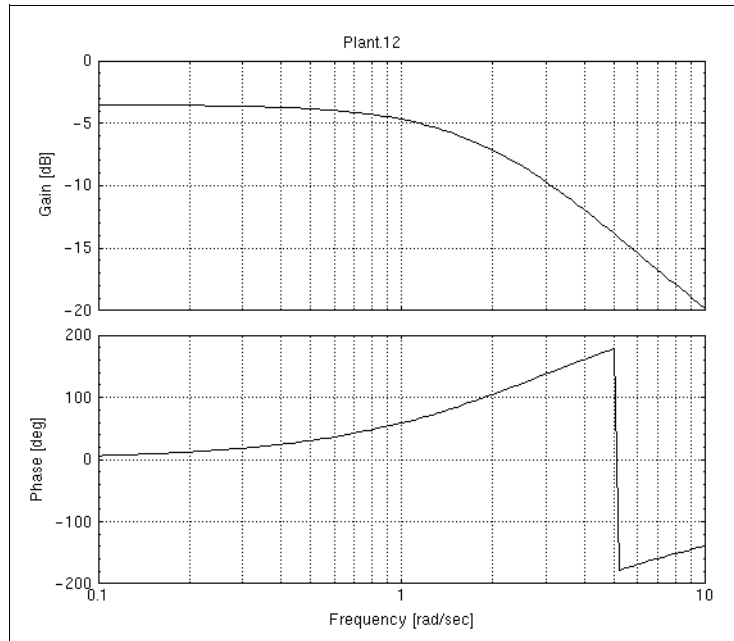
Figure 12-2 **Open-Loop classical 1 System Before Editing**



3. To obtain the Bode plot of the open-loop Plant transfer function, select Plant. In this case, we want to use Plant as both the input and output, so we select only that block.
4. Select Tools→Open-Loop Frequency Response. The Open-Loop Frequency Response dialog appears.
5. Click Bode.

Figure 12-3 shows the resulting plot.

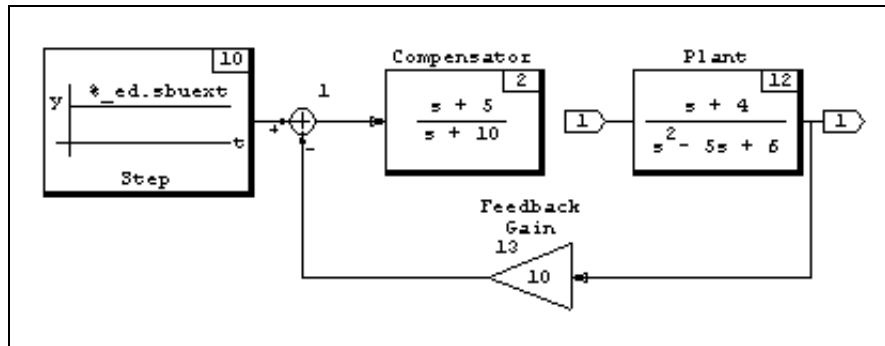
Figure 12-3 **Open Loop Bode Plot (Typical)**



6. Close the plot window, and upload the model to the Catalog Browser.

The open-loop system that SystemBuild processed is available as SuperBlock `_Analysis_System`; an edited version of the [Figure 12-2](#) system is shown in [Figure 12-4](#).

Figure 12-4 **Open-loop Analysis Model**

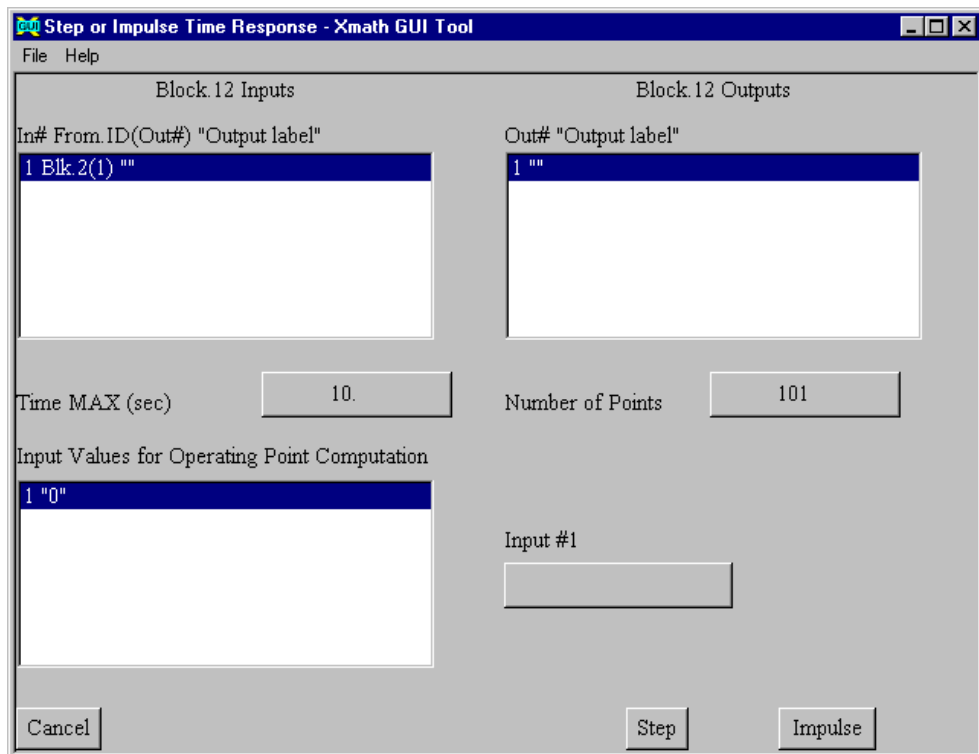


Note that in [Figure 12-4](#), the original loop is broken at the input to the plant and replaced by an external input; therefore `_Analysis_System` is effectively the plant as we wanted to see it.

12.4 Time Response

As soon as one or two blocks have been selected as the input and output blocks for the open-loop system, the Time Response menu item is enabled. If you select `Tools`→`Time Response`, the Step or Impulse Time Response dialog appears (see [Figure 12-5](#)).

Figure 12-5 Time Response Dialog





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 is used for both the input and output block.

The inputs and outputs areas of the dialog operate the same as for *Open-Loop Frequency Response*.



NOTE: 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 Input Values for Operating Point Computation 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 is performed.

You can select either step or impulse response, which results in the modified system to be linearized and the Xmath function for step or impulse to be invoked. A plot appears with the desired analysis result.

[Example 12-2](#) provides another opportunity to work with the **classical.cat** model.

Example 12-2 **Getting Step Response Plot of classical.cat**

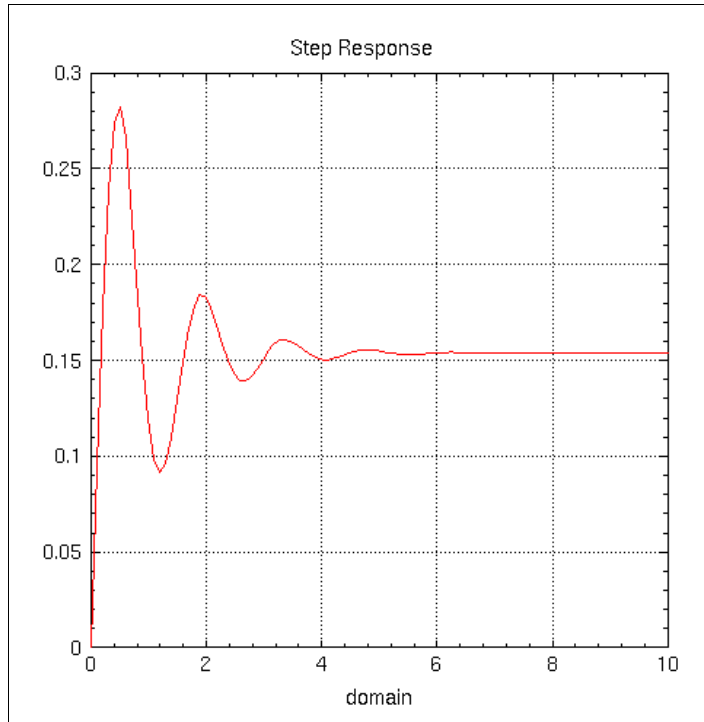
Consider the system shown in [Figure 12-2](#).

To perform a time response analysis on **classical.cat**:

1. Select the Plant block as both input and output.
2. Select Tools→Time Response.
3. Click Step.

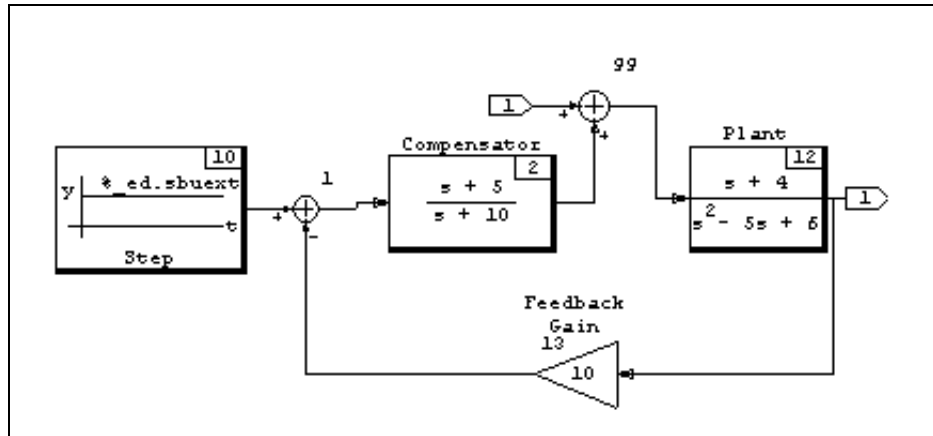
The resulting step response is shown in [Figure 12-6](#).

Figure 12-6 **Step Response Plot**



4. Close the plot window, and upload the model to the Catalog Browser.
The open-loop system that SystemBuild processed is available as SuperBlock _Analysis_System; an edited version of the [Figure 12-2](#) system is shown in [Figure 12-7](#).

Figure 12-7 Time Response Analysis Model



12.5 Point-to-Point Frequency Response

A basic SISO feedback loop typically includes three exogenous signals (see [DoFrTa92]). Figure 12-2 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 12-8. (This diagram is for illustrative purposes only.)

Example 12-3 Loading classical 2 Into a SuperBlock Editor

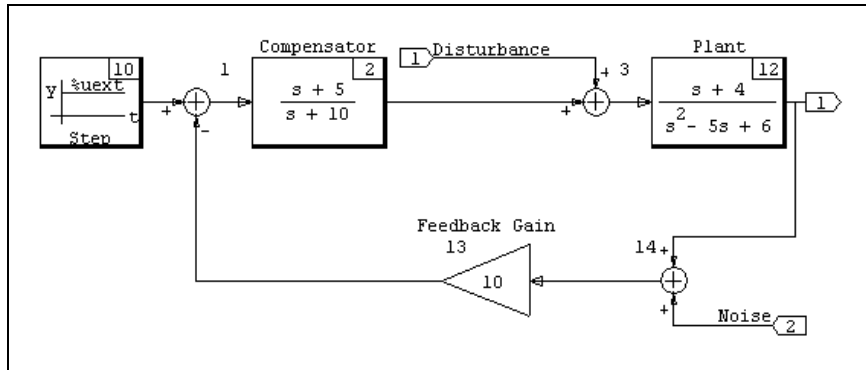
To load classical 2 into a SuperBlock Editor:

1. If you do not still have `classical.cat` loaded in your Catalog Browser, copy this file to your local directory:

```
copyfile "$SYSBLD/examples/classical_example/classical.cat"
```

2. Load the file into SystemBuild, and then display classical 2 in a SuperBlock Editor (see Figure 12-8).

Figure 12-8 **Open-Loop classical 2 System Before Editing**



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 12-8). To illustrate how this can be done, consider the system in Figure 12-2 again. Suppose that the disturbance signal to the plant is not already built into the model, as is the case in Figure 12-2, and you are interested in the transfer function from the disturbance signal at the plant input to the output of the feedback loop. See Example 12-4.

Example 12-4 **Adding a Disturbance Signal to classical 1**

To add a disturbance signal to classical 1:

1. Load classical 1 into a SuperBlock Editor.
2. 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.

3. 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.

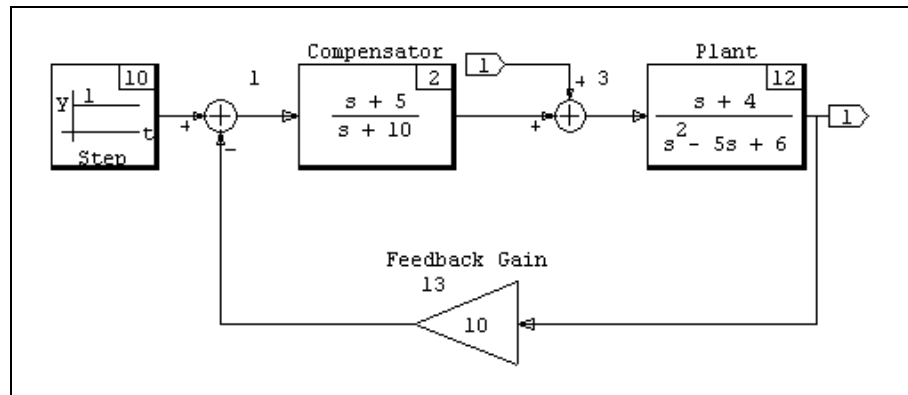
4. Select Tools→Point to Point Frequency Response.

The Point-To-Point Frequency Response dialog 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 12-9](#). This SuperBlock is named *classical 4*, and is available in the file you loaded earlier, *classical.cat*.

Figure 12-9 **Closed-loop Example for Analysis**

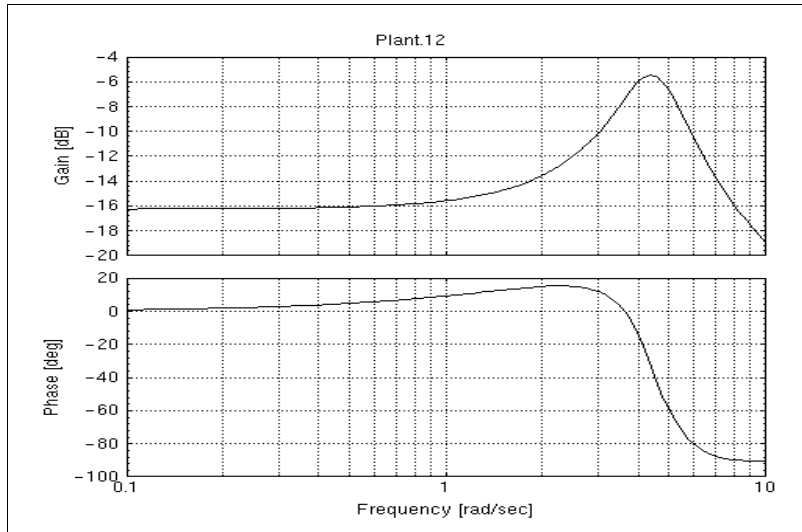


The fields on the dialog and the three types of frequency analyses operate the same as was explained in the previous section for *Open-Loop Frequency Response*.

5. Click Bode.

The plot shown in [Figure 12-10](#) appears.

Figure 12-10 **Bode Plot of the Point-to-Point Example**

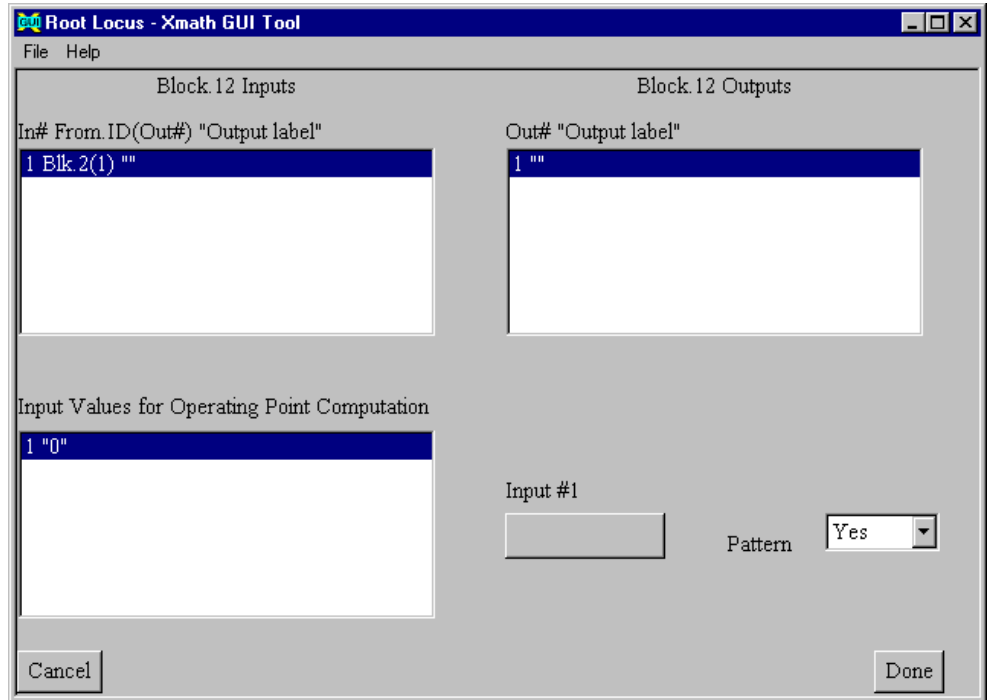


12.6 Root Locus

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 you select one or two blocks 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 appears (see [Figure 12-11](#)).

Figure 12-11 **Root Locus Dialog**



NOTE: The block considered as the input block is the first block selected, and the second block is the output block. If you select only one block, that block is used for both the input and output block.

The inputs area of the dialog lists the input channels of the selected input block, and the outputs area lists the output channels of the corresponding selected output block. You can select one channel each for the open-loop system input and output. This defines 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 are drawn on the root locus plot.



NOTE: The `root locus()` function invoked in this way is the standard Xmath interactive `root locus()` function; you might need to rearrange the screen windows to gain access to the Interactive Root Locus dialog, from which you can change the interactive gain value. Refer to online Help or the *Control Design Module User's Guide* for further details.

In the following sections, we provide several examples using this tool.

12.6.1 Application to a Linear System

[Example 12-5](#) uses the system in [Figure 12-2](#) again.

Example 12-5 **Creating a Root Locus of classical 1**

To create a root locus plot of classical 1:

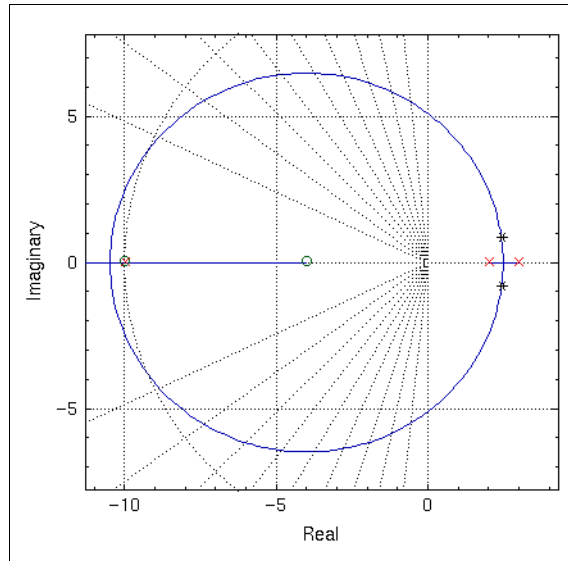
1. Load classical 1 into a SuperBlock Editor.
2. Select the Plant block.
3. Select Tools→Root Locus.

The Root Locus dialog appears (see [Figure 12-11](#)).

4. Accept the defaults, and click Done.

The system shown in [Figure 12-4](#) is created and linearized. The resulting root locus plot shown in [Figure 12-12](#) appears, along with the Interactive Root Locus dialog.

Figure 12-12 **Root Locus Plot**



5. Make some changes in the Interactive Root Locus dialog, and observe how they affect your plot.



NOTE: If your model has other dynamics that are not in the signal path of interest, these dynamics 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.

12.6.2 Application to a Multirate, Nonlinear System

To illustrate some of the hidden steps that are executed for an analysis, let us consider the multirate, nonlinear system in [Example 12-6](#).

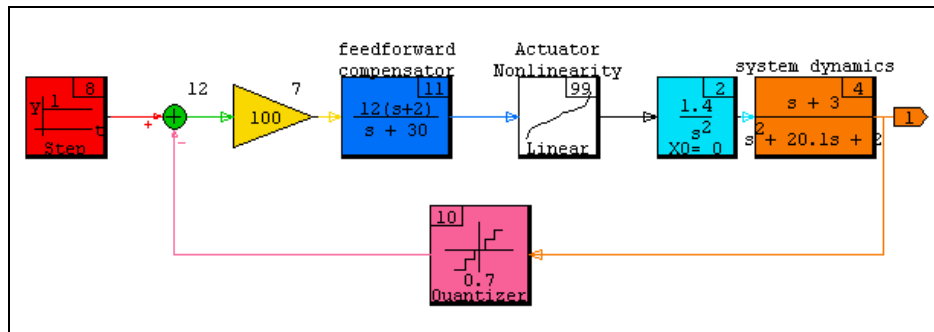
Example 12-6 Creating a Root Locus of a Multirate, Nonlinear System

To create a root locus of a multirate, nonlinear system:

1. Copy the model to your local directory by typing the following command in the Xmath command area:

```
copyfile "$SYSBLD/examples/classical_example/mws_demo.cat"
```

2. Load the file in SystemBuild, and open BUILT_MODEL in a SuperBlock Editor.



3. Select the feedforward compensator; then select Edit→Make SuperBlock. Upload the model to the Catalog Browser.
4. Go to the Catalog Browser, and select the new SuperBlock (_makesb). Select Tools→Transform.

The Transform SuperBlock dialog comes on view.

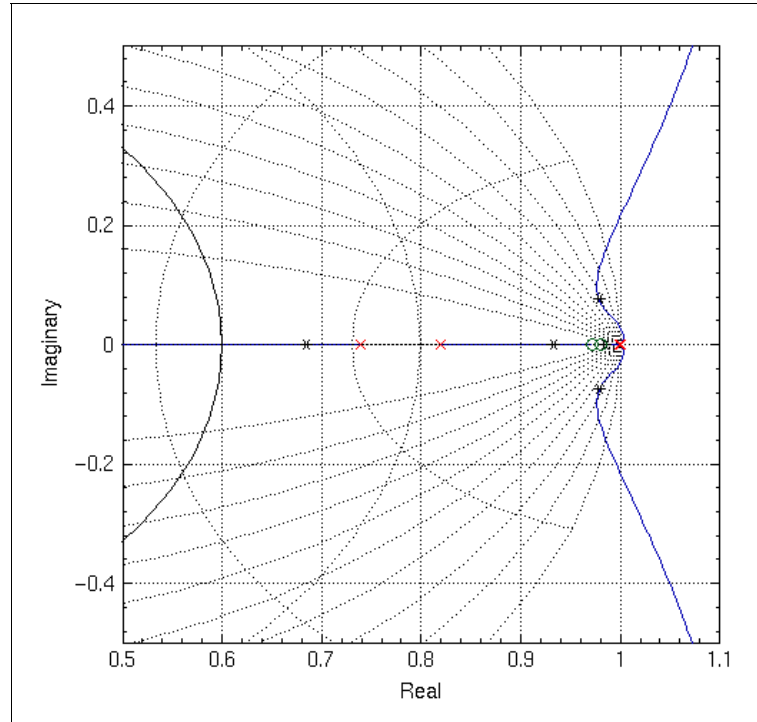
5. In the Transform SuperBlock dialog, make the type discrete, and specify a Sample Period of 0.01. Select OK.
6. In the editor, select the Gain block (ID 7) as input; then Ctrl-click to select the system dynamics block (ID 4) as output.
7. Select Tools→Root Locus.
The Root Locus dialog comes on view.
8. Click Done.

The Interactive Root Locus dialog, as well as the root locus plot, comes on view.

9. In the Interactive Root Locus dialog, change X Min to .5, X Max to 1.1, Y min to -.5, Y Max to .5, and the Feedback Loop Gain to 0.7; remember to press Return (or Enter) after entering data in each field. Click Recompute.

The root locus obtained, which is shown in [Figure 12-13](#), 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 [10.5 Multirate Linearization](#) on [p.230](#) for more on multirate linearization.

Figure 12-13 **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 menu forms, and by the state initial conditions. See [13.2 Operating Points](#) on [p.280](#)

for further explanation of operating points. The effect of the operating point on a linear analysis is illustrated in [Example 12-7](#).

Example 12-7 **Showing the Effect of the Operating Point on Linear Analysis**

To show the effect of the operating point on linear analysis:

1. From the Xmath Commands window, load the original **mws_demo.cat** model:

```
load "$SYSBLD/examples/mws_example/mws_demo.cat"
```

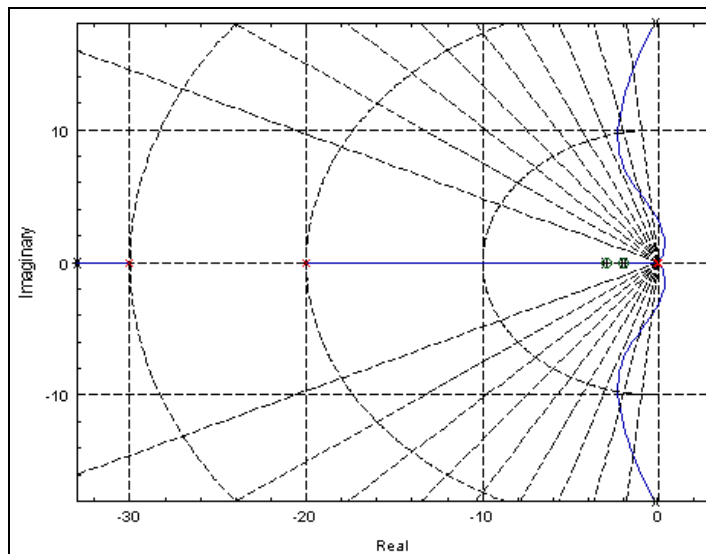
2. Open BUILT_MODEL in a SuperBlock Editor.
3. Select the Gain block (ID 7) as input; then Ctrl-click to select the system dynamics block (ID 4) as output.
4. Select Tools→Root Locus.

The Root Locus dialog comes on view.

5. Click Done.

The Interactive Root Locus dialog, as well as the root locus plot, comes on view.

6. In the Interactive Root Locus dialog, change the root locations interactively. Take note of where the roots are when the Feedback Loop Gain is 100.



7. Make a SuperBlock out of the five blocks connected between the Gain block and system dynamics (IDs 7, 11, 99, 2, and 4). Select File→Update.

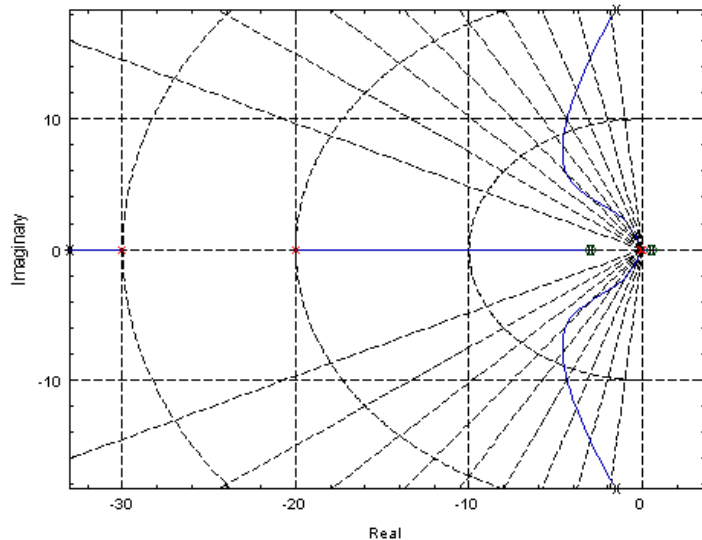
The new SuperBlock's name is the default name, `_makesb`.

8. To rename `_makesb`, go to the Catalog Browser, expand the SuperBlocks hierarchy in the Catalog view, and select the `_makesb` SuperBlock; select Edit→Rename, and name the new SuperBlock **newsb**.



NOTE: If you do not rename the SuperBlock, root locus fails.

9. Open `newsb` in a SuperBlock Editor.
10. In the editor, do a root locus analysis with the same input and output blocks (IDs 7 and 4), and launch the Root Locus tool. Note that this time there is a field to input values for an operating point computation. Click Done.
11. In the Interactive Root Locus dialog, set the Feedback Loop Gain is 100, and compare the values with those you noted in [Step 6](#).



Note that now when the Feedback Loop 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 the Gain block 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, 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.

12.7 Parameter Root Locus

The functionality of the Parameter Root Locus option differs from the Root Locus option. When you define a SISO system, $G(s)$, the Root Locus option automatically creates a closed-loop system as described in [12.6 Root Locus](#). 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 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.

[Example 12-8](#) provides an example of parameter root locus applied to a nonlinear system.

Example 12-8 Using Parameter Root Locus on a Nonlinear System

To use the Parameter Root Locus tool with example1:

1. Copy the model to your local directory by typing the following command in the Xmath command area:

```
copyfile "$SYSBLD/examples/classical_example/pr1.cat"
```

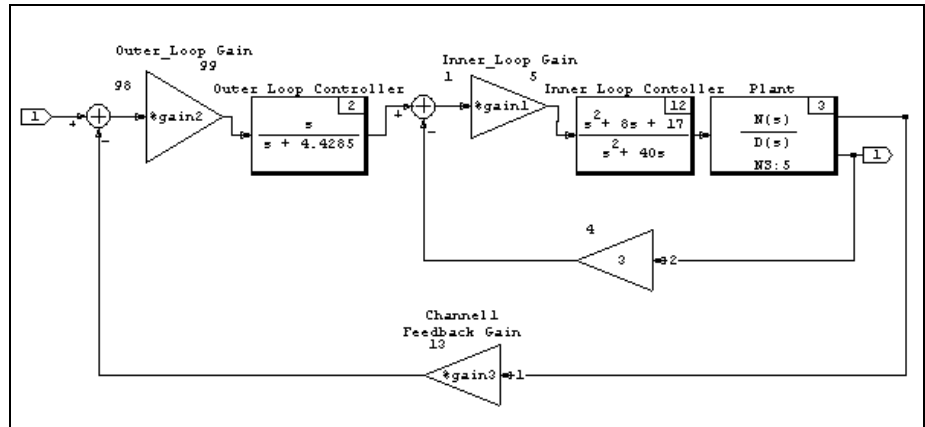
2. Load the file in SystemBuild, and open example1 in a SuperBlock Editor.

The nonlinear system is shown in [Figure 12-14](#), where the gains for Inner_Loop Gain, Outer_Loop Gain, and Channel 1 Feedback Gain are parameterized.



NOTE: The unorthodox block placement is to enable you to read the text on these blocks in this document.

Figure 12-14 example1

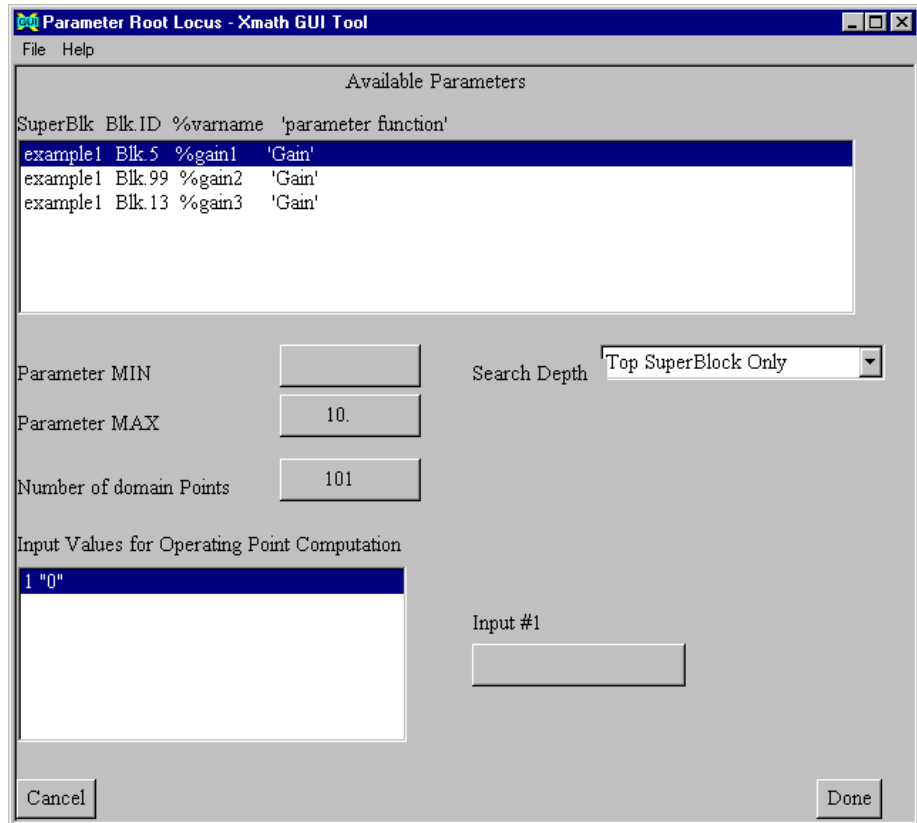


3. Select Tools→Parameter Root Locus.

The Parameter Root Locus option is enabled with no blocks selected.

The Parameter Root Locus dialog shown in [Figure 12-15](#) appears. It lists all scalar %Variables referenced in any block residing in the currently displayed SuperBlock and the hierarchy of SuperBlocks under it. You can select one %Variable, as well as a range for varying the parameter and the number of points.

Figure 12-15 Parameter Root Locus Dialog

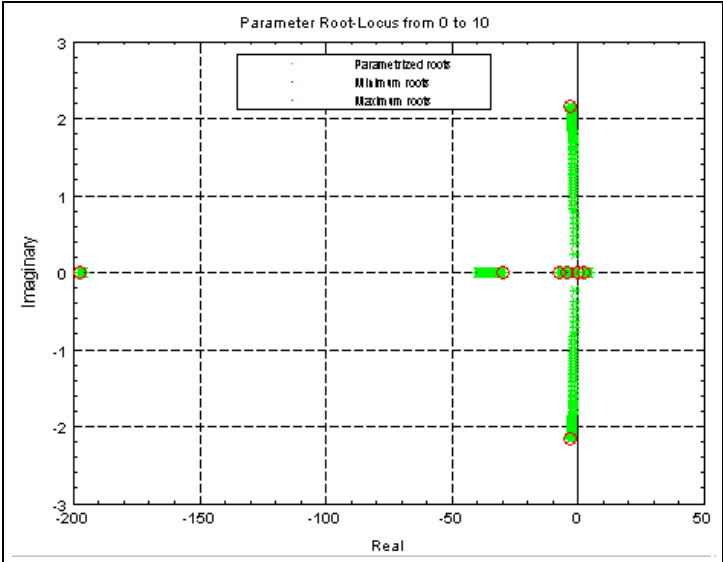


4. Click Done.

The root locus is computed by varying the %Variable from MIN to MAX with a stepsize 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 12-16 shows the results.

Figure 12-16 Parameter Root Locus Plot of the example1 Diagram



13

Advanced Simulation

In this chapter, we discuss a number of advanced simulation topics:

- *Explicit vs. Implicit Models*
- *Operating Points*
- *Inserting Initial Conditions*
- *Matrix Blocks in the Simulator*
- *Sim Integration Algorithms*
- *State Events*

13.1 Explicit vs. Implicit Models

Continuous simulation models can be implicit or explicit:

- A model is *explicit* when the state derivative can be written as an explicit function of the state.
The majority of SystemBuild blocks are explicit.
- A model is *implicit* if it contains implicit blocks and/or algebraic loops.
You can find implicit blocks on the Implicit palette.

13.1.1 Explicit Models

Explicit models are defined by an underlying ordinary differential equation (ODE) where the state derivative is explicitly computed from the state:

$$(\dot{x}(t) = f_e(x(t), t)) \quad (13-1)$$

Note that the external input is not included as one of the arguments on the right side of the equation. 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.

13.1.2 Implicit Models

Implicit models are defined by a residual (which is a function of time), the state, and the state derivatives:

$$\delta(t) = f_i(\dot{x}(t), x(t), t) \quad (13-2)$$

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) because 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 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.

If a model contains algebraic loops but no implicit blocks, it can be simulated using both implicit and explicit integration algorithms. The analyzer issues 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.

Constraints

You can use constraints to prevent the solution from drifting away from any predefined manifold known for the solution. You can impose constraints with implicit UCBs, constraint blocks and/or algebraic loops. (Example 13-6, p.299 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 over-determined 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 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. If auxiliary constraints are used, the δ vector is partitioned into two segments, which we refer to as δ_r (required constraints) and δ_a (auxiliary constraints).

Simulation State

The simulation state is described by implicit states, explicit states, and implicit outputs:

$$x = \begin{bmatrix} x_i \\ x_e \\ y_i \end{bmatrix} \quad (13-3)$$

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 (see [Implicit Outputs](#)). To save the state of the simulation, you must also save the implicit outputs.

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. The methodology 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 is to be done over the states or the state derivatives. Both the ImplicitUserCode block and the ImplicitVariable block have a combo box where you can set this search mode. If the diagram contains no state derivatives, the search is done over the states, and vice versa. If you want to bypass the operating point computation, specify `initmode = 4`. For more details, see [14.3.5 Operating Points](#) on p.344.

Examples

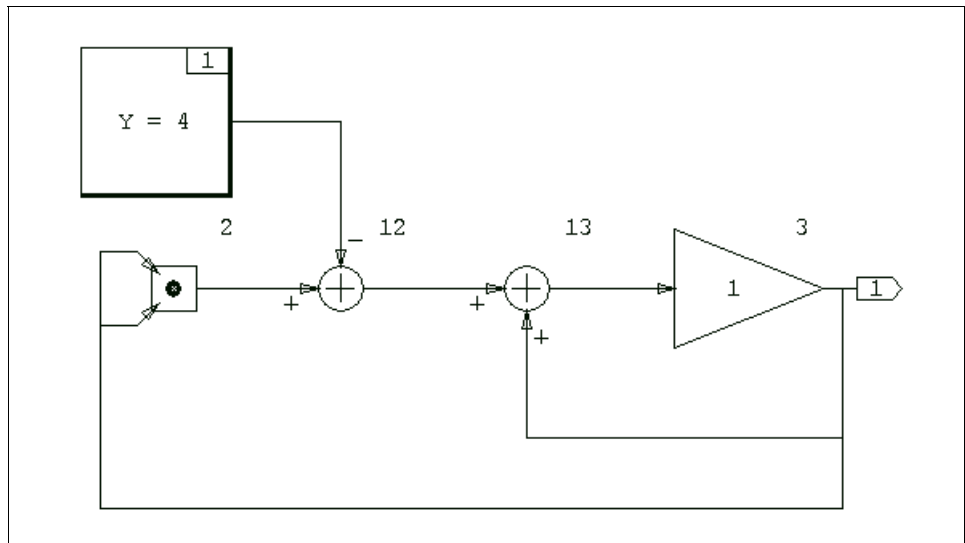
We illustrate the effect of implicit outputs and give examples of some of the available implicit blocks in this section.

Example 13-1 **Algebraic Loop**

A model containing an algebraic loop is shown in [Figure 13-1](#). The effect of the feedback loop containing the gain block is twofold:

- An implicit output is created after the summation block.
- The first input to the summer evaluates to zero.

Figure 13-1 **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), produced the desired result.



NOTE: Feeding the output of the Gain block into the DotProduct block avoids singularity of the Jacobian, making it possible for ODASSL to solve the equation without problems.

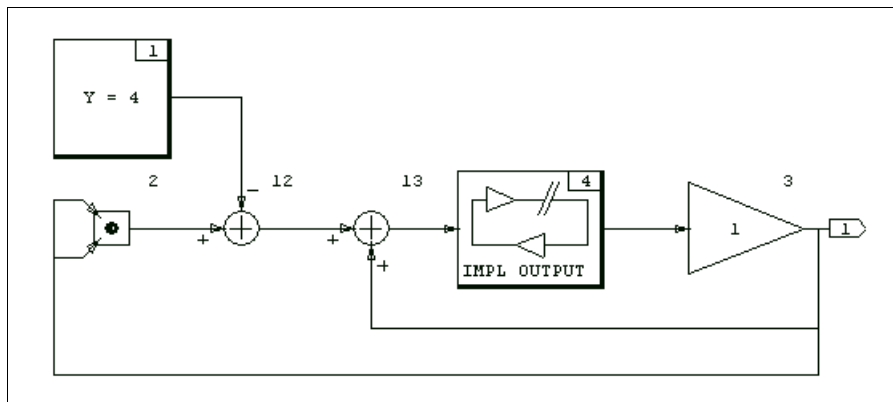
Disadvantages of solving the equation using this model are:

- 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()** function. 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.

Example 13-2 **The Implicit Output Block**

In this example, the ImplicitOutput 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 13-2](#).

Figure 13-2 **Solving $x^{**2} = 4$ Using an ImplicitOutput Block**



Because the model now contains an implicit block, it can only be simulated with implicit integration algorithms.

Example 13-3 **The Implicit Variable and Constraint Blocks**

The most appropriate way of solving $x^{**2} = 4$ is to use a combination of an ImplicitVariable block and a Constraint block. The ImplicitVariable block has an output vector consisting of two segments:

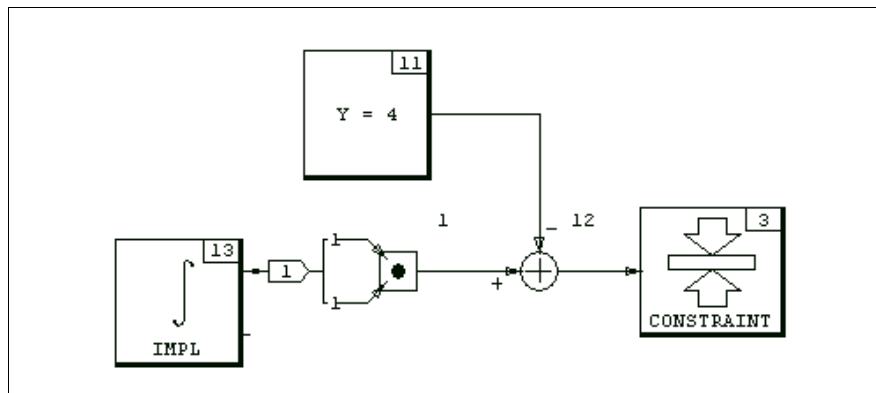
- The implicit state vector
- The implicit state derivative vector

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 for the derivative cannot be zero because this leads 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.

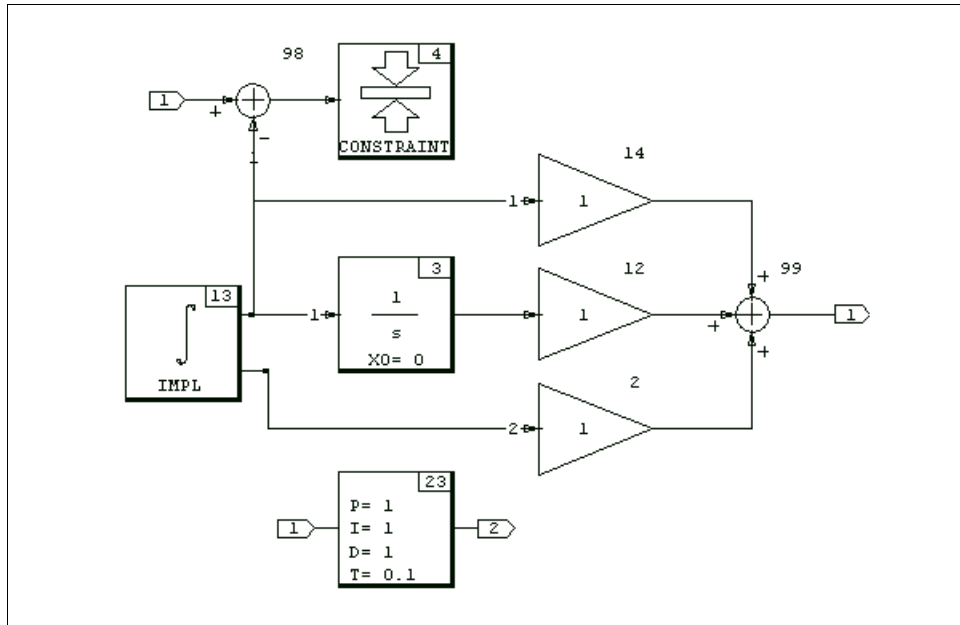
Figure 13-3 Solving $x^2 = 4$ Using ImplicitVariable and ImplicitConstraint Blocks



Example 13-4 **An Exact PID Controller**

A more useful example is the use of the ImplicitVariable and ImplicitConstraint blocks to simulate an exact PID controller (see [Figure 13-4](#)).

Figure 13-4 **An Exact PID Controller**

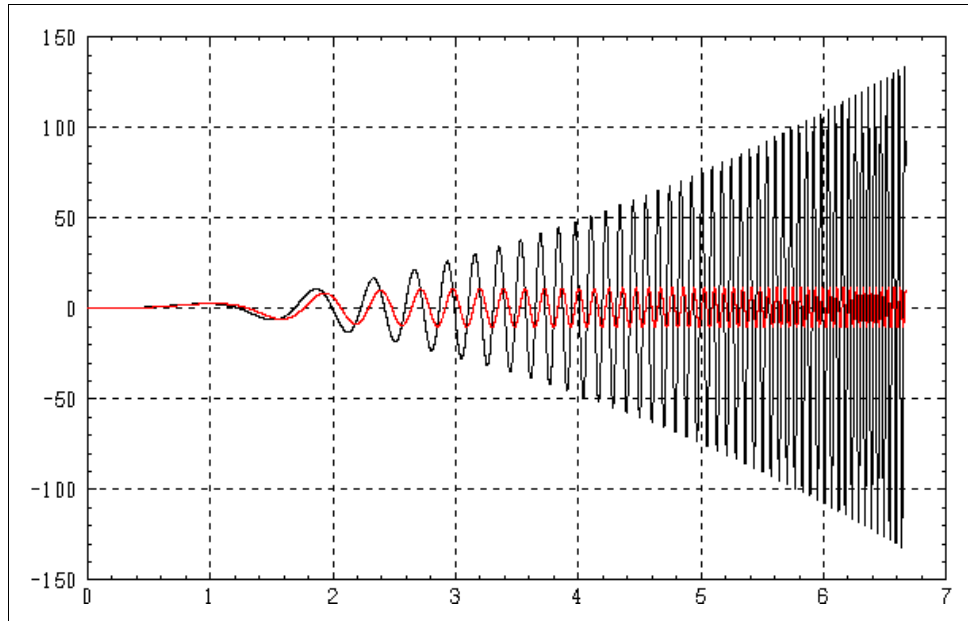


For the state derivative output of the ImplicitVariable block to represent the derivative of the input, the constraint is imposed so that its state output is equal to the external input. The commands used to simulate this system and plot the output are:

```
t = [0:10000]'/1500;  
[ ,y] = sim("imppid", t, sin(t.^3), {ialg=6});  
plot(t, y)
```

[Figure 13-5](#) compares this output with the output of the PID Controller block on the dynamic palette using the default parameters.

Figure 13-5 Comparison of Exact vs. Approximate PID Controllers.



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.

Example 13-5 **The Inverted Pendulum**

For this example we refer forward to [16. Fixed-Point Arithmetic](#), where an inverted pendulum example is used to illustrate `ImplicitUserCode` block capabilities. This example can also be simulated using `ImplicitVariable` and `ImplicitConstraint` blocks, so that a UCB is not required. An interesting aspect of this example is that it can be simulated under a variety of conditions, among which are the cases of one or two auxiliary constraints. You can find the implicit block version of the inverted pendulum example in `SYSBLD/examples/pend_imp_no_iucb`. Load the catalog and run the MathScript found in that directory.

13.2 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 for all states, state derivatives, block outputs, and implicit algebraic loop variables. The calculation of the operating point is influenced by the values of the absolute and relative tolerances (see [Absolute and Relative Tolerances](#)).

This section discusses the methods for finding the operating point.

13.2.1 Continuous Subsystem

The operating point for a continuous subsystem is found by evaluating all continuous block outputs based upon the system's initial states and inputs. If the system contains algebraic loops or implicit states (via the `ImplicitUserCode` 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 do not occur.

If algebraic loop initial conditions are specified with the **yimp0** keyword, the operating point computation for the algebraic loops utilizes the **yimp0** values as initial conditions for the Newton-Raphson solver.¹

13.2.2 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 $-\sqrt{\epsilon}$.

1. Note that operating point computations for implicit states, as well as algebraic loops, are performed only for continuous subsystems.

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 to TRUE). The computations are based on the initial states and inputs.

At **initmode = 3**, the initialization is the same as **initmode = 2**, except that there is no sample and hold between subsystems.

Using **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, you can avoid the problem by setting **initmode = 0**, which may result in a different initial subsystem execution order; 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: Avoid algebraic loops in discrete systems.

If a discrete subsystem contains algebraic loops, a computational delay occurs because some block outputs are used in computations before they are calculated during each time interval. This computational delay represents a pseudo-dynamic, which may produce non-steady transients in the simulation results.

13.3 Inserting Initial Conditions

The **sim()**, **simout()**, and **lin()** functions allow you to set initial conditions 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.

You can supply **x0**, **xd0**, and **yimp0** values as optional keywords to the **sim()**, **simout()**, and **lin()** functions, which changes the operating point. These keywords insert initial dynamic state values (**x0**), initial state derivatives for ImplicitUserCode 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. You can change the catalog values by editing the block diagram only.

For the syntax and method of operation of the **sim()**, **simout()**, and **lin()** initial condition features, see 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; upon resumption of a simulation, the Padé states revert to the values they had at the end of the last simulation.

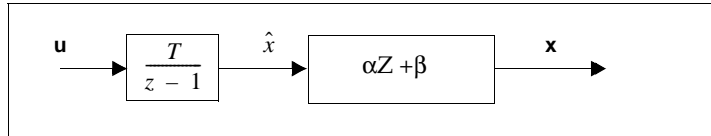
The simulator's **x0** initial condition option does not allow initialization of memory states, which include state transition diagram (STD) states and DataStore initial conditions. These initial conditions are values stored in the SystemBuild catalog except when a **sim()** is resumed, in which case 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.

SystemBuild allows you to specify initial *outputs* for the transfer function blocks, NumDen, PoleZero, and ComplexPoleZero via their block dialogs. Using the internal state-space representations of these blocks, SystemBuild assigns appropriate initial conditions in these blocks to produce the desired initial outputs. However, this mapping is not necessarily unique and is not visible to the user; 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 13-6](#), if the initial condition x_0 is defined in the

block parameter dialog, it is applied at the output of the integrator. By contrast, the `sim() x0` keyword option initializes the integrator's state, \hat{x} .

Figure 13-6 **1/s Substitution in Discrete Integrator**



13.4 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 blocks are extremely simple and introduce no error. The ScalarGain block uses sequential scalar multiplication; the MatrixMultiply, LeftMultiply, and RightMultiply blocks are implemented using repeated dot products. The multiplications might introduce some error as a consequence of the floating-point or fixed-point multiplications being carried out (see [16.1.4 Multiplication](#) on p.405).

This leaves us with the MatrixInverse, MatLeftDivide, and MatRightDivide blocks that compute $1/A$ or solve $AX=B$ or $XA=B$ for the unknown X . These blocks involve complicated algorithms that might produce a large relative error, depending upon the input. In simulation, these blocks are implemented using the Gaussian elimination algorithm. This algorithm's behavior is well-known. We can make definitive statements about how much error the solution X or the inverse $1/A$ contains, thanks to the reciprocal conditioning number, which is generated by the standard LINPACK implementation of Gaussian elimination. The simulator halts in a MatrixInverse, MatLeftDivide, or MatRightDivide block if the reciprocal condition number goes exactly to zero; this indicates that it is impossible to invert or divide by the matrix in a meaningful manner.

13.5 Sim Integration Algorithms

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

- Continuous
- Discrete
- Hybrid (a combination of continuous and discrete subsystems)

Given user-defined initial conditions and input vector, the procedure of simulating the SystemBuild model or obtaining a sequence of solutions to the system equations 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 Wind River 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	Over-determined differential-algebraic equation

Integration methods can be divided into four classes: one-step, multi fixed-step, variable-step, and stiff system solvers.

You can find a number of references for integration algorithms in the [Bibliography](#).

13.5.1 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 (**GEARS**)

The default integration algorithm is 5 (Variable-step Kutta-Merson). You can set the algorithm globally using the command:

```
SETSBDEFAULT,{ialg=alnumber}
```

where *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, use the Simulation dialog by selecting Tools→Simulate from a SystemBuild Editor, or supply the keyword in the `sim()` function call:

```
y = sim(model,t,u,{ialg = alnumber})
```

13.5.2 Overview of the Algorithms

SystemBuild currently provides nine different integration algorithms. For numerically well-conditioned and non-stiff problems, you can expect almost any of the algorithms to yield reliable answers, although the execution times 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 replaced by $\frac{\Delta x}{\Delta t}$. Literally implementing this procedure yields Euler's method.

Euler Integration Method

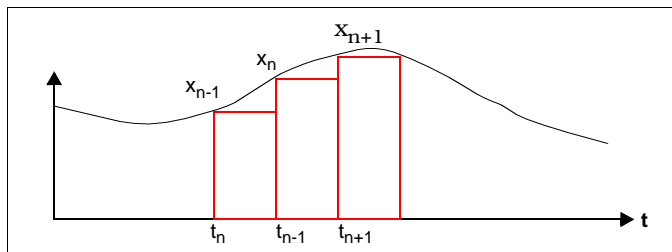
Euler integration is an explicit, first-order method with one function evaluation per step:

$$\begin{aligned}x_{n+1} &= x_n + hf(x_n, t_n) \\ \text{where } h &= t_{n+1} - t_n\end{aligned}\tag{13-4}$$

Note that the stepsize, h , is taken from the time vector that you supply.

This method is equivalent to approximating the area under the solution curve with a series of rectangles, as shown in [Figure 13-7](#).

Figure 13-7 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 stepsize increases the effect of round-off errors and compromises the computational accuracy and speed.

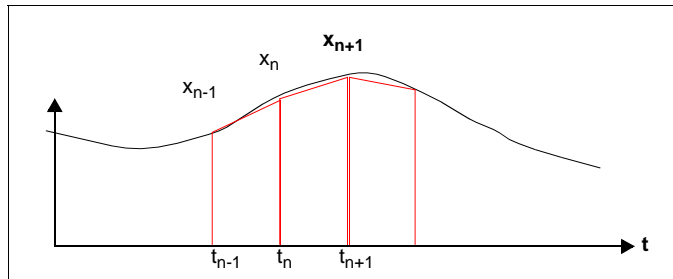
Second Order Runge-Kutta (Modified Euler) Method

This method requires two function evaluations per step, using the following equations:

$$\begin{aligned}k_1 &= hf(x_n, t_n) \\k_2 &= hf(x_n + k_1, t_n + h) \\x_{n+1} &= x_n + \frac{k_1 + k_2}{2}\end{aligned}\tag{13-5}$$

The second-order Runge-Kutta method improves on the accuracy of the Euler method by fitting trapezoids under the solution curve instead of rectangles, as seen in [Figure 13-8](#). The local truncation error is $O(h^3)$.

Figure 13-8 **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:

$$\begin{aligned}k_1 &= hf(x_n, t_n) \\k_2 &= hf\left(x_n + \frac{k_1}{2}, t_n + \frac{h}{2}\right) \\k_3 &= hf\left(x_n + \frac{k_2}{2}, t_n + \frac{h}{2}\right) \\k_4 &= hf(x_n + k_3, t_n + h) \\x_{n+1} &= x_n + \frac{k_1 + 2k_2 + k_4}{6}\end{aligned}\tag{13-6}$$

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:

$$\begin{aligned}
 k_1 &= hf(x_n, t_n) \\
 k_2 &= hf\left(x_n + \frac{k_1}{3}, t_n + \frac{h}{3}\right) \\
 k_3 &= hf\left(x_n + \frac{k_2 + k_1}{6}, t_n + \frac{h}{3}\right) \\
 k_4 &= hf\left(x_n + \frac{3k_3 + k_1}{6}, t_n + \frac{h}{2}\right) \\
 \tilde{x} &= x_n + \frac{4k_4 + 3k_3 + k_1}{2} \\
 k_5 &= hf(\tilde{x}, t_n + h) \\
 x_{n+1} &= x_n + \frac{k_5 + 4k_4 + k_1}{6}
 \end{aligned} \tag{13-7}$$

Variable-Step Kutta-Merson Method

As shown in reference [F62] in the *Bibliography*, the difference between terms \tilde{x} and x_{n+1} in the fixed Kutta-Merson method shown above 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:

$$\text{ERR} = \frac{\tilde{x} - x_{n+1}}{5} \tag{13-8}$$

The maximum stepsize is limited by the time increment $t_{n+1} - t_n$ or the value of the keyword **dtmax**. Because of its accuracy, reliability, moderately efficient speed, and its successful performance in a wide range of problems, the variable-step Kutta-Merson method has been chosen as the default integration algorithm in SystemBuild.

Stiff System Solver

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 are not 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 `ImplicitUserCode` 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; additionally, 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 succeeds in most instances. If an operating point cannot be found, however, this means that one of the following is true:

- Either the problem or the given initial conditions are ill-posed (physically unrealizable)
- The true solution cannot be reached from the current estimate of its value.

DASSL presents 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, you should inspect the SystemBuild model and the initial conditions 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:

$$\begin{aligned} g(\dot{x}, x, t) &= 0; \\ \dot{x}(t_0) &= \dot{x}_0 \\ x(t_0) &= x_0 \end{aligned} \tag{13-9}$$

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:

$$J = \frac{\partial g}{\partial x} + c \frac{\partial g}{\partial \dot{x}} \tag{13-10}$$

where the constant c is determined by the backward differentiation formula. The Newton-Raphson iteration equation

$$x_n^{k+1} = x_n^k - J(x_n^k)^{-1} g(x_n^k, x_n^k, u_n) \tag{13-11}$$

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 fails.

The algorithm in SystemBuild is based on the DASSL stiff system solver developed by Linda Petzold at Sandia National Laboratories. [Selecting an Integration Algorithm](#) on p.179 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](#) on p.305 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 because 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:

$$\dot{x}_{n+1} = x_n + \frac{h(55f_n + 59f_{n-1} + 37f_{n-2} - 9f_{n-3})}{24} \quad (13-12)$$

f is the differential equation evaluated at step n :

$$f_n = \dot{x}_n = f(x_n, t_n) \quad (13-13)$$

The Adams-Moulton Corrector step is computed as:

$$x_{n+1} = x_n + \frac{h(9\tilde{f}_{n+1} + 19f_n + 5f_{n-1} + f_{n-2})}{24} \quad (13-14)$$

In the above equation \tilde{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 [13.5.4 Computing the Maximum Integration Stepsize in Variable-Step Integration Algorithms](#).

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:

$$\begin{aligned} \dot{x} = f(x, u) &\approx f(x(t_k), u(t_k)) + \left. \frac{\partial f}{\partial x} \right|_{x(t_k), u(t_k)} (x - x(t_k)) \\ &+ \left. \frac{\partial f}{\partial u} \right|_{x(t_k), u(t_k)} (u - u(t_k)) + \text{h.o.t.} \end{aligned} \quad (13-15)$$

Defining:

$$\begin{aligned} A &= \left. \frac{\partial f}{\partial x} \right|_{x(t_k), u(t_k)} \\ B &= \left. \frac{\partial f}{\partial u} \right|_{x(t_k), u(t_k)} \\ x_{k+1} &= x(t_{k+1}) \\ x_k &= x(t_k) \\ h &= t_{k+1} - t_k \\ \Delta u(t) &= u(t) - u(t_k) \end{aligned}$$

we obtain:

$$\dot{x}_{k+1} = Ax_{k+1} + f(x_k, u_k) + B\Delta u(t_{k+1}) - Ax_k \quad (13-16)$$

and solving for x_{k+1} :

$$x_{k+1} = x_k + \int_0^h e^{A(h-\tau)} [f(x_k, u_k) + B\Delta u(\tau)] d\tau \quad (13-17)$$

The method is A-stable, with the same stability properties as trapezoidal integration.

To implement the algorithm, a secant approximation of Δu is used:

$$\Delta u(t) = \frac{u_{k+1} - u_k}{h} t \equiv m_k t \quad (13-18)$$

Substituting, we obtain:

$$x_{k+1} = x_k + \int_0^h e^{A(h-\tau)} f(x_k, u_k) d\tau + \int_0^h e^{A(h-\tau)} B m_k \tau d\tau \quad (13-19)$$

The final form of the algorithm is:

$$x_{k+1} = x_k + I_1 f(x_k, u_k) + I_2 m_k \quad (13-20)$$

where I_1 and I_2 are the two integrals above. These integrals are precomputed at the beginning of the simulation.

The QuickSim 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 be x_k . Assume that $x(t_k) = x_k$ and write expressions for $x(t_{k+1})$ and x_{k+1} :

$$\begin{aligned} x(t_{k+1}) &= x(t_k) + \dot{x}(t_k)h \\ &= x_k + f(x_k, u_k)h \end{aligned} \quad (13-21)$$

and

$$\begin{aligned} x_{k+1} &= x_k + \int_0^h e^{A(h-t)} [f(x_k, u_k) + Bm\tau] d\tau \\ &= x_k + \int_0^h \left[I + A(h-\tau) + \frac{A^2(h-\tau)^2}{2!} + \dots \right] [f(x_k, u_k) + Bm\tau] d\tau \\ &= x_k + f(x_k, u_k)h + \frac{1}{2}Ah^2 + h.o.t. \end{aligned} \quad (13-22)$$

Therefore:

$$\|x_k - x(t_{k+1})\| = O(h^2)$$

Over-determined Differential Algebraic System Solver

The over-determined differential algebraic system solver (ODASSL) 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. An ImplicitUserCode block (UCB) is provided to enable you to incorporate DAEs into SystemBuild. The Implicit UCB ([14.1.2 Implicit UCBs](#) on [p.325](#)) 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 over-determined DAE form offers the most convenient approach.

Consider the implicit differential equation (IDE):

$$f(\dot{x}, x, t) = 0 \quad (13-23)$$

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, ODASSL, or GEARS (ialg options 6, 9, and 10, respectively), the DAE must have an index of 0 or 1 because these algorithms are not designed to handle systems of index greater than 1. In particular, DASSL and ODASSL fail if the Jacobian is singular:

$$J = \frac{\partial f}{\partial x} + c \frac{\partial f}{\partial \dot{x}} \quad (13-24)$$

See [14.3.5 Operating Points](#) on [p.344](#) 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 UCB, the initial conditions $\dot{x}(t_0)$ and $x(t_0)$ must be consistent when a simulation is started. They must satisfy the following equation:

$$f(\dot{x}(t_0), x(t_0), u(t_0)) = 0 \quad (13-25)$$

Otherwise the algorithms might 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:

$$f_c(\dot{x}(t_0), x(t_0), u(t_0)) = 0 \quad (13-26)$$

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 [Example 13-6](#).

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.

Using ODAEs in Multibody System Dynamics

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:

$$\begin{aligned} \dot{p} &= v \\ M(p)v &= f(p, v, u) - \frac{\partial}{\partial p}g_p(p)^T\lambda \\ 0 &= g_p(p) \end{aligned} \tag{13-27}$$

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 (13-27) 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:

$$\begin{aligned} g_v(p, v) &= \dot{g}_p(p) = 0 \\ g_a(p, v, \dot{v}) &= \dot{g}_v(p, v) = 0 \end{aligned} \tag{13-28}$$

Possible Formulations for Holonomic Systems

Three types of formulations are possible with holonomic systems. The most reliable numerical results are usually obtained from the *Index 1 Formulation with Two Constraints* below.

Unconstrained DAE Formulations

Index 3 Formulation

$$\begin{aligned}0 &= \dot{p} - v \\0 &= M(p)\dot{v} - f(p, v, u) + \frac{\partial}{\partial p}g_p(p)^T\lambda \\0 &= g_p(p)\end{aligned}\tag{13-29}$$

There are $n_p+n_v+n_l$ states and the same number of equations.

13

Index 2 Formulation

Instead of $0=g_p(p)$, use the constraint $0=g_v(p,v)$.

Index 1 formulation:

Instead of $0 = g_p(p)$, use the constraint $0 = g_a(p, v, \dot{v})$

Index 2 Formulation with One Constraint

DAEs

$$\begin{aligned}0 &= \dot{p} - v \\0 &= M(p)\dot{v} - f(p, v, u) + \frac{\partial}{\partial p}g_p(p)^T\lambda \\0 &= g_v(p, v)\end{aligned}\tag{13-30}$$

Constraints

$$0 = g_p(p) \quad (13-31)$$

Index 1 Formulation with Two Constraints

DAEs

$$\begin{aligned} 0 &= \dot{p} - v \\ 0 &= M(p)\dot{v} - f(p, v, u) + \frac{\partial}{\partial p}g_p(p)^T\lambda \\ 0 &= g_a(p, v, \dot{v}) \end{aligned} \quad (13-32)$$

Constraints

$$\begin{aligned} 0 &= g_v(p, v) \\ 0 &= g_p(p) \end{aligned} \quad (13-33)$$

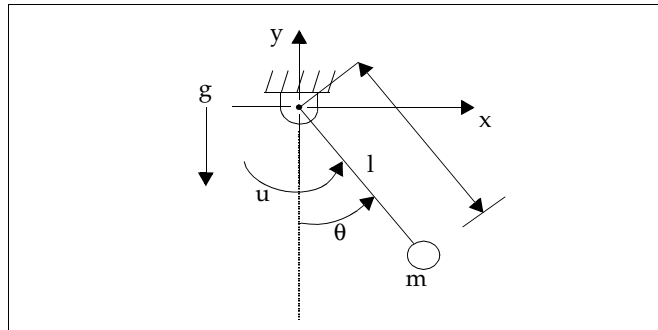
Example of a Holonomic System

The formulations in the previous section are demonstrated in [Example 13-6](#).

Example 13-6 Pendulum Example

A simple pendulum is illustrated in [Figure 13-9](#).

Figure 13-9 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 an ODE:

$$\ddot{\theta} = \frac{g}{l} \sin \theta + u \quad (13-34)$$

For purposes of illustration, the equations of motion are derived using the coordinates x and y . The pendulum oscillates on a circle of radius l ; therefore, the position constraint is:

$$g_p = \frac{x^2 + y^2 - l^2}{2} = 0 \quad (13-35)$$

where:

$$\begin{aligned} m\ddot{x} &= -\frac{y}{l^2} - x\lambda \\ m\ddot{y} &= -\frac{x}{l^2}u - mg - y\lambda \\ 0 &= \frac{1}{2}(x^2 + y^2 - l^2) \end{aligned} \quad (13-36)$$

The above set of equations constitutes an index-3 unconstrained formulation.

For index-2 and index-1 formulations, we use the velocity constraint,

$$g_v = \dot{g}_p = x\dot{x} + y\dot{y} = 0 \quad (13-37)$$

and the acceleration constraint,

$$g_a = \dot{g}_v = x\ddot{x} + \dot{x}^2 + y\ddot{y} + \dot{y}^2 = 0 \quad (13-38)$$

in place of the position constraint above.

Thus, an index-1, two-constraint formulation of this problem would be as follows:

$$\text{Let } \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \lambda \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \quad \begin{array}{l} \text{(External input was taken to be} \\ \text{zero in this example.)} \end{array} \quad (13-39)$$

Then:

$$\text{DAE} \quad \left\{ \begin{array}{l} x_1 - x_3 = 0 \\ x_2 - x_4 = 0 \\ m\dot{x}_3 + x_1x_5 = 0 \\ m\dot{x}_4 + x_2x_5 + mg = 0 \\ x_1\dot{x}_3 + x_3^2 + x_2\dot{x}_4 + x_4^2 = 0 \end{array} \right. \quad (13-40)$$

$$\text{Velocity level constraint:} \quad x_1x_3 + x_2x_4 = 0$$

$$\text{Position level constraint:} \quad \frac{1}{2}(x_1^2 + x_2^2 - l^2) = 0$$

This example is coded as an implicit UCB in the SystemBuild **examples** directory.

To run the pendulum example:

1. From the Xmath command area, copy the files from the examples directory to your current directory:

```
copyfile "$SYSBLD/examples/pendulum_imp/*"
```

2. In the Xmath command area type:

```
execute file = "pend_imp.ms"
```

This command loads and simulates 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 (see [BG73] in the [Bibliography](#)). 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 the Wind River 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 of the Newton corrector has been implemented to make the algorithm suitable for over-determined 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 below:

- (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 method 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 method; conversely, Gear's Jacobian is more accurate.

- ODASSL imposes any auxiliary constraints (that is, any equations that make the system over-determined) in an exact manner. Gear's method 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.

13.5.3 Absolute and Relative Tolerances

The absolute and relative tolerances are used in determining the operating point as well as the step size in several integration methods.

The convergence test used for computing the operating point is based on a error criterion similar to that used by DASSL and ODASSL (see *Stiff System Solver*).

The variable-step Kutta-Merson, stiff system solver, and variable-step Adams-Moulton integration methods all make use of user-definable absolute and relative tolerances to determine whether the order and/or the stepsize need 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 reltol keyword is specified in the sim() function.
ABSTOL	Absolute tolerance (default = $\sqrt{\epsilon}$, where ϵ is the machine epsilon). The abstol value is specified as a sim() 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 modifies 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()** function tolerance parameters **abstol** and **reltol**. These multiplication factors are stored in row vectors and have the default value of 1. They are used only by

variable-step integration algorithms and can help improve the speed and/or accuracy of the simulation results.

Variable-Step Kutta-Merson Method

The variable-step Kutta-Merson algorithm uses the following test:

$$ERR = \frac{\dot{x} - x_{n+1}}{5} \quad (13-41)$$

$$\text{If } \|ERR\| \leq RELTOL \times \|x\| \text{ or } \|ERR\| \leq ABSTOL$$

then the solution for x is good. (See the [Fixed-Step Kutta-Merson Method](#) on p.288 for the definition of \dot{x} .) Otherwise, the stepsize h is decreased, and the Kutta-Merson equations are recalculated.

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

$$v(i) = \frac{ERR(i)}{RELTOL \times |x(i)| + ABSTOL} \quad (i=1, \dots, N) \quad (13-42)$$

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:

$$\|X\|_2 = \sqrt{\sum_{i=1}^n v_i^2} \quad (13-43)$$

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 yields more accurate answers. The RMS norm is less accurate, but the algorithm usually executes faster. 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 in terms of computations to solve than other systems. In DASSL, the expense is more strongly dependent on the tolerance than it is with other algorithms.

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:

$$|ERR| < h_{n+1} TOL \quad (13-44)$$

where:

$$TOL = RELTOL \times \|x\| + ABSTOL \quad (13-45)$$

In particular, the local truncation error *ERR* is computed as:

$$ERR = |h_n(g_{k+1,1} - g_{k,1})\phi_{k+1}(n)| \quad (13-46)$$

The index *k* is the order of the method and *n* is the time step index. The 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:

$$r = \left(\frac{TOL}{2 \times ERR} \right)^{\frac{1}{k+1}} \quad (13-47)$$

13.5.4 Computing the Maximum Integration Step Size in Variable-Step Integration Algorithms

At the end of each integration step, the simulation scheduler decides the step size for the next step. It uses five factors to compute the maximum step size the variable-step integrators can take, although they may internally choose a smaller step size in order to satisfy error requirements. Fixed-step size integrators use the step size provided by the first four factors listed below.

1. The user-defined time vector defines the times when the output is posted. The step size 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, provides control over the spacing of posted outputs and thus over the integration step size. It may override factor 1 because the integration step size 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 step size.
4. The sampling of discrete subsystems might affect the length of an integration step. At a given time, the scheduler checks for any upcoming discrete events. The continuous integration step size 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 step size is reduced to the time instant of the state event.

13.5.5 Sample Simulation

In [Example 13-7](#), 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.

Example 13-7 **Spring-Mass System for Comparing Integration Algorithms**

For evaluating and comparing the speed and accuracy of the integration algorithms in SystemBuild, a spring-mass system was modeled with viscous damping proportional to the position multiplied by the velocity and with a nonlinear cubic spring:

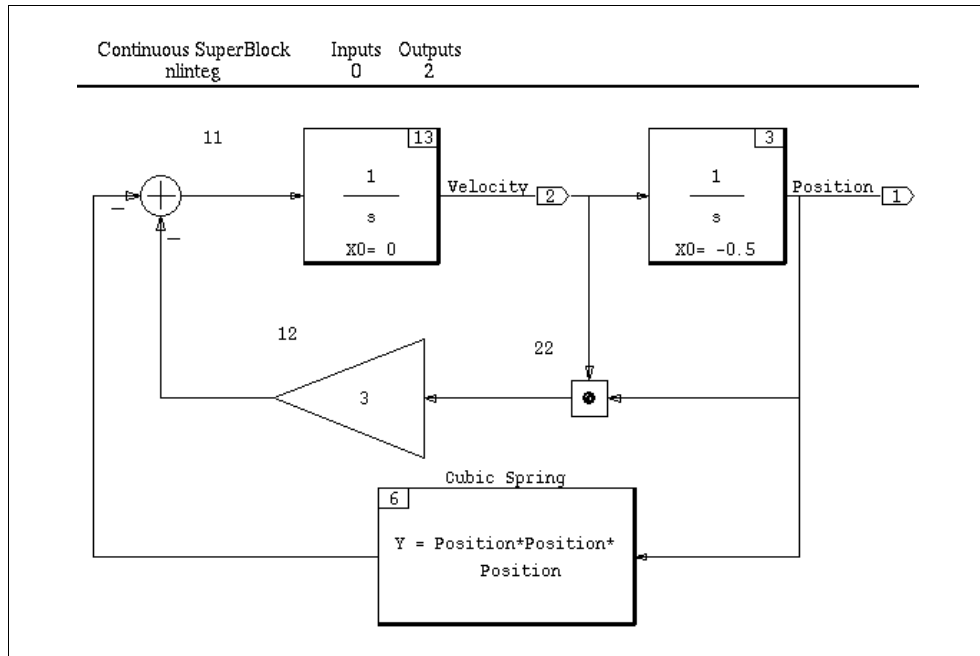
$$\ddot{x} + 3x\dot{x} + x^3 = 0; x_0 = -5; \dot{x}(t_0) = 0. \quad (13-48)$$

Reference [McL50] in the *Bibliography* gives the exact solution for the response of the above system as:

$$x(t) = 2 \left(\frac{t + \frac{1}{x_0}}{\left(t + \frac{1}{x_0}\right)^2 + \left(\frac{1}{x_0}\right)^2} \right) \quad (13-49)$$

The SystemBuild block diagram model for this system is shown in [Figure 13-10](#).

Figure 13-10 **Block Diagram of the Cubic Spring Model**

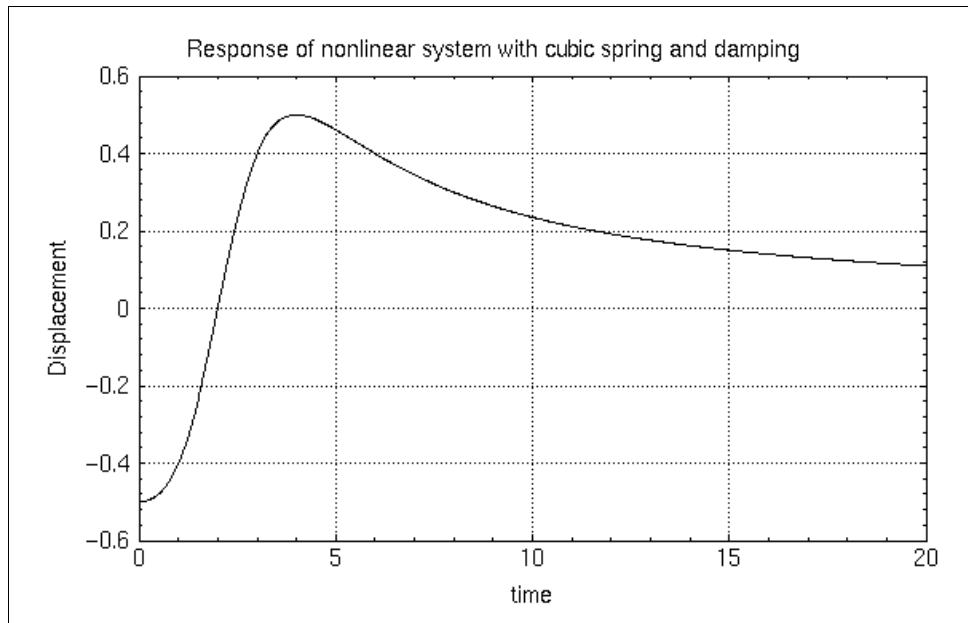


You can copy the data file for this model to your local directory with the following Xmath command:

```
copyfile "$SYSBLD/examples/ialgs/cubic_spring*"
```

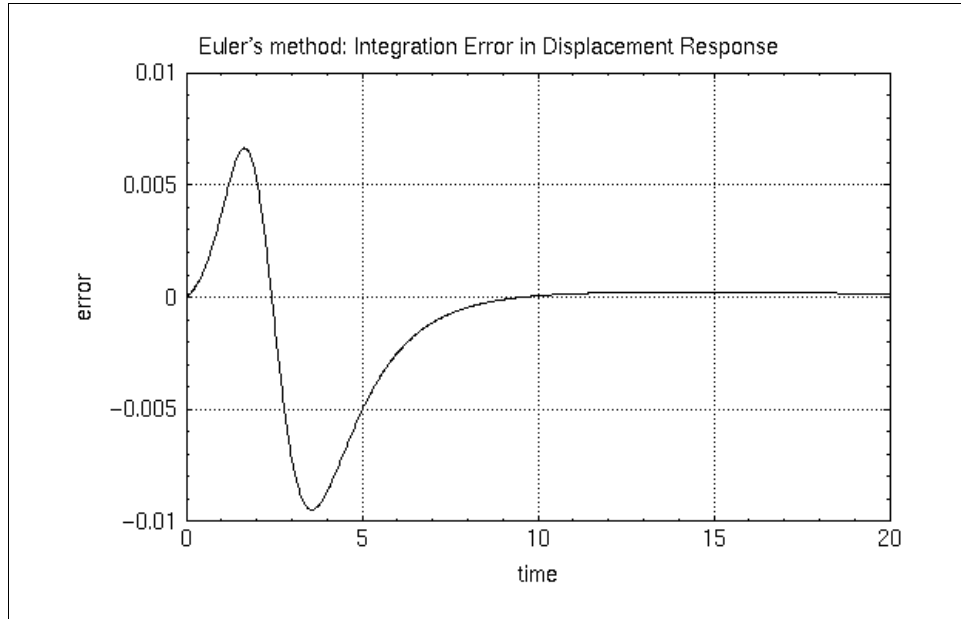
The remainder of this example compares the various errors exhibited by each integration algorithm, where the error was computed by taking the difference of the exact value and the value yielded by the algorithm of interest. The position response for this svsystem is shown in [Figure 13-11](#).

Figure 13-11 **Position Response**



The integration error for the Euler algorithm is shown in [Figure 13-12](#). This algorithm displays the largest errors, with a deviation of up to 10% of the maximum amplitude of the response.

Figure 13-12 **Errors in Euler Integration**



Figures [13-13](#) and [13-14](#) present the errors of the remaining six algorithms. In [Figure 13-14](#), 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.

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 13-13](#). Note the vertical scale; the error magnitudes are less than one millionth of the response magnitude.

Figure 13-13 **Errors in Fixed and Variable-Step Integration Algorithms (1)**

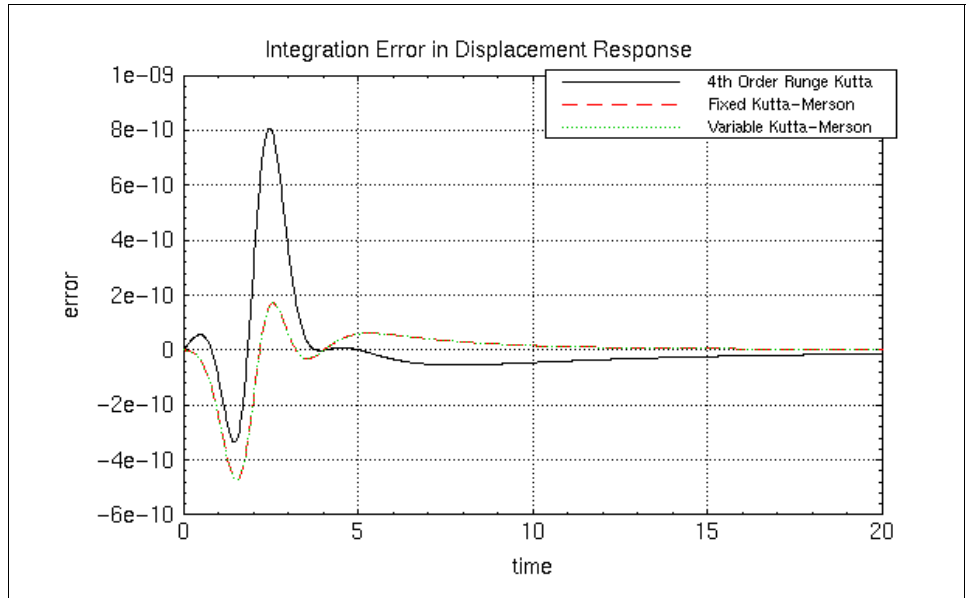
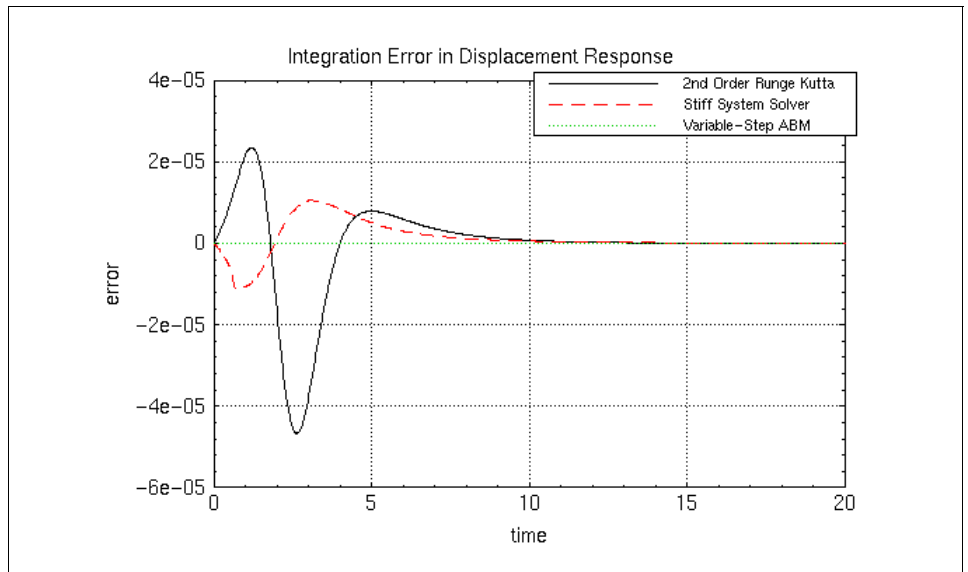


Figure 13-14 **Errors in Fixed and Variable-Step Integration Algorithms (2)**



Varying **reltol** had no observable effect for the variable-step Kutta-Merson method for this specific example. [Figure 13-15](#) presents the effect of changing the relative tolerance **reltol** for variable-step Adams-Moulton. [Figure 13-16](#) and [Figure 13-17](#) show the effect of varying **reltol** for variable-step Kutta-Merson and for the stiff system solver.

Figure 13-15 **reltol and Variable-Step Adams-Moulton Method**

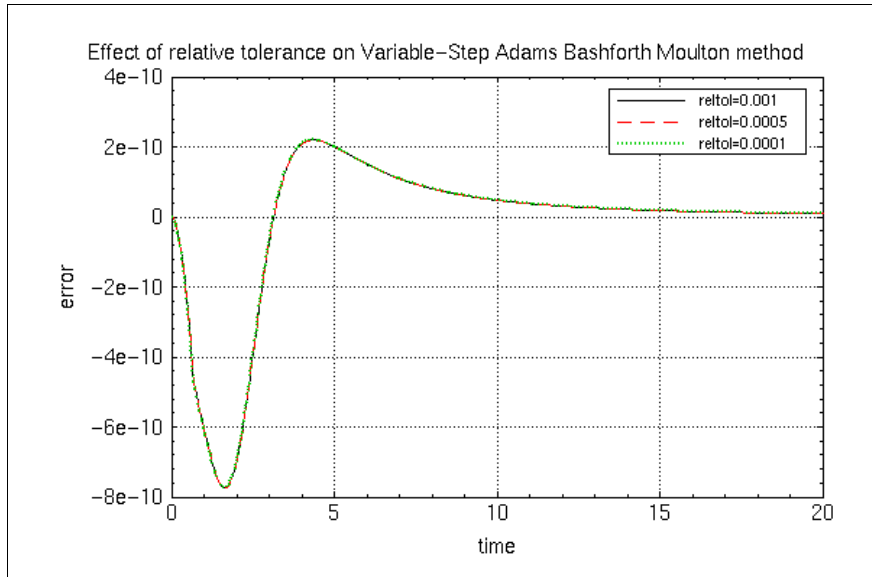


Figure 13-16 **reltol and Variable-Step Kutta-Merson; All Curves Superimposed**

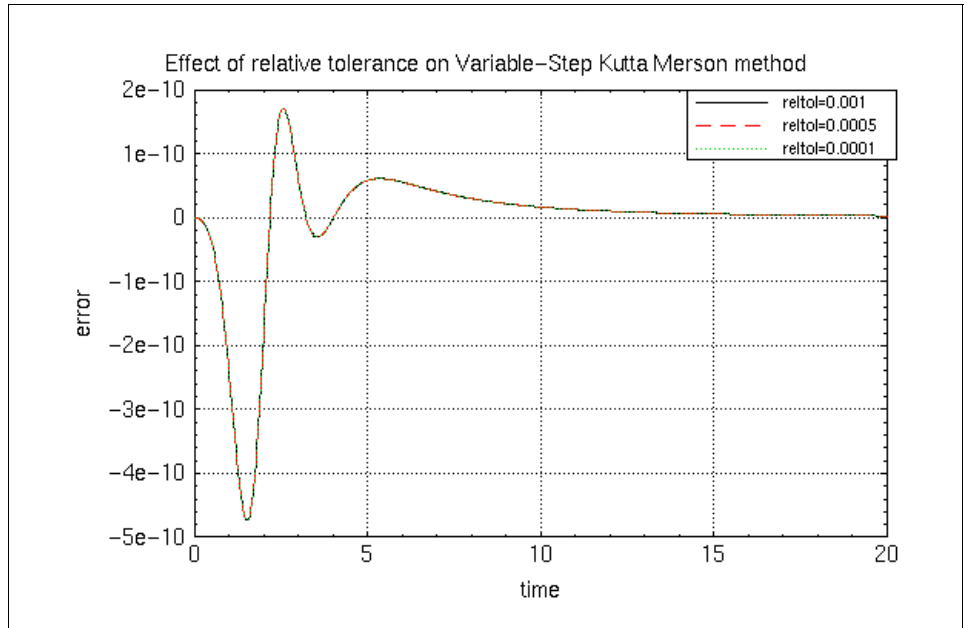
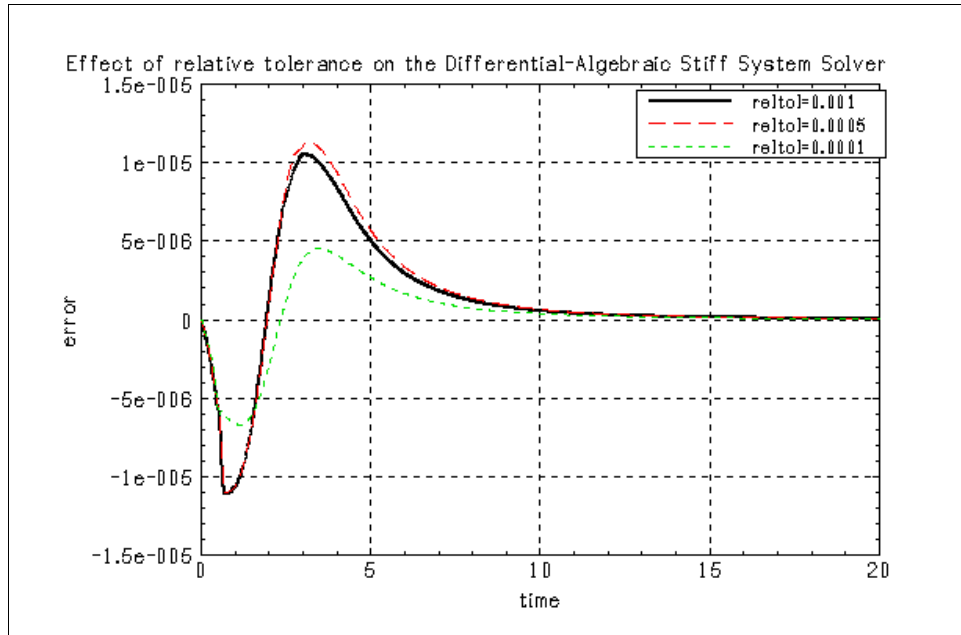
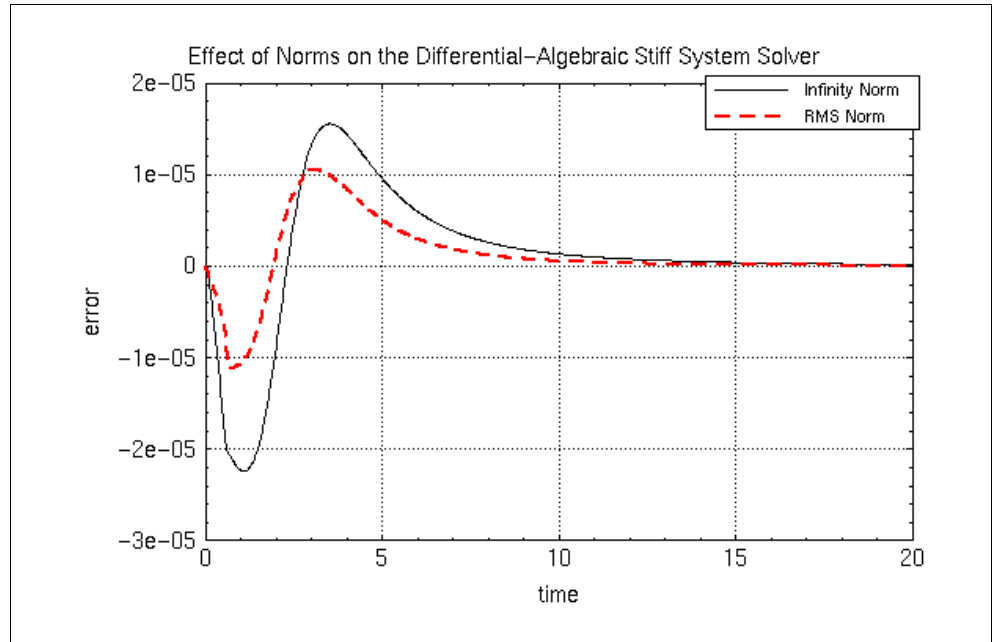


Figure 13-17 **reltol and the Stiff System Solver**



For this example, changing certain options can improve accuracy without adding too much computational burden. Depending on the type of problem, you can improve the performance of the stiff system solver by changing the norm computation method. [Figure 13-18](#) 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 on which SystemBuild is running. In particular, the machine epsilon of your specific platform effects some of these results. (The machine epsilon for these simulations was $\epsilon = 2.2204E-16$.)

Figure 13-18 Norms and the Stiff System Solver



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.

13.6 State Events

A difficulty in the modeling of continuous systems occurs when a system shows any of a class of local discontinuities (*state events*²), 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

2. In some literature such events are called *switch events*.

continuous UserCode 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—that is, potential changes in either the model or the controller.

State events are also used to simulate interrupts associated with asynchronous trigger SuperBlocks (see [p.135](#)).

The objective of state event modeling is to circumvent attempts by the numerical integration code to integrate across drastic changes (that is, instantaneous changes of state values in system equations) in the system. Usually such attempts 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. If state events are handled properly, observe that 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). We recommend that you first try to use ZeroCrossing blocks because they are simpler to use and do not require writing code to handle state events. For more complex situations where multiple state events are monitored or are interrelated, the state event capability within a UCB might be preferable. Examples of the latter include a limited slider with multiple hard stops and a Coulomb friction model (Limited Slider and UserCode block (Coulomb) demos). (See [1.4 Running SystemBuild Demos](#), [p.4](#) for instructions on how to run demos.)

13.6.1 ZeroCrossing Block

The ZeroCrossing block is located on the User Programmed palette. Any signal connected to the ZeroCrossing block is monitored for a sign change,³ and the exact instant of the sign change is pinpointed.⁴ Once the zero crossing is found, the output of this block changes from 0 to 1 or from 1 to 0, depending on its

3. 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.
4. A time point will be generated at the zero-crossing time.

previous state (that is, $y(k+1) = 1 - y(k)$ for the k th zero crossing). The block output is always initialized to 0 at the beginning of a simulation.

The simulator always restarts (and the operating point is recomputed) when a zero crossing is found. Therefore, the output of the ZeroCrossing block can be used to detect discontinuities and trigger various events elsewhere in the model (for example, resetting a resettable integrator to avoid integrating over a discontinuity). Note that asynchronous trigger SuperBlocks whose trigger signal is the output of the ZeroCrossing block 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 or a double edge. A *single-edge* transition is from zero or negative to positive. A *double-edge* transition is *either* a transition from zero or negative to positive *or* a transition from positive to zero or negative.

The interface of the ZeroCrossing block is simple: the number of input signals monitored is the same as the number of output signals for selection of zero crossings.

See the Integrator block in online Help.

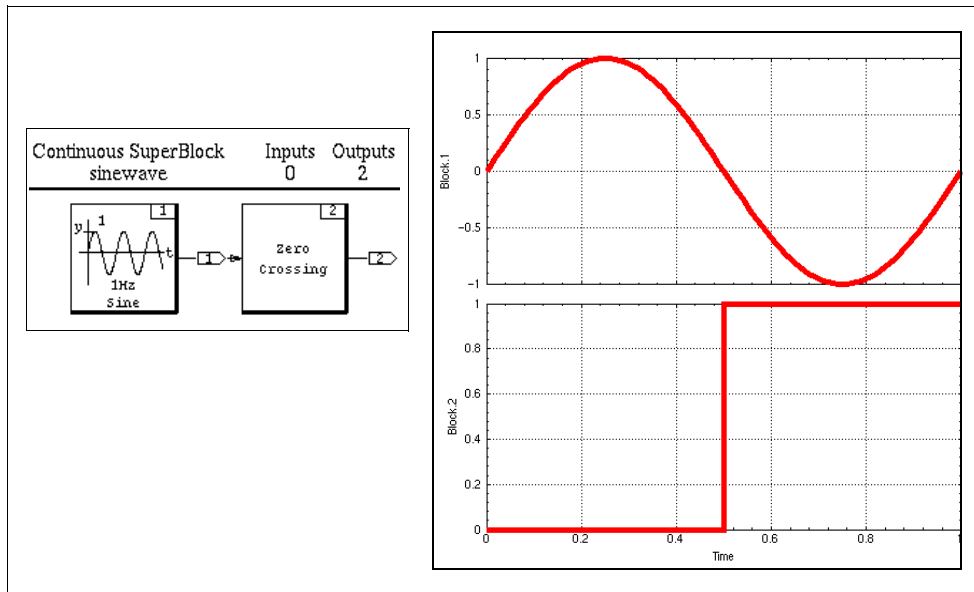
Example Using a Sinusoid Signal

You can create a new SuperBlock that contains a sinusoid signal as input to a ZeroCrossing block with the following commands issued in the Xmath Commands window:

```
createsuperblock "sinewave",{inputs=0,outputs=2}
createblock "sinwave",{id=1}
createblock "zerocrossing",{id=2}
createconnection 1,2;createconnection 1,0;createconnection 2,0
t = [0:.01:1]';
y = sim("sinewave",t,{extend,graph});
```

Figure 13-19 shows the SuperBlock that is created and the output from each block in it. Note the step function change in the output of the ZeroCrossing block when the sin wave crosses 0.

Figure 13-19 Zero Crossings of a Sine Wave



Example Using a Bouncing Ball

Example 13-8 uses a bouncing ball model with a ZeroCrossing block.

Example 13-8 Impact of a Bouncing Ball

In this example, a bouncing ball is modeled. 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 13-20](#).

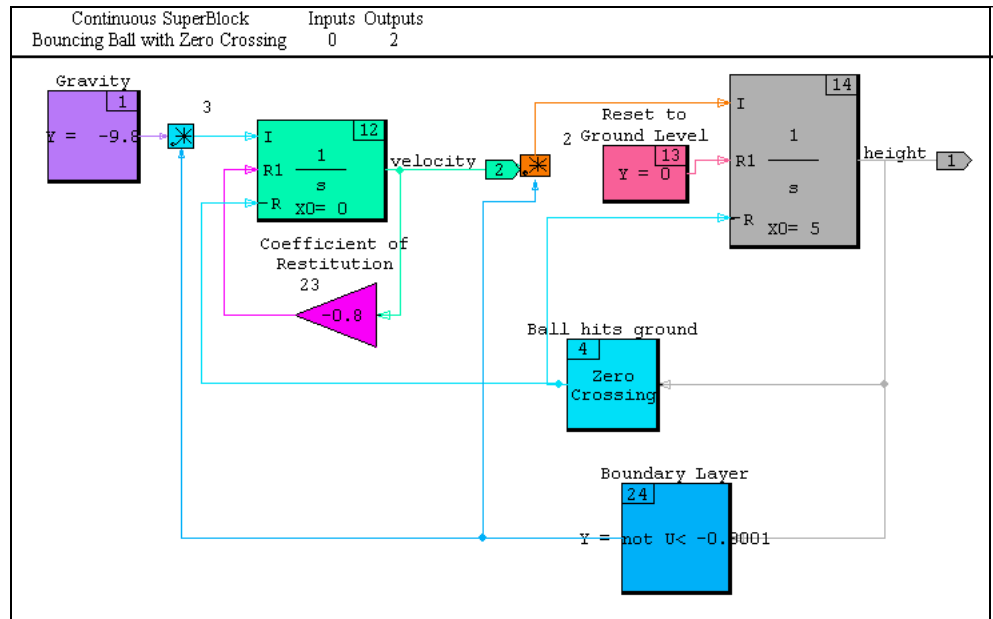
To run the bouncing ball example:

1. Copy the example to your local directory:

```
copyfile "$SYSBLD/examples/bouncer_example/bouncingball.cat"
```

2. Load the model into the Catalog Browser, and then open the Bouncing Ball with Zero Crossing model in a SuperBlock Editor.

Figure 13-20 **Bouncing Ball Model**



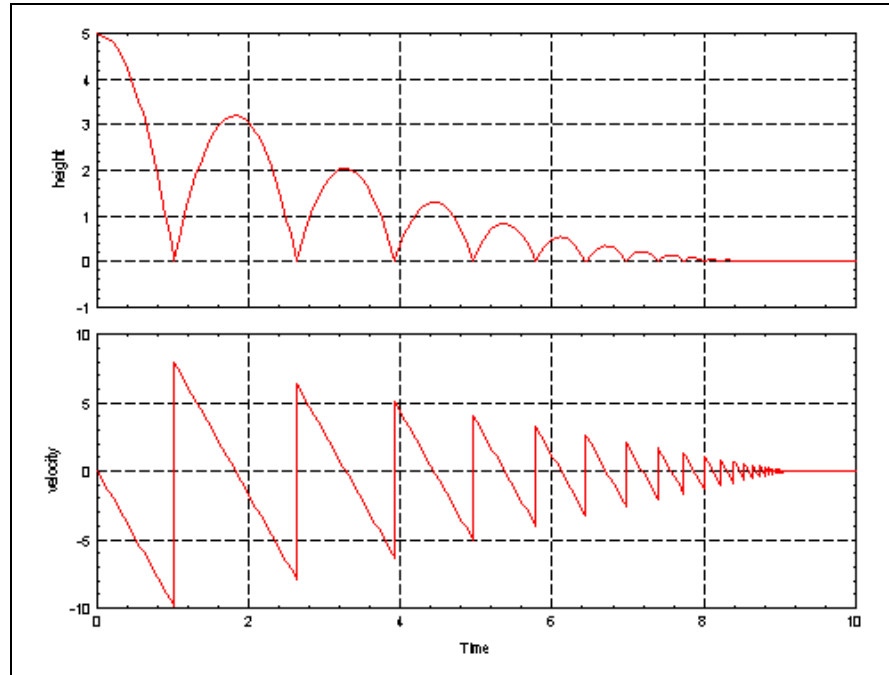
3. Type the following commands into the Xmath command area to simulate the model:

```
t = [0:0.05:10]';
y = sim("Bouncing Ball with Zero Crossing", t, {graph, extend})
```

[Figure 13-21](#) shows the output of the simulation. 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 13-21 **Plot of Bouncing Ball Example**



13.6.2 Continuous UserCode Blocks

You can also find the UserCode block on the User Programmed palette. The dialogs for both explicit and implicit UCBs include a States field for entering the number of state event signals. The signals are monitored for a sign change. When one occurs, the exact time of the change is pinpointed, and actions that you specify are then executed. The mechanisms for having this occur are described in the following paragraphs.

Associated with each UCB is a user-written program for defining state events. The user-written code defines two things:

- The signal to monitor for occurrence of a state event
- The actions to be executed at the instant of the event

Usually, the state event problem is formulated as follows:

$$\dot{x} = f(x, u, t) \quad (13-50)$$

$$y = g(x, u, t) \quad (13-51)$$

$$z = h(x, u, t) \quad (13-52)$$

where (13-51) and (13-51) describe the system, and (13-52) defines a set of monitoring functions whose roots (or zero crossings) signal a possible state event.

To help you build UCBs for state events, we provide the following template files:

`SYSBLD/src/usr01.c`

`SYSBLD/src/iusr01.c`



NOTE: `iusr01.c` is intended for implicit UCBs only. See [14.1.2 Implicit UCBs](#) on p.325 for further details.

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)

Two flags are provided to be used for state-event simulations: **monit** and **event**. The UCB is called with **monit=1** during every integration time step. In the **monit** section, the monitor function (that is, the zero-crossing signal defined by (13-52)) should be calculated. 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 zero-crossing event is found, the integrators take a step just up to the time of the event, and then the **event** section code described in the template is 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—that is, continued from the reset values.

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.

- 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—that is, stays at zero at more than one integration step. You must take care to avoid this situation; that is, you must ensure 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 are 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 ZeroCrossing 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 (that is, $y = T$), and feeding the signal ($T_{event} - T$) 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 (for example, the DataPathSwitch block). Also, within a UCB, user-written code in the **event** section can activate a flag called by user code in the **state** section when determining which state update equations to use.

The following restrictions and limitations apply to UCBs:

- State events are only defined for continuous UCBs.
- No zero crossings can be detected at time zero.
- 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 are not detected.
- Degenerate zeros (that is, successive zero values) are not detected as zero crossings. Degenerate zeros result in a failure of the simulation.

- Fixed step integrators do not calculate the zero crossing instant as accurately as the variable step algorithms.
- QuickSim (**ialg = 8**) does not detect any zero crossings because this algorithm calculates its solution based on a linearization **at $t = 0$** .

14

UserCode Blocks

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.

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 UCB template files 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 is limited to what AutoCode supports.

Before using UCBs, you must familiarize yourself with programming, compilation, and linking procedures for your hardware platform and environment, which is largely what this chapter is about.

The main topics in this chapter are as follows:

- *The Numerics of UCBs*
- *The Structure of UCBs*
- *How SystemBuild Executes UserCode Blocks*
- *Variable Interface UserCode Blocks*
- *UCB Programming Considerations*
- *Building, Linking and Debugging UCBs*
- *Posting Error Indications*
- *Simulation API*

See online Help for the UserCode block for information on the block dialog itself.



NOTE: 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 [].

14.1 The Numerics 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.

14.1.1 Explicit UCBs

In continuous systems, the explicit UCB can represent a set of first-order ordinary differential equations (ODE) of the form:

$$\begin{aligned} \dot{x} &= f(x, u) \\ y &= g(x, u) \end{aligned} \tag{14-1}$$

In discrete systems, it can represent a set of first-order difference equations of the form:

$$\begin{aligned}x_{k+1} &= f(x_k, u_k) \\ y_k &= g(x_k, u_k)\end{aligned}\tag{14-2}$$

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 calculates x and passes 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 [14.2.5 State Events](#) on p.338).

14.1.2 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:

$$\begin{aligned}0 &= f(x, \dot{x}, u) \\ y &= g(x, \dot{x}, u)\end{aligned}\tag{14-3}$$

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 side of the equation must be in the proper form:

$$f(x, \dot{x}, u) = f(x, u) - \dot{x}\tag{14-4}$$

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:

$$\begin{aligned}0 &= f(x, \dot{x}, u) \\0 &= f_c(x, \dot{x}, u) \\y &= g(x, \dot{x}, u)\end{aligned}\tag{14-5}$$

where f has dimension nx and the constraint equation f_c has dimension nc , 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, nc , is specified in the ImplicitUserCode Block dialog on the Parameter tab.

To see how these equations are solved, see [Over-determined Differential Algebraic System Solver](#) on p.294. This section includes an example of a pendulum.

14.2 The Structure of UCBs

In this section, we discuss the seven [Modes of Operation](#) in UCB functions and the [UCB Templates](#) that support the creation of these modes. Then we tell you about [UserCode Function Calling Arguments](#). Following these topics are two somewhat specialized topics, [Direct Terms](#) and [State Events](#).

14.2.1 Modes of Operation

There are seven distinct modes of operation associated with UCBs: **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 14-1](#). The sequence and the modes of operation the UserCode function is required to compute depends on the information specified in the Parameters tab of the UserCode Block dialog.

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 is called in **INIT** mode twice:

- Once while executing **OUTPUT** mode
- Again while executing **STATE** mode

If the UCB reference does not have states, the UCB is called in **INIT** mode only once (while executing the **OUTPUT** mode).

Typically, no action is required in the **INIT** section because the initial conditions defined on the Parameters tab of the UserCode Block 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: If the UCB performing these tasks has states, it is your responsibility to ensure that these operations occur only once (even though the UCB is 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 x , 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 *only* required 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 blocks) 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 instructs the simulator to call your UserCode function to evaluate the following two expressions in the *F* and *Y* arguments when a linearization is needed:

$$\delta \dot{x} = \left. \frac{\partial f}{\partial u} \right|_{x_{op}, u_{op}} \delta x + \left. \frac{\partial f}{\partial u} \right|_{x_{op}, u_{op}} \delta u \quad (14-6)$$

$$\delta y = \left. \frac{\partial g}{\partial x} \right|_{x_{op}, u_{op}} \delta x + \left. \frac{\partial g}{\partial u} \right|_{x_{op}, u_{op}} \delta u \quad (14-7)$$

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, for example, by closing files, deallocating memory, and other housekeeping. No parameters are passed with this mode.

14.2.2 UCB Templates

UserCode function templates are furnished in the *SYSBLD/src* directory. The templates, **usr01.c** (C language) and **usr01.f** (FORTRAN), are for explicit UCBs, while **iusr01.c** and **iusr01.f** are for implicit UCBs. Each template consists of a single function prototype with sections designed to contain the computational algorithms associated with the seven distinct modes of operation.

If you want the same UCB code to work for both SystemBuild simulation and AutoCode, then you must use the **sa_user.c** template found in *CASE/ACC/src*. See the *AutoCode User's Guide* for more information.

Copy the appropriate file to your working directory and insert your custom simulation algorithms in the example function.

14

14.2.3 UserCode Function Calling Arguments

The arguments for the explicit UserCode function are:

```
USR(IINFO, RINFO, U, NU, X, F, NX, Y, NY, RPAR, IPAR)
```

The arguments for the implicit UserCode function 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.

Refer to the listing of the **usr01** or **iusr01** templates for more information.

The meaning of U , Y , X , and F vary 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 on the Parameters tab of the block dialog so that a given UCB can be reused in a model with different parameters. You can enter the *RPAR*, *IPAR*, and initial conditions parameters as %Variables. This lets you modify the routine logic and equation coefficients between simulations without having to re-edit the model.

The sections on the following pages define the types of variables you encounter using UCBs.

***IINFO* Array**

IINFO is an integer array (see [Table 14-1](#)) that contains information flags used for communication with the simulation engine. Except as noted in the discussion of *INIT Mode*, 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 14-1 **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 <i>F</i> .
IINFO(4)=1	OUTPUT mode. Compute outputs in <i>Y</i> .
IINFO(5)=1	LIN mode. Compute linearization in <i>F</i> and <i>Y</i> .
IINFO(6)=1	MONITOR mode. Compute monitor function for state event detection in <i>F</i> .
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).
IINFO(14)=1	Update with converged state values.
IINFO(15)=1	Integration algorithm (IALG).

RINFO Array

RINFO is a real array (see [Table 14-2](#)) that contains timing and related information for the called routine. UserCode may not modify any values in RINFO.

Table 14-2 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)	Simulation start time	Simulation start time	Simulation start time
RINFO(6)	Simulation end time	Simulation end time	Simulation end time
RINFO(7)	reitol	reitol	reitol

Mode Parameters

[Table 14-3](#) defines all the Mode parameters associated by all the modes of the UCBs.

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

Table 14-3 **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 14-3 **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:

$$df = \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial u} du + \frac{\partial f}{\partial \dot{x}} d\dot{x} \quad (14-8)$$

$$dg = \frac{\partial g}{\partial x} dx + \frac{\partial g}{\partial u} du + \frac{\partial g}{\partial \dot{x}} d\dot{x} \quad (14-9)$$

$$df_c = \frac{\partial f_c}{\partial x} dx + \frac{\partial f_c}{\partial u} du + \frac{\partial f_c}{\partial \dot{x}} d\dot{x} \quad (14-10)$$

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.

14.2.4 Direct Terms

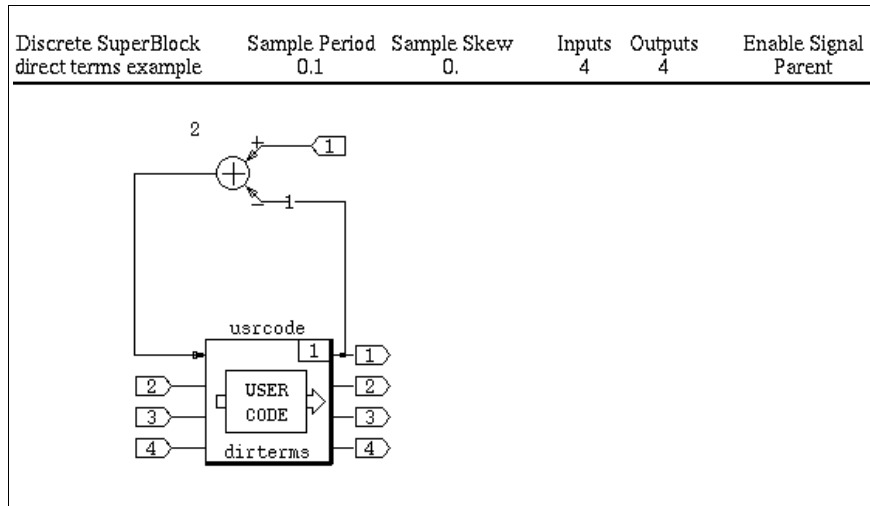
As previously mentioned, UCBs are systems that solve the following linked pair of equations,

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

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 14-1](#) not only possible, but solvable without resorting to implicit solvers.

Figure 14-1 **UCB with Direct Terms**



If none of the outputs is 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 is 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 assumes that this construct forms an algebraic loop that either requires

the use of an implicit System Solver (if the SuperBlock is continuous) or the insertion of a time delay between the UCB and the Summer.

To make an input a direct term, set the Input Direct Terms vector on the UCB's Parameters tab. This vector is of the same size as the number of inputs to the UCB. 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.

Properly identifying direct terms assists the simulator and AutoCode in the detection of algebraic loops in the model. More correctly stated, this capability allows the simulator to note that some loops that are seemingly algebraic are actually not.



NOTE: 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 14-1](#) provides an example of using direct terms.

Example 14-1 **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 tskew   = 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;
}

```

14.2.5 State Events

In continuous systems, both explicit and implicit UCBs support integration of piecewise-continuous models, that is, 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:

$$f_m(x, u, t) \quad (14-11)$$

and, for implicit systems, it is of the form:

$$f_m(\dot{x}, x, u, t) \quad (14-12)$$

where f_m is of order NSE , the Number of State Events specified on the Parameters tab of the UCB block 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 (for example, at every integration step), this technique becomes very inefficient.

If Number of State Events is greater than zero on the Parameters tab of the block 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 locates this point within a tolerance (**ztol**) with respect to the time axis and halts 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 (that is, $f(x, \dot{x}, u) = 0$) 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 makes individual calls for each event.

14.3 How SystemBuild Executes UserCode Blocks

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 is executed in a simulation. These are discussed in the following sections.

14.3.1 Execution of STATE Versus OUTPUT Modes in the UserCode

Usually when a block is executed, it is called in two passes: **OUTPUTS** and **STATE** modes (see [14.2.1 Modes of Operation](#) on p.326). 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 is not called during STATE updates*. Thus, in order to debug the integration algorithm, internal steps for states and derivatives (that is, derivative evaluations at non-converged time points), the UserCode block must be created with at least one (dummy) state. This ensures that the block is in the appropriate chain of blocks to be executed during state updates.

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, place the UserCode block in the rightmost pane using a Sequencer block. This ensures that when the analyzer sorts the blocks, the UserCode block is the last one to be executed.

14.3.2 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, and so forth

14.3.3 Initialization

Simulation INIT Modes

The initialization type for simulation is set by the **initmode** keyword, and it can take the values [0–4]. The default value is **initmode=3**.

Type 0 Initialization

Only the continuous subsystems are initialized. The UserCode block is executed once to initialize all system outputs. This is called the **INITUserCode()** 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 an ImplicitUserCode block or algebraic loops, so that a steady-state operating point can be found. Discrete subsystem outputs are left at **-sqrt(eps)**.

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.

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.

Type 3 Initialization

This is the same initialization type as for type 2 except that outputs are propagated instantaneously between subsystems.

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(eps)**.

Impolite UCB Initialize Mode

In the ImplicitUserCode Block dialog, the Initialize Mode field allows you to set 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 definition.

Table 14-4 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** and **xd0** override the dialog defaults.

14.3.4 Numerical Integration Algorithm

The integration algorithm can take the values [1–10] (see [13.5.1 Comparing Integration Algorithms](#) on p.285).

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.

The timestep `tk` is chosen at every point by computing the minimum of:

- The difference between the current time and the next discrete, triggered, or enabled event
- The difference between the current time and the next external output time determined by the user time vector.
- **dtmax**
- **dtout**

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 (that is, 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 the Xmath Comands window or the SuperBlock Editor. The debugging feature provides access to these values.

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

14.3.5 Operating Points

In this section, we provide some notes about operating points for UCBs.

Implicit Integration Algorithm Operating Point

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.

Implicit implementation algorithms (DASSL, ODASSL and GEARS) require that the operating point be consistent (that is, the residual δ is 0 or very small) at the beginning of a simulation. Under normal conditions, SystemBuild's operating point solver computes 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:

$$\frac{\partial}{\partial x} f(x, \dot{x}, u) \quad (14-13)$$

or

$$\frac{\partial}{\partial \dot{x}} f(x, \dot{x}, u) \quad (14-14)$$

(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 has a special feature that facilitates this when it is used with `initmode = 4` (see [Type 4 Initialization](#)). 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 the residuals $\delta = f(x_0, x_{d0}, u_0)$, *not the derivatives*, 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.

Computing the Operating Point Jacobian Matrix

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.

Computing the Implicit Solver Jacobian Matrix

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:

$$J = \frac{\partial f}{\partial x} + c \frac{\partial f}{\partial \dot{x}} \quad (14-15)$$

where $f = f(x, \dot{x}, u)$, and c is an iteration constant determined by the Implicit Solver.

14.4 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.

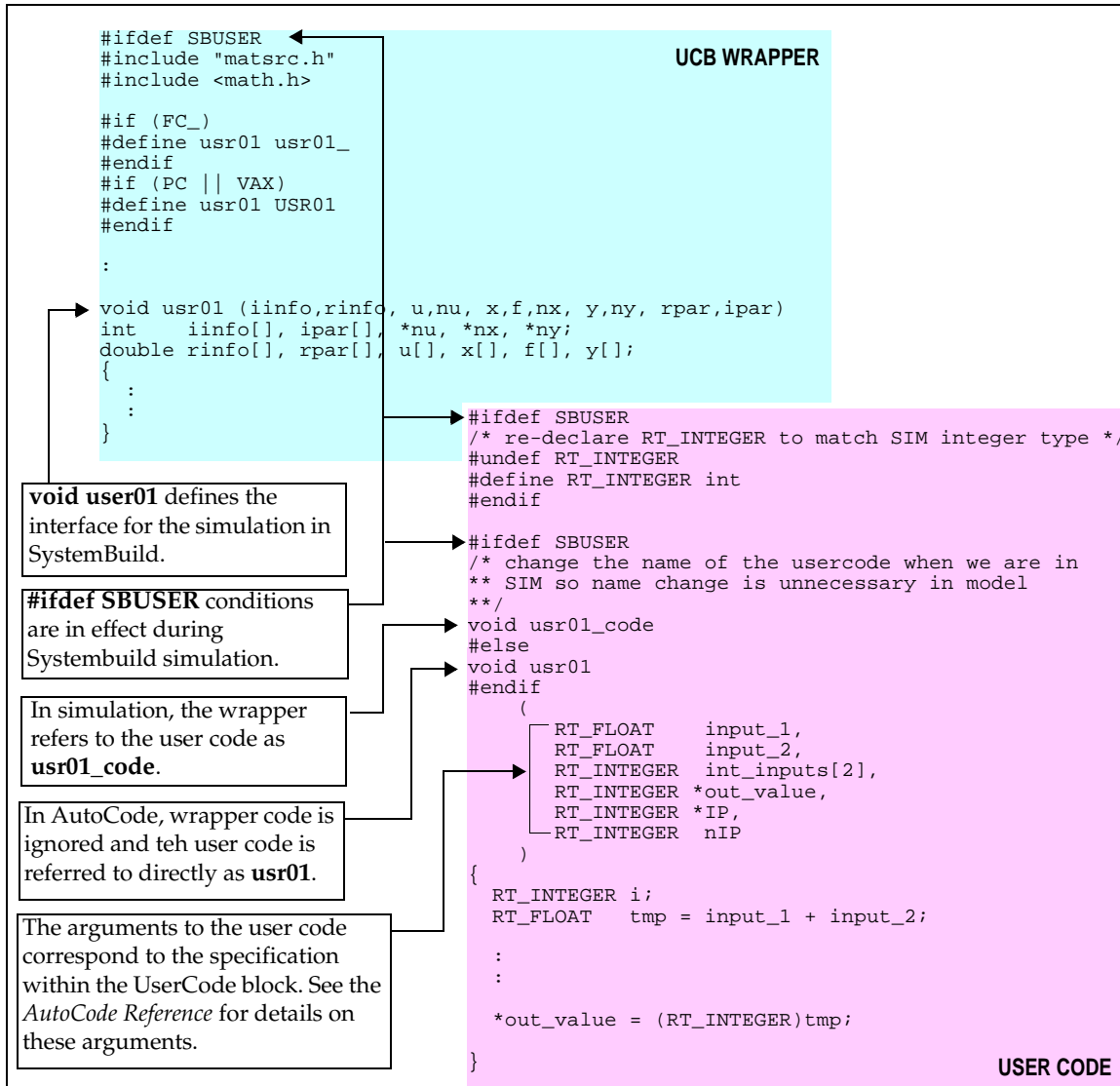
Although SystemBuild UCB code (see [14.2.1 Modes of Operation](#) on [p.326](#)) cannot be used in AutoCode, handwritten UCB code intended to be linked with AutoCode can be linked into a SystemBuild simulation if the user code conforms to a standard interface. A standard UCB Interface Type can be either 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 data type specified in the block diagram.

The variable interface accommodates mixed data types and optional arguments. It passes the input signal data types and output data types specified in the UserCode Block dialog. In addition, signal labels and names from the dialog 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.

this code for both SystemBuild and AutoCode. [Figure 14-3](#) highlights portions that are of particular interest.

Figure 14-3 Excerpts from samp_vucb.c



14.4.2 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. 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:

- Create local variables that match the data type and shape of variable interface UCB inputs and outputs.
- Convert the data from the `sim()` interface to the local variables (see [Converting Data from the `sim\(\)` Interface](#)).
- Initialize the *INFO* status record (if needed) to indicate the appropriate mode (*INIT*, *OUTPUT* or *STATE*).
- Call the variable interface UserCode using the local variables as arguments to the function.
- Convert the local variables representing the output data back into the `sim()` interface.
- 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 data types other than float, the `sim()` data must be converted to the appropriate data type.

To perform data type conversion, use the AutoCode SA Library header files and some of the macros defined within them. Refer to the *AutoCode Reference* 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**.

14.4.3 Specifying the Variable Interface

The UserCode block parameters are fully documented in online Help. To see this information, type **help usercode** from the Xmath command line, or click the Help button in 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 activated when the Interface Type is set to Variable; select the appropriate action for each (Yes or No).

Specifying Data Types

Input data types are inherited from source blocks. Output data types are specified on the Outputs tab of the UserCode Block dialog.

Specifying Input Shapes

Input labels and names determine the structure of input data to the user code. See [Creating Sequential Names for Vectors and Matrices](#) on p.99. The precedence for creating the shape (scalar or vector) is determined by:

1. Input names (specified on the Inputs tab of the UserCode Block dialog)
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 and Matrices](#) on p.99. The precedence for creating the shape (scalar or vector) is determined by:

1. Output names (specified on the Outputs tab of the UserCode Block dialog)
2. Output labels (specified on the Outputs tab of the UserCode Block dialog)
3. UCB block name

14.4.4 Running a Variable Interface Example

We provide an example that you can simulate and for which you can generate code.

Simulating the Variable Interface UCB in SystemBuild

To simulate the variable interface example:

1. Using the following commands, 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"
```

2. Load the file **test_vucb.cat** into SystemBuild.
This file contains both model data and Xmath variables.
3. Open the SuperBlock **test_vucb**.
4. Select Tools→Simulate.
5. On the Parameters tab of the SystemBuild Simulation Parameters dialog, make the Time Vector **t** and the Input Data **u**. Give the Output Variable the name **tv_sb**. Enable Plot Outputs and Typecheck. Then click OK.

The simulation results are shown in a strip plot and saved to the variable *tv_sb*.

Generating Code for a Variable Interface UCB in SystemBuild

To generate code for the variable interface example:

1. In the SystemBuild Catalog Browser, select the SuperBlock **test_vucb**.
2. Select Tools→AutoCode.
3. Click OK.

For instructions on compiling and linking the user code, see the section titled "Standalone Simulation" in the *AutoCode User's Guide*. Then you can compare the generated code outputs with the simulation outputs.

14.5 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 you are 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, and so forth.
- 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 are usable under both SystemBuild and your other packages.

14.6 Building, Linking and Debugging UCBs

The simulator is responsible for reading information about UserCode from the model and other sources and then taking that information and collecting, compiling and executing UserCode. This section explains this process. As much as possible, the discussion is 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 resides 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 Release 6.0, new simulator executables (**simexe.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.



CAUTION: Any time you simulate or generate code for a model that contains UCBs, that model should exist in a separate directory. Otherwise, you risk mixing objects between models because there is only one **simucb** shared library per working directory.

14.6.1 Collecting UserCode Files

UserCode files can be specified from the Parameters tab of the UserCode Block dialog, with the keywords **csource** and **fsource**, in the makefile itself, or with the **ucbcode loc sim** 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.

Parameters Tab of the UserCode Block Dialog

The primary location for entering a filename is the Parameters tab of the UserCode Block dialog. This filename may be a simple filename (for example, **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, **%UCBCODELOC%\usrcode.c** on a Windows system). Only environmental variables that exist before SystemBuild is invoked are recognized by the simulator.

CSOURCE and FSOURCE

You can specify additional source code files using the keywords **csource** (for C language files) and **fsource** (for FORTRAN language files). You can specify **csource** and **fsource** files 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 filenames are 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. (The environment variable **\$\$SYSBLD** is available from within the product, and it takes the UNIX form of environment variables.) You can copy the makefile into the local project directory; if it exists there, the simulator uses 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 attempts 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

You can use the **sim()** command line option **ucbcode loc** to specify a location for user code that is different from the local directory. If you supply this option, the simulator assumes that all UCB source files, as well as the UCB shared library, are located in the directory specified with the **ucbcode loc** option; the exception is any source code file that has a complete path name as its specification, in which case, the simulator looks for the file there.

You can also set this location using **SETSBDEFAULT**. The keyword **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"}
```

14.6.2 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, **makefile** 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 are reported by the simulator, which then terminates.

Windows

On Windows platforms, a dynamic link library (DLL) is created. Because a DLL is a name-resolved standalone 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 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, user code attempts to call a non-existent function, the simulator immediately stops executing with a dynamic linker error.

Supported Compilers

Wind River tests the UCB capabilities of the simulator using the compilers (and compiler versions) that are officially supported by Wind River. If an unsupported compiler is used to compile and link UCB shared libraries, Wind River 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 *SystemAdministrator's Guide* or on our website at <http://www.windriver.com>.

14.6.3 Debugging User Code

Follow these guidelines when debugging user code:

1. To debug user code in the simulation environment, compile and link the code with debug information.

UNIX

On UNIX platforms, the code is compiled and linked with the debug information automatically.

Windows

On Windows platforms, you need to copy the makefile, **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 ensure that the top portion of the makefile contains the line:

```
DBG = Yes
```

Next, create the UCB shared library using the **linksim** command. When you eventually enter the debugger, this guarantees that the UCB shared library is already be created and loaded into the simulator. For more information on the **linksim** command, consult 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 from the Tools menu on the SuperBlock Editor.

Once the simulator is invoked, a GUI dialog appears.



CAUTION: Do not click OK in this dialog until the debugging session is ready for it ([Step 5](#)).

UNIX

On UNIX systems, you 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 invokes the debugger and attaches it to the simulation process.

Windows

On PC systems, an OS dialog also comes up, notifying you that a debug exception has occurred and asking if you would like to invoke the debugging environment. Do so.

4. Once you are in the debugging environment, do any initialization you like (setting break points, data watch points, and so forth).



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 generally have a trailing underscore (UNIX systems) or are in all capital letters (Windows systems) due to the mixed C/FORTRAN language support in the simulator.

5. After the debugging session is initialized, ensure that the simulator is running (by using the **cont** command on most UNIX debuggers or by pressing the F5 key in the Windows environment).
6. Once the simulator is running, you may click OK in the Xmath GUI dialog. At this point the simulator resumes, and you can continue the debugging session.

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 UCB doesn't crash the simulator and you do not issue the command **undefine simexe** in the Xmath command area.

14.7 Posting Error Indications

You can write messages from your UCB code, and SystemBuild provides messages, as appropriate, at simulation time. These two mechanisms are discussed below.

14.7.1 Writing User Messages to the Xmath Window

Use the function `stdwrt()` supplied by Wind River 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)
```

14.7.2 Simulation Errors

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:

```
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.

The following is a list of 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.

14.8 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. It has these classes of capabilities:

- *Gathering UCB Reference Information*
- *Accessing and Modifying Variables*
- *Accessing Simulation Debugging Information*, including information about the state of the continuous integration



CAUTION: UCBs that take advantage of the SIMAPI capabilities are not transferable to the AutoCode environment.

For C language UserCode blocks, adding a reference to the header file **simAPI.h** provides access to the SIMAPI functionality.

This capability is not recommended for FORTRAN UCBs. If you attempt SIMAPI access, it is your 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. (The environment variable **\$SYSBLD** is available within MATRIX_X.)

14.8.1 Gathering 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 allows 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 you enter 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 initializes the **SB** and **BLK** structures that are 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.

[Example 14-2](#) uses the SIMAPI to get the name of the currently executing block reference.

Example 14-2 **Getting 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**.

[Example 14-3](#) uses the SIMAPI to get the label associated with output number three of the current UCB reference.

Example 14-3 **Getting the Label Associated with a Specific Output**

```
BLK    *blk;
SB     *sb;
char   *name;

SIMAPI_GetUCBBlockInfo(iinfo, &sb, &blk);
name = SIMAPI_GetBlockOutputLabel(blk, 3);
```

14.8.2 Accessing and Modifying Variables

The SIMAPI also provides users the ability to access variable information (%variable and ReadVariable/WriteVariable 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( );
```

Returns the number of variables (ReadVariable/WriteVariable block and %variables) 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 data type 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 is set to 0 (that is, if the variable is a 3x4 array, this array is 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 is allocated by this routine and is of size `SIMAPI_GetVarStorageSize()`. The data is of whatever type is specified by `SIMAPI_GetVarDatatypeNameblock()`.

```
long SIMAPI_PutVarData(long varnum, void *data);
```

Edits the selected variable. 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 is flushed (the edited value is 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()`.

Once a variable edit has been flushed (by `SIMAPI_FlushVars()`), it cannot be canceled.

[Example 14-4](#) uses the SIMAPI to retrieve the name of the third variable:

Example 14-4 Retrieving the Name of a Variable

```
char *varname;  
varname = SIMAPI_GetVarName(3);
```

Example 14-5 uses the SIMAPI to change the value of a known scalar variable named *myvar* to 20:

Example 14-5 Changing the Value of a Variable

```
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);
    }
}
```

14.8.3 Accessing 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 variables are input arguments to functions listed in this section:

<i>extinp_index</i>	<i>extout_index</i>	<i>impout_index</i>	<i>impout_index</i>
<i>is_available</i>	<i>IS_Jacobian</i>	<i>state_index</i>	<i>status</i>

The following variables are output arguments to functions listed in this section:

<i>deriv_value</i>	<i>extout_value</i>	<i>n_ext_inputs</i>	<i>state_name</i>
<i>extinp_name</i>	<i>iinfo</i>	<i>n_ext_outputs</i>	<i>state_value</i>
<i>extout_name</i>	<i>impout_name</i>	<i>n_imp_outputs</i>	<i>n_ext_inputs</i>
<i>extinp_value</i>	<i>impout_value</i>	<i>n_states</i>	

Note that all functions described in this section return an integer status value:

```

SIMAPI_OK = 0,          /* Call was successful */
SIMAPI_Unsuccessful = 1, /* Call failed */

```

The following subsections list the functions available through the debugging interface. In the usage discussions, the function prefix **SIMAPI_** has been omitted from the descriptions for brevity.

Functions to Initialize and Terminate Debug Data

```

SIMAPI_InitializeUserDebug(void);
SIMAPI_TerminateUserDebug(void);

```

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 debugging capabilities to prepare the debugging environment for subsequent debug simulations.

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);

```

You first call the ***Dimension** access functions to get the dimensions of signals in the model. You can call the ***Dimension** functions anywhere; use the **INIT** section for greater efficiency because the **STATE** and **OUTPUT** sections of the **UserCode** block are called many times throughout the simulation.

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);

```

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; use the INIT section for efficiency.



NOTE: 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.

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);
```

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.

The **GetStateDerivativeValue()** function should be called in the **STATE** section of the UserCode block to access the state derivatives during intermediate integration algorithm evaluations.

The **GetExternalInputValue()** function can be called in the **STATE** or **OUTPUT** section of the UserCode block.

The **GetImplicitOutputValue()**, **GetExternalOutputValue()** functions are called in the **OUTPUT** section of the UserCode block.

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 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 $(n_{yimpr}) \times (n_{yimpr})$.

The dimension of the Implicit Solver Jacobian is $(n_x + n_{yimpr}) \times (n_x + n_{yimpr})$, where n_x is the number of states in the model, and n_{yimpr} 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.

The dimensions of the Jacobian matrices are as follows:

OP_Jacobian: $(n_imp_outputs) \times (n_imp_outputs)$

IS_Jacobian: $(n_states + n_imp_outputs) \times (n_states + n_imp_outputs)$

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);
```

`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 (that is, function call to the UserCode block). Based on this information, you can decide which simulation data to access.

14.8.4 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. [Example 14-6](#) leads you through it.

Example 14-6 Using the Debugging Feature

To run the UCB example:

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.
3. In the SuperBlock Editor, drag and drop a Sequencer block from the Software Constructs palette to the model. Place the Sequencer to the right of all the blocks in the SuperBlock.
4. Drag a UserCode block from the User Programmed palette, and place it to the *right* of the Sequencer block.

This ensures that the UserCode block is the last block to execute in the block update sequence.

5. 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 1 ensures that the block is executed during the STATE update pass of the simulator.

6. Set additional parameters as indicated below:
 - 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 now displays an array of eight zeros as the default values of the integer parameters.
- Set the values of the Integer Parameters as shown in the table below, where the integer parameters array is referred to as *IPAR[1:8]*). 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.

The table also shows the files that are created for the purpose of writing the relevant data.

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

7. Simulate your model.
8. Examine the data in the files listed in the table above.

14.8.5 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, 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 if you would like to write the simulation data to a file and inspect the results. For more advanced applications, you are encouraged to read and freely re-use the source code of **usrdebug.c** for writing your own UserCode block debugging applications.

15

BetterStateChart Blocks

BetterState is a product for creating statecharts whose use has been integrated into SystemBuild. You can place a statechart in a SystemBuild model by using the BetterStateChart block; this works similarly to placing a SuperBlock inside another SuperBlock, although there are a few differences. The BetterState chart can be a small part of your model, or it can be its entirety; in the latter case, you might use SystemBuild as a timer or scheduler.

This chapter presents the basics of using BetterStateChart block references in SystemBuild. For information on BetterState usage, see the *BetterState User's Guide*. For specifics about the BetterStateChart block, see online Help, which also contains simple examples using the block.

Much of this chapter consists of [Models That Use BetterStateChart Blocks](#). The final major topic covers [BetterStateChart Blocks in Simulation Models](#).

15.1 Using BetterStateChart Blocks in SystemBuild: The Basics

SuperBlocks can have one of several types, and BetterState has two different control implementations. You cannot mix these indiscriminately. The first part of this section explains what you can use together and the restrictions that exist.

The second part of this section discusses the creation of BetterStateChart references, before and after you create the BetterState chart that the block references.

15.1.1 SuperBlock Types and BetterState Control Implementations

SuperBlocks are one of the following types:

- Continuous
- Discrete
- Trigger
- Procedural

We are including trigger and procedural SuperBlocks as discrete SuperBlocks for purposes of this discussion; within this chapter, use of the term *discrete SuperBlock* means discrete, trigger, or procedural SuperBlock.

BetterState charts have one of the following control implementations:

- Event-driven
- Procedural

You must use these together as follows, including the restrictions listed:

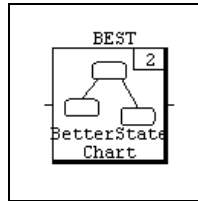
SuperBlock Type	BetterState Control Implementation	Restrictions
Continuous	Event-Driven	The BetterStateChart block must be preceded by a ZeroCrossing block. All chart transitions must be event triggered; however, they can also have conditions.
Discrete Trigger Procedural	Procedural	All chart transitions must be triggered by conditions.
Discrete Trigger Procedural	Event-driven	Event source must be a ZeroCrossing block or an external input. All chart transitions must be event triggered; however, they can also have conditions.

15.1.2 Creating References to BetterState Charts in SystemBuild Models

You can create a reference to a BetterState chart that either exists or that hasn't yet been created. This procedure assumes that you have an existing SuperBlock.

To create a reference to a BetterState chart:

1. From the SuperBlock Editor, open the Palette Browser.
2. Click the BetterState palette.
3. Drag the BetterStateChart block into the SuperBlock Editor.



4. Place the cursor over the BetterStateChart block, and press Return or Enter to open the BetterStateChart Block dialog.

If your SuperBlock is continuous, the dialog is the BetterStateEventChart Block dialog; if your SuperBlock is discrete, the dialog is the BetterStateChart Block dialog. In either case, the default name of the block is `_BEST`.

Parameter	Value	% variable
BetterState File Name		
Procedure SuperBlock Names	See Code Tab	
Number of State Events	0	
Optional List of Files		
Priority	0	

5. If the BetterState chart exists in this SystemBuild model, change the name to the name of the BetterState chart.

The appropriate properties are transferred to this dialog.

6. If the BetterState chart does *not* exist, specify the number of Inputs and Outputs at this point.

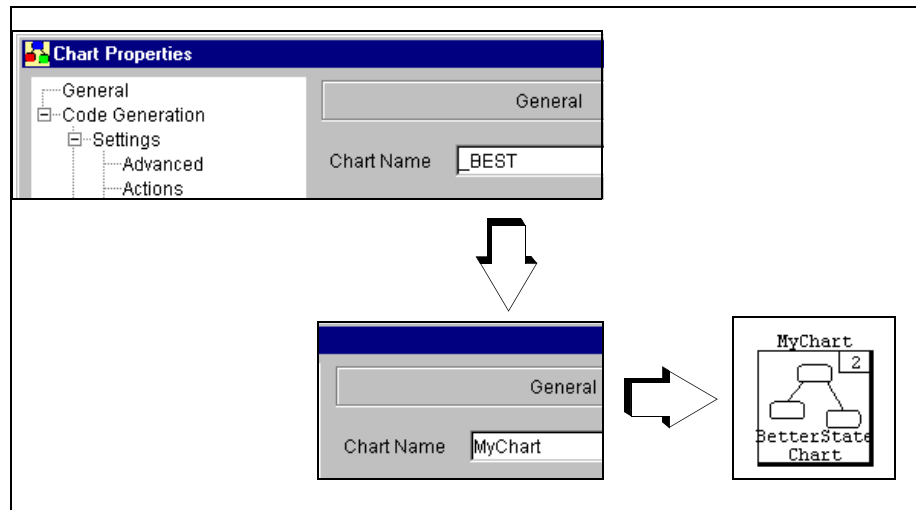
The defaults are 1 each.

7. Enter any additional BetterStateChart block properties, and click OK.

Online Help documents these fields.

Once you have a BetterStateChart block in a SuperBlock, you can double-click the block to bring up the parent chart in a BetterState Editor (Statechart window). The SuperBlock Editor that contains the BetterStateChart block remains on view.

If the parent BetterState chart doesn't exist, the Statechart window is empty, and you can create a chart. When you provide the name in the SuperBlock Editor, it is transferred to BetterState. When you leave the name `_BEST` in the SystemBuild Editor and change it in the BetterState Editor via the Chart Properties dialog, your new name is transferred back to the reference block in SystemBuild (see below).



15.1.3 Using SystemBuild Variables in BetterState Charts

You can declare SystemBuild global variables in ReadVariable and WriteVariable blocks. You can use them in BetterState charts by importing them into a chart (Edit→Import from the Data Dictionary). When you use this menu selection, the Import Global Variables dialog appears. It contains all the available global variables. Once you select the ones you want to use, they then appear in the Data Dictionary under the Locals/Globals tab with a scope of Global Extern. Conversely, you can first define them in the Data Dictionary as Global Externs, but you must then define them in SystemBuild. Once they are declared in the Data Dictionary, they are available for use in the chart where they are declared.

15.1.4 Using Procedure SuperBlocks in BetterState Charts

BetterState charts have actions associated with states and transitions. You indicate, via the appropriate dialogs, what actions to take either with user code or by calling a procedure SuperBlock. These actions are the only way that a user can incorporate SystemBuild SuperBlocks in a BetterState chart.

There are four types of actions:

- State on-entry actions
- State during actions
- State on-exit actions
- Transition actions

Using a procedure SuperBlock works the same way for each of them. Therefore, we document them once.

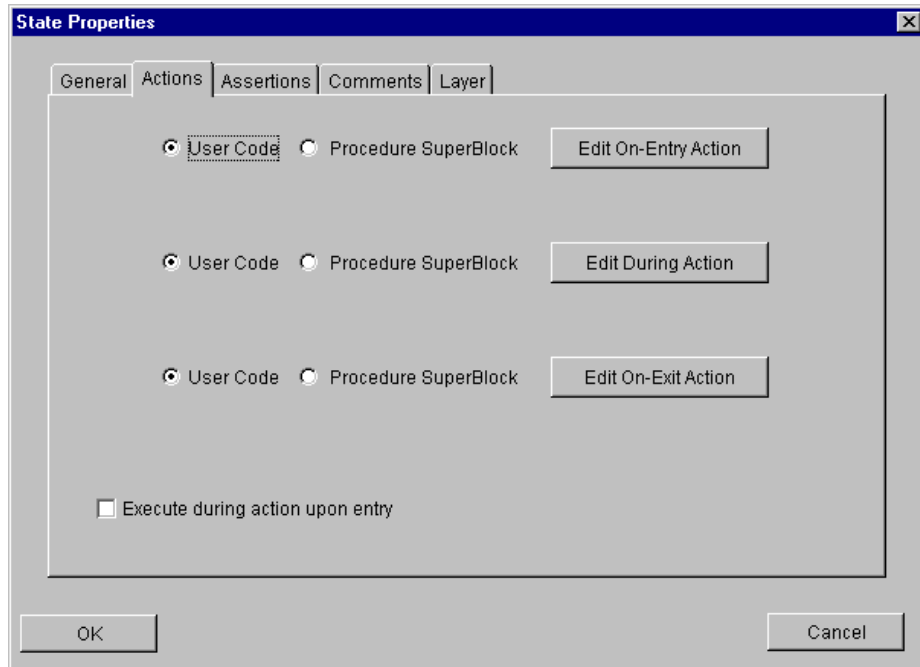


NOTE: You can find information not covered below, such as details about the user interface and various fields, in the *BetterState User's Guide*.

Using Procedure SuperBlocks as Actions

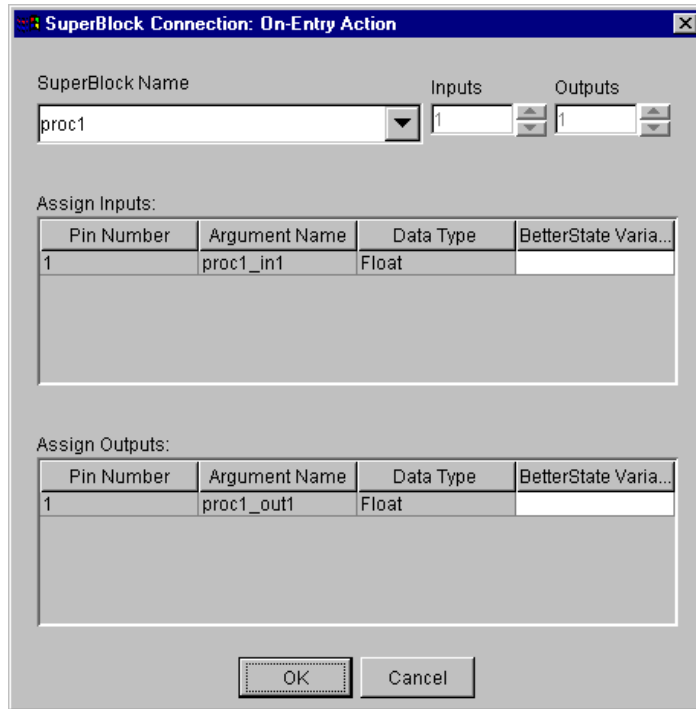
To use a procedure SuperBlock as an action in a BetterState chart:

1. From the BetterState Editor, double-click a state or transition.
The State or Transition Properties dialog comes on view.
2. Click the Action(s) tab.



3. Click the Procedure SuperBlock radio button.
4. Click the Edit *action_type* Action button, where *action_type* is one of the actions listed above.

The SuperBlock Connection dialog for the specified action appears. Procedure SuperBlocks available in the current SystemBuild model appear in the drop-down menu for the SuperBlock Name.



5. Select an existing procedure SuperBlock in the SuperBlock Name field, or define your own SuperBlock.

If the SuperBlock exists, its parameters appear in the dialog, and they are read-only.

If you define a new SuperBlock, after typing the name, press Return or Enter. Then the other fields in the dialog become active, and you can define them.



NOTE: Define the BetterState variables for both input and output in this dialog.

In both cases, a procedure icon with the name of the procedure SuperBlock appears in your diagram as long as you have the visual properties set up to show the action.

Navigating to Procedure SuperBlocks from BetterState

You can navigate to any procedure SuperBlock used in a SystemBuild model through the Catalog Browser. Moreover, BetterState provides a special navigation feature that allows you to open a procedure SuperBlock in a SuperBlock Editor from BetterState. If you define a procedure SuperBlock from BetterState, this feature transfers the properties that you define from BetterState to the SuperBlock dialog; otherwise, the properties are not transferred to SystemBuild. (If you define the SuperBlock from SystemBuild before you use the navigate feature from BetterState, the properties that you define in SystemBuild take precedence.) Once the SuperBlock exists in SystemBuild, the property fields in the SuperBlock Connection dialog become read only.

To navigate to a procedure SuperBlock from BetterState:

1. Select the state or the transition whose action references the procedure SuperBlock that you want to open in an editor.
2. Right-click to bring up the Shortcut menu.
3. Select `Navigate→procedure_SuperBlock (action_type Action)`, where *procedure_SuperBlock* is the name of the desired SuperBlock and *action_type* is On-Entry, During, or On-Exit for states or Transition for transitions.

The procedure SuperBlock comes on view in a SuperBlock Editor. Using this methodology transfers any properties that you defined in the Connection dialog if this is the first time that you are viewing this procedure SuperBlock in a SuperBlock Editor. The Statechart window remains on view.

15.2 Models That Use BetterStateChart Blocks

This section describes how you might use the BetterStateChart block in real models. Here are three types of models that make good use of the BetterState block:

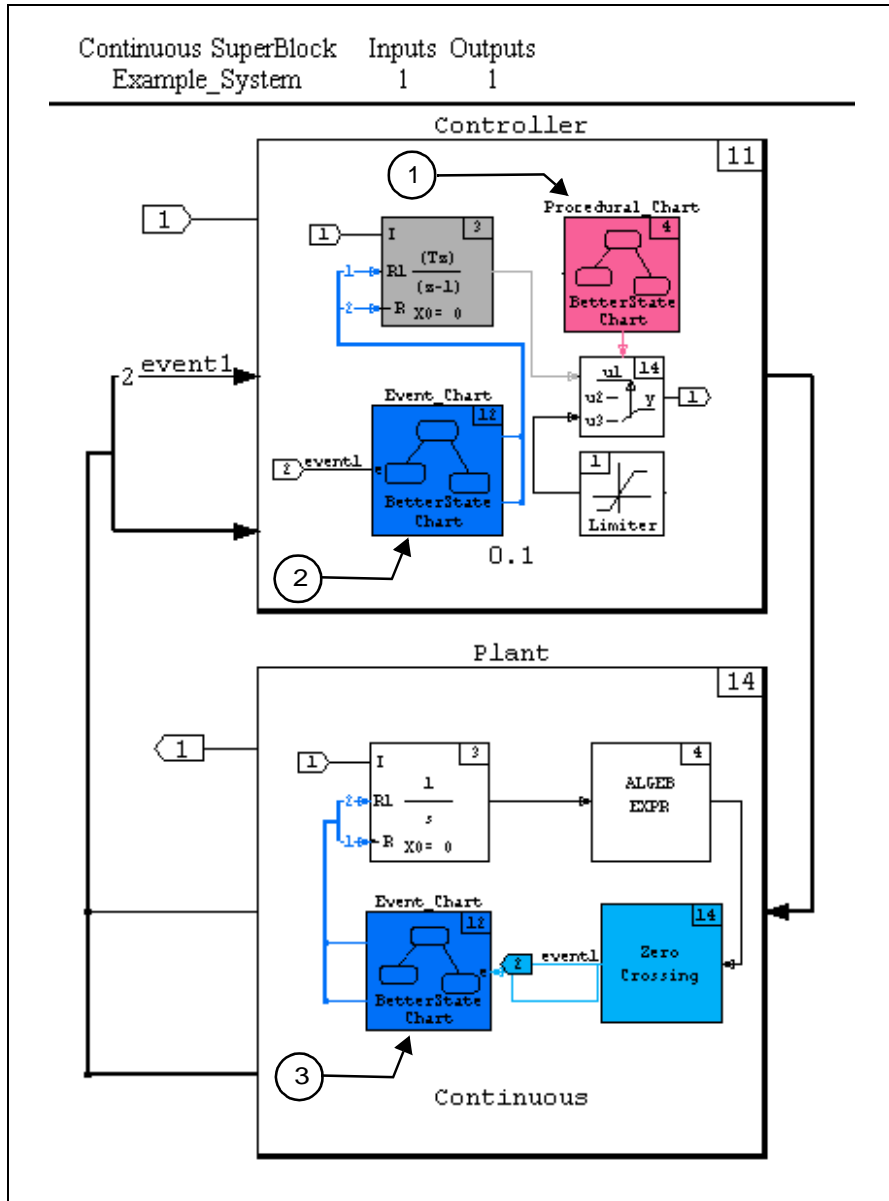
1. *Modeling Conditional Logic* (discrete SuperBlock with procedural BetterState chart)
2. *Event-Based Controller* (discrete SuperBlock with event-driven BetterState chart)

3. *Plants Whose Operating Point Changes Based on Conditions*—that is, piece-wise linear behavior (continuous SuperBlock with event-driven BetterState chart)

Figure 15-1 shows a hypothetical example of a controller and plant that contain BetterStateChart blocks. The essential blocks in this model are shown in color, and not all blocks that would be required for a real model are present. Each BetterStateChart block in this model represents one of the cases above; the callout numbers provide the correspondence. We typically simulate both the plant and controller and then generate code only for the controller. The plant is a continuous SuperBlock; note the ZeroCrossing block that precedes the event-driven BetterStateChart within the plant. The controller is represented by a discrete SuperBlock; note the procedural BetterStateChart block and the event-driven BetterStateChart block with its events supplied by an external input.

In the sections that follow, we illustrate one example of each usage of the BetterStateChart block; these examples are included in the product. We present models that illustrate points that we hope will assist you in learning how to use the tools to the best advantage as quickly as possible.

Figure 15-1 Controller and Plant with BetterStateChart Blocks



15.2.1 Modeling Conditional Logic

top is a simple model of a discrete SuperBlock with two instances of the same BetterState chart. Use the following command in the Xmath command area to load this model:

```
load file = "$SYSBLD/examples/conditionalLogic/discreteBestProc.cat"
```

Each chart has two inputs: a sine wave and a ramp (see Figure 15-2). The only difference between the two charts is that the two sine waves have different phase angles.

Figure 15-2 Discrete SuperBlock top with Two Instances of BetterState chart1

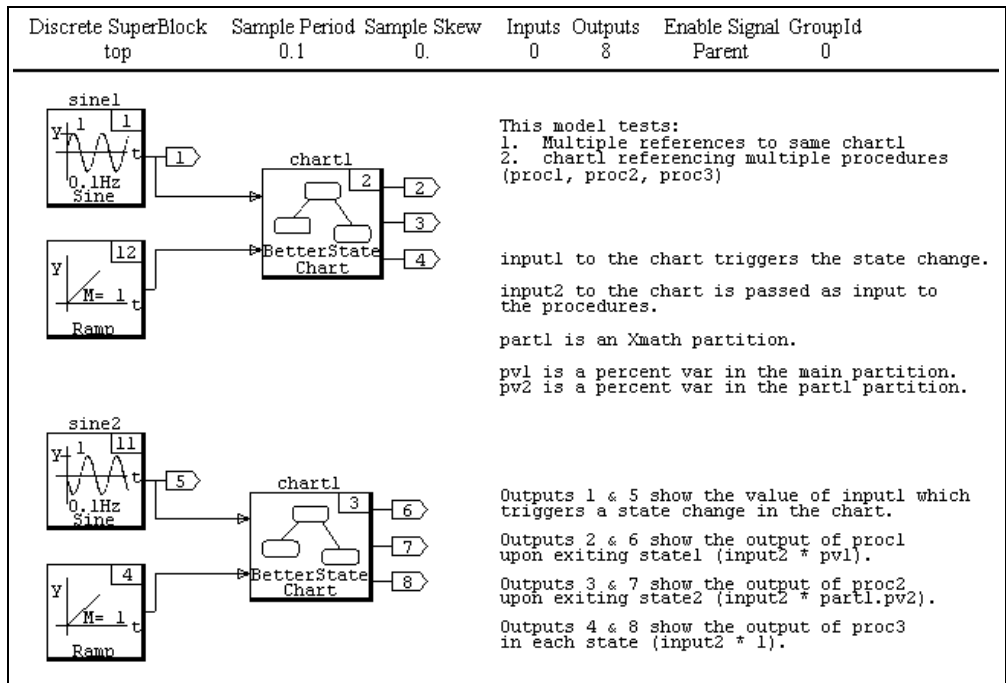
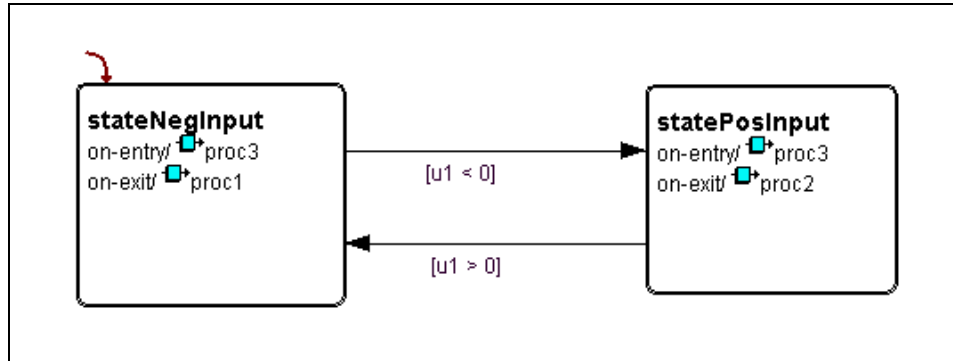


Figure 15-3 shows BetterState chart1.

Figure 15-3 BetterState chart1 for SuperBlock top



Looking at the Data Dictionary (File→Data Dictionary), you can see that the input arguments to chart1 are **u1** and **u2**, which correspond to the sine wave and the ramp, respectively. The conditions on the transitions cause the chart to change states when the sine wave changes sign.

Notice that the actions are calls to procedure SuperBlocks (see [Using Procedure SuperBlocks in BetterState Charts](#)). Each of these procedure SuperBlocks is a simple gain block, but each has a different gain; these are set up in SystemBuild as %variables so that you can change the gain if you wish. The input of each procedure SuperBlock is **u2**, or the ramp, and the output of each gain block becomes an output to the chart and the SuperBlock.

The chart changes state every time that the sine wave changes sign, and **proc3** is the on-entry action for each state. The ramp is multiplied by the gain of 2 each time. The output of **proc3** is the third output of the chart.

Similarly, the on-exit actions are called when the states change:

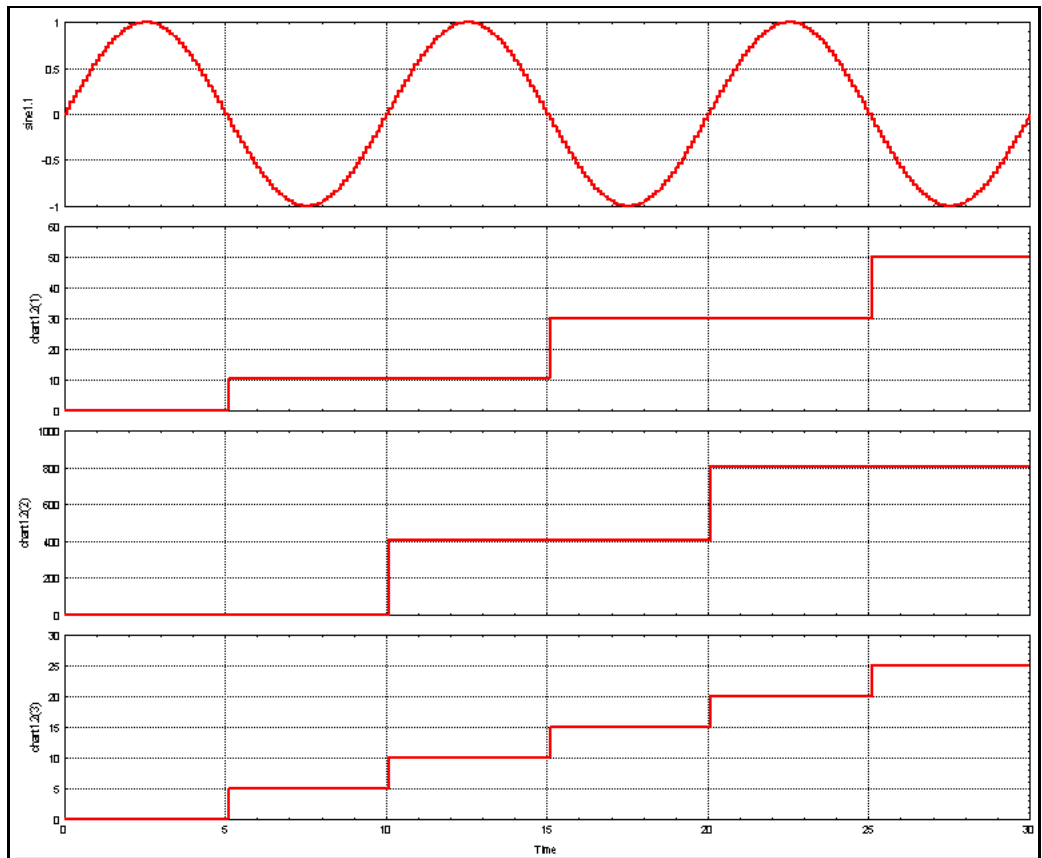
- When the sine wave turns negative, **proc1** multiplies the ramp by a gain of 40; this is the first chart output.
- When the sine wave turns positive, **proc2** multiplies the ramp by a gain of 1; this is the second chart output.

To simulate the model:

1. Double-click top in the Contents pane of the Catalog Browser to bring up the SuperBlock in a SuperBlock window.
2. From the SuperBlock Editor window that contains top, select Tools→Simulate. The SystemBuild Simulation Parameters dialog appears.
3. Enter [0:0.01:30]' into the Time Vector/Variable, and enable Plot Outputs. Click OK.

The first four outputs for the model appear in [Figure 15-4](#).

Figure 15-4 **Partial Plot Output for SuperBlock Model top**

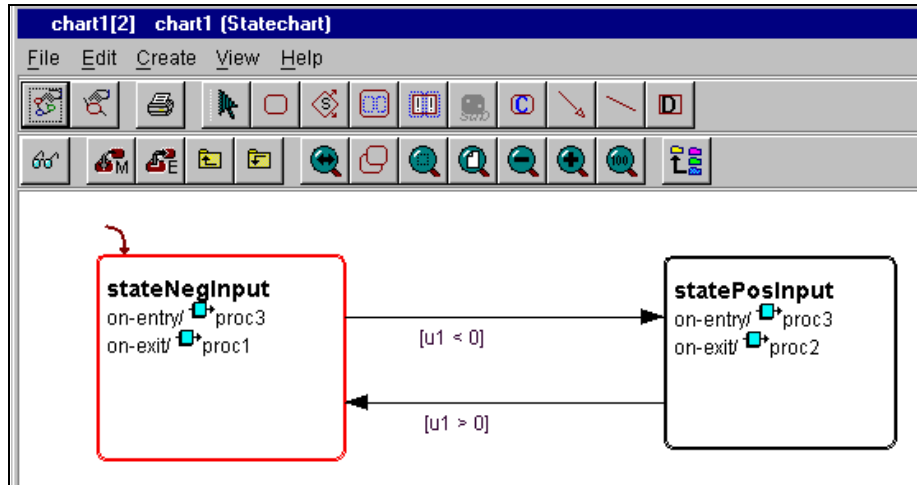



Your plot will show all eight outputs; the second four are out of phase with the first four.


To run chart animation during interactive simulation:

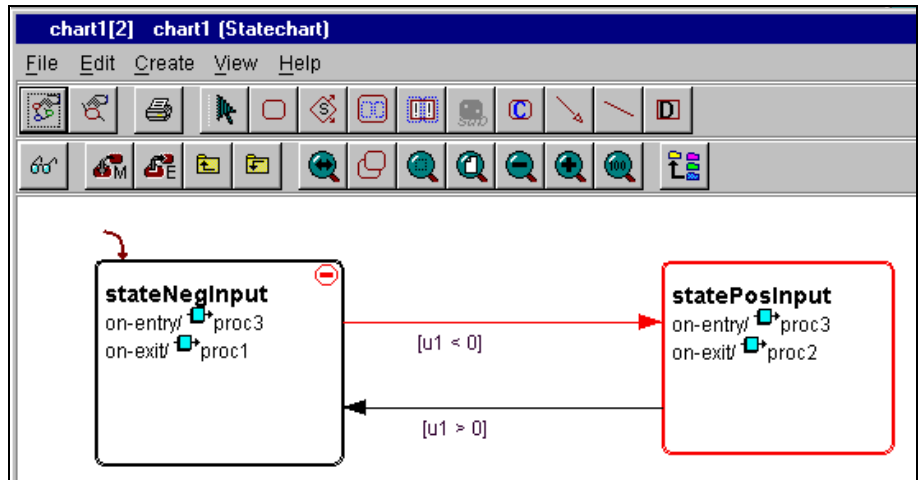
1. From the SuperBlock Editor window that contains top, select Tools→Simulate. The SystemBuild Simulation Parameters dialog appears.
2. Enter [0:0.01:300]' into the Time Vector/Variable, and enable Interactive. Click OK. The Interactive Simulator window comes on view.
3. From the Interactive Simulator window, double-click the first instance of chart1 (ID 2).

Another BetterState Editor window comes on view. It contains chart1 with stateNegInput shown in red, which indicates that it is the current (initial) state.



4. If your monitor is large enough, arrange your windows so that at least the left portion of the Interactive Simulator window is showing and the full BetterState Editor with the chart animation is showing.
5. From the Interactive Simulator window, click the Simulate button to start the simulation. 

You can watch the simulation move between the states. Note that the last state visited is indicated by the circled bar symbol ().



For additional information on chart animation, see the *BetterState User's Guide*.

15.2.2 Event-Based Controller

To illustrate an event-based controller, we have a simple apple vending machine where you can deposit nickels, dimes, and quarters and request coin return or dispense an apple via icons. The signals generated by user actions are set up to pass through zero; passing these signals through a ZeroCrossing block then creates events for the BetterState block.

A separate SuperBlock, VendingMachineController, isolates the vending machine controller, which provides more flexibility for simulating the model and generating code for the controller.


The VendingMachineLogic BetterState chart controls the vending machine logic itself. The chart contains one state with three threads of control: MoneyCollector, CoinReturn, and Dispenser. These threads all behave independently of each other. Each has an idle state that transitions to a non-resting state when a particular event occurs; each event corresponds to the clicking of an icon in the SystemBuild model. After setting the appropriate variable(s), each thread returns to its idle resting state to await another event. The Dispenser thread dispenses an apple and any change that is due.

To load and simulate this model:

1. Input the following command into the Xmath command area:

```
exec file = "$SYSBLD/examples/VendingMachine/VendingMachine.ms"
```

When the model is loaded, the VendingMachine SuperBlock appears in the Interactive Simulator window (see [Figure 15-5](#)).

2. Make the Interactive Simulator window the active window as necessary.
3. Click the Simulate button to start the simulation. 
4. Click the IA icons in the red area to simulate the vending machine.

The clock rotates twice for the simulation time. By forcing the clock to update regularly, the simulation is slowed down enough to provide interaction time for users who have fast computers.

5. When the simulation has completed, close the Interactive Simulator window.

The plot output appears; see [Figure 15-6](#) for a sample set of plots.

Figure 15-5 VendingMachine SuperBlock

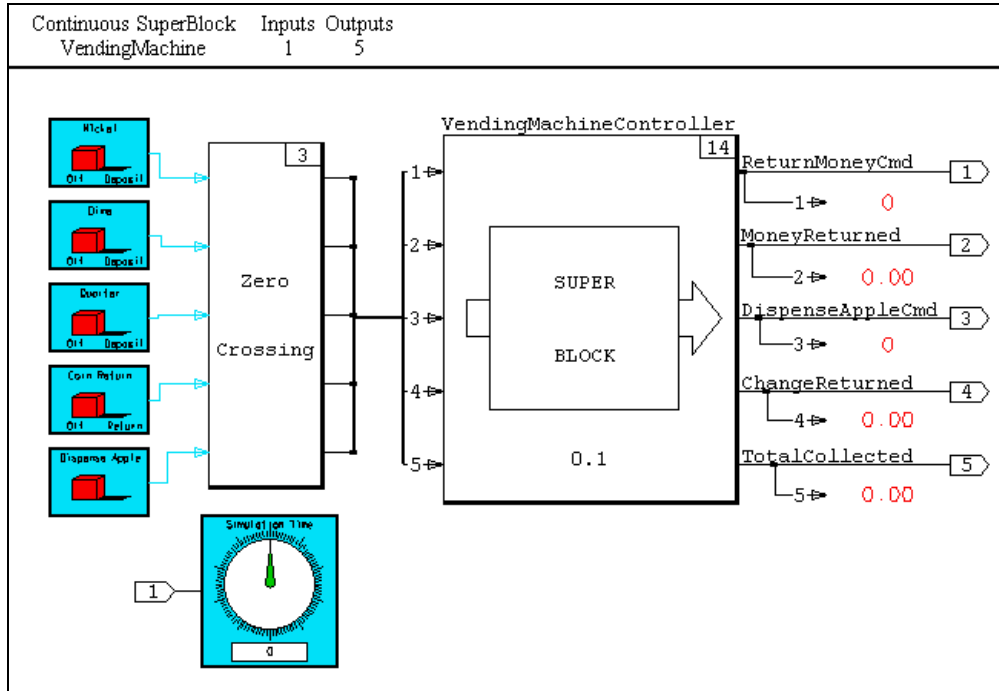
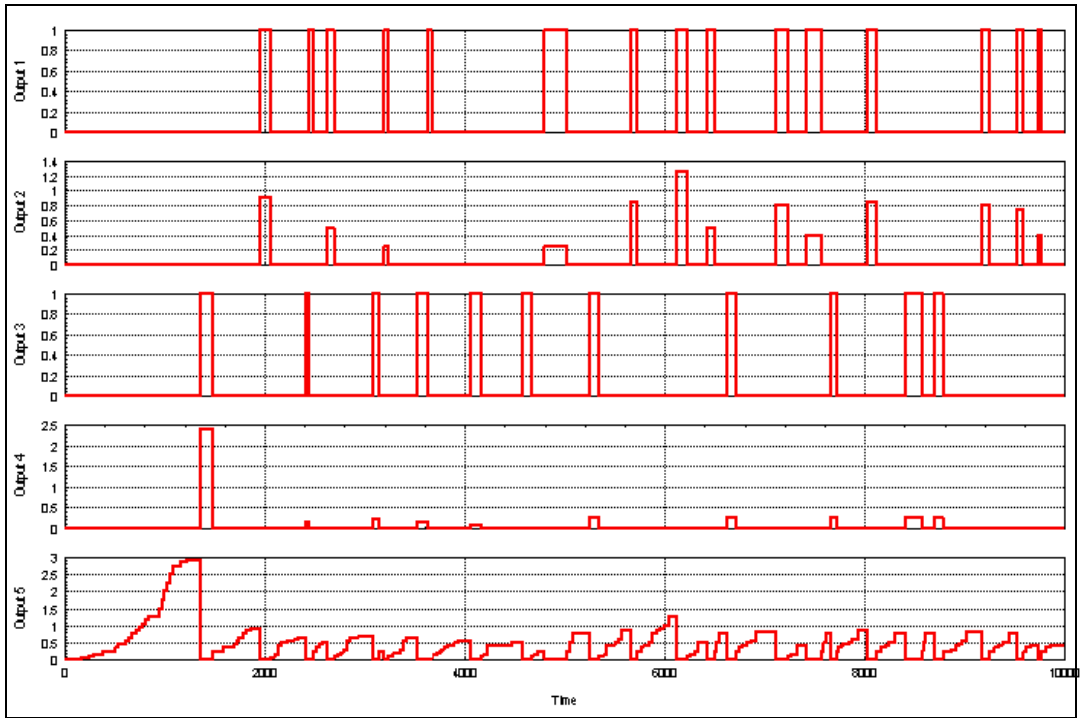
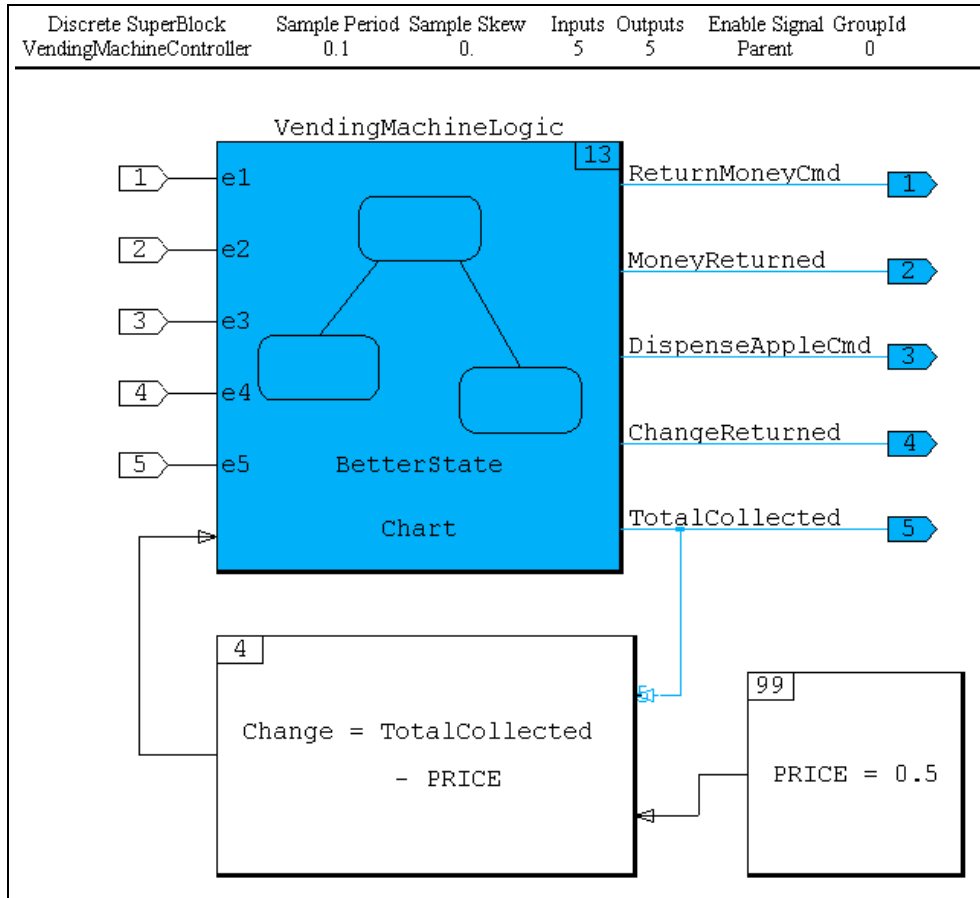


Figure 15-6 VendingMachine Sample Plot Output



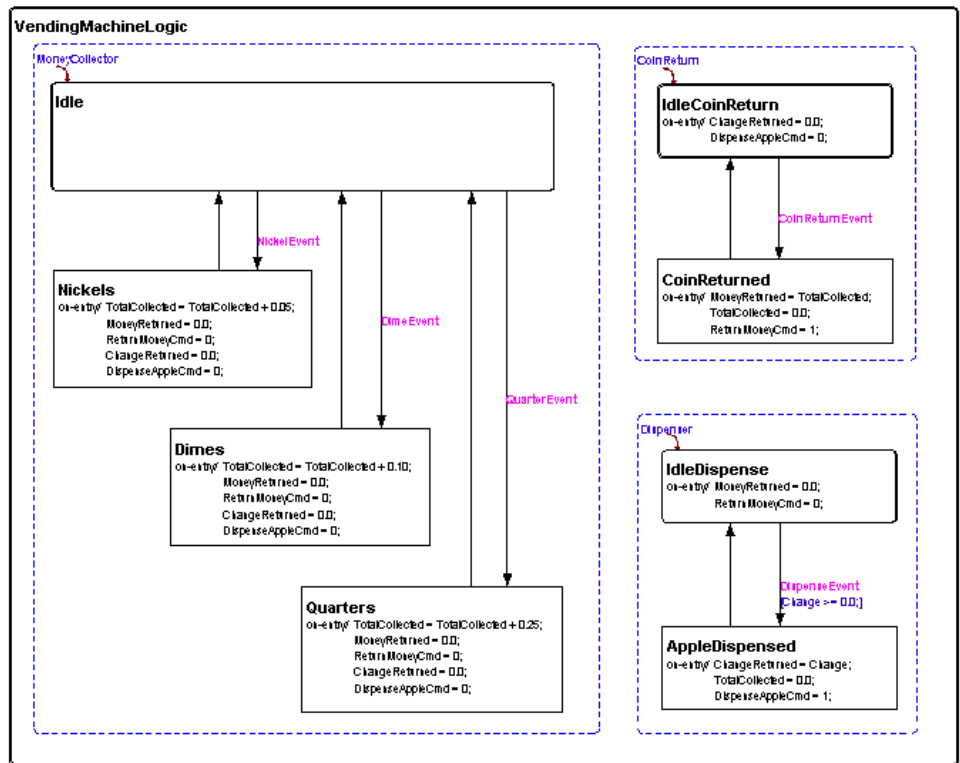
From the Catalog Browser, double-click the VendingMachineController to see it in an editor window (see Figure 15-7).

Figure 15-7 **VendingMachineController SuperBlock**



From the SuperBlock Editor or the Catalog Browser, double-click the VendingMachineLogic BetterStateChart block to bring it up in a BetterState Editor (see Figure 15-8).

Figure 15-8 VendingMachineLogic BetterState Chart



You can find another example of event-based controllers in *SYSBLD/examples/cSB_eBest.cat*. This model is an open-loop system that converts a sine wave into a square wave. See online Help for the BetterStateChart block.

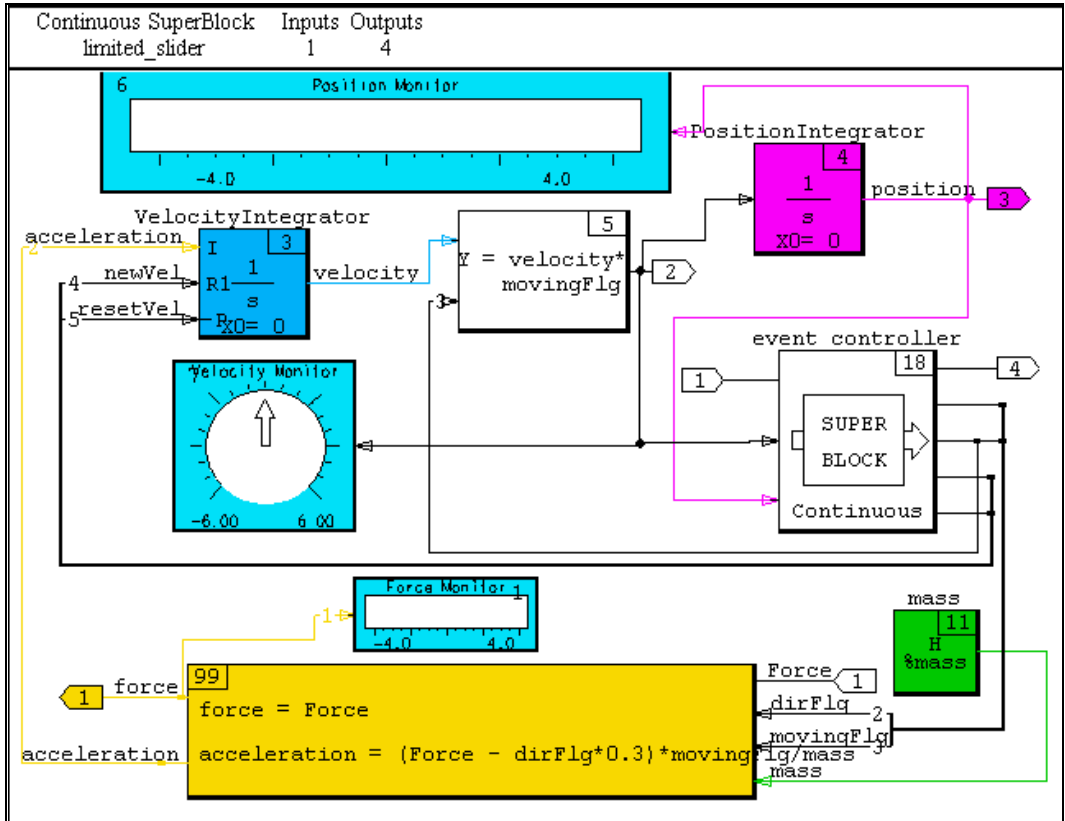
15.2.3 Plants Whose Operating Point Changes Based on Conditions

The final example is a real model, which is somewhat more complex than the other examples presented in this chapter. You can run the limited slider as a demo; see [Running SystemBuild Demos](#) for instructions; the Limited Slider demo provides a selection of different forces.

The model is a mass, located between two walls. It starts in the middle and can move in either direction. You can choose the force—or see the model work under a variety of different forces; the mass is restrained by friction, which is modeled as a static friction constant and a dynamic friction constant. A coefficient of restitution represents the velocity loss when the mass collides with a wall.

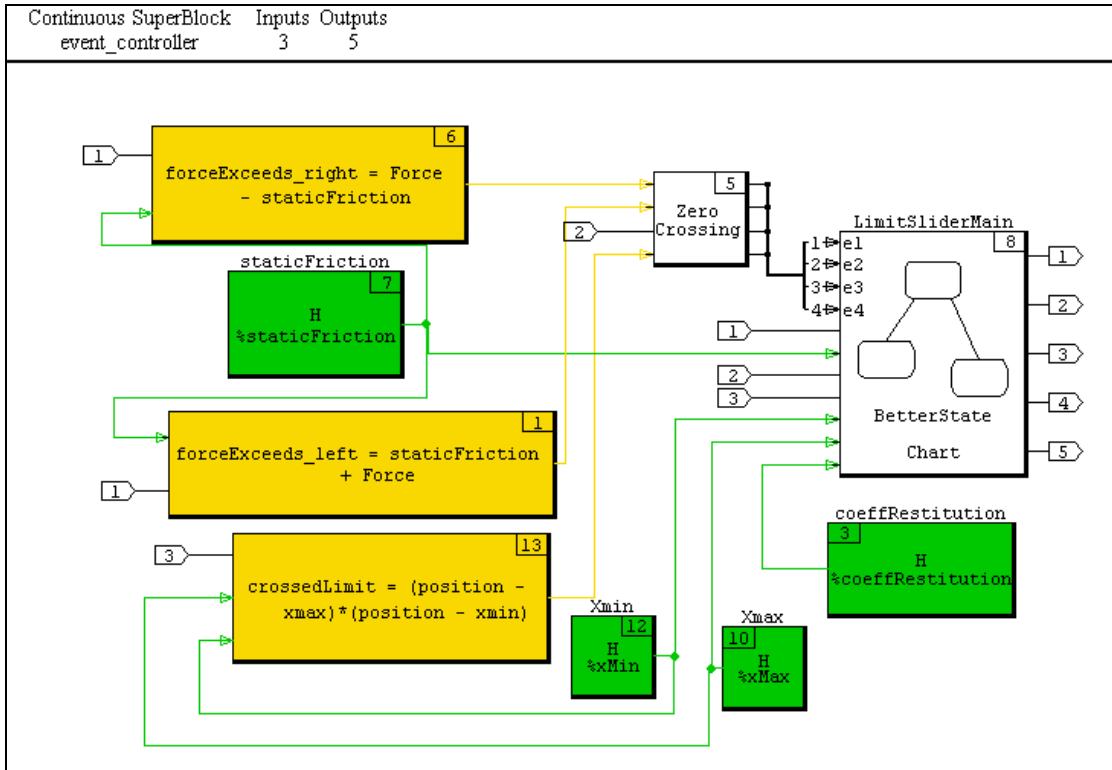
The highest level SuperBlock is `limited_slider`, a continuous SuperBlock that contains the equation to compute acceleration, followed by a resettable integrator to compute velocity and another integrator to compute position (see [Figure 15-9](#)).

Figure 15-9 `limited_slider` SuperBlock



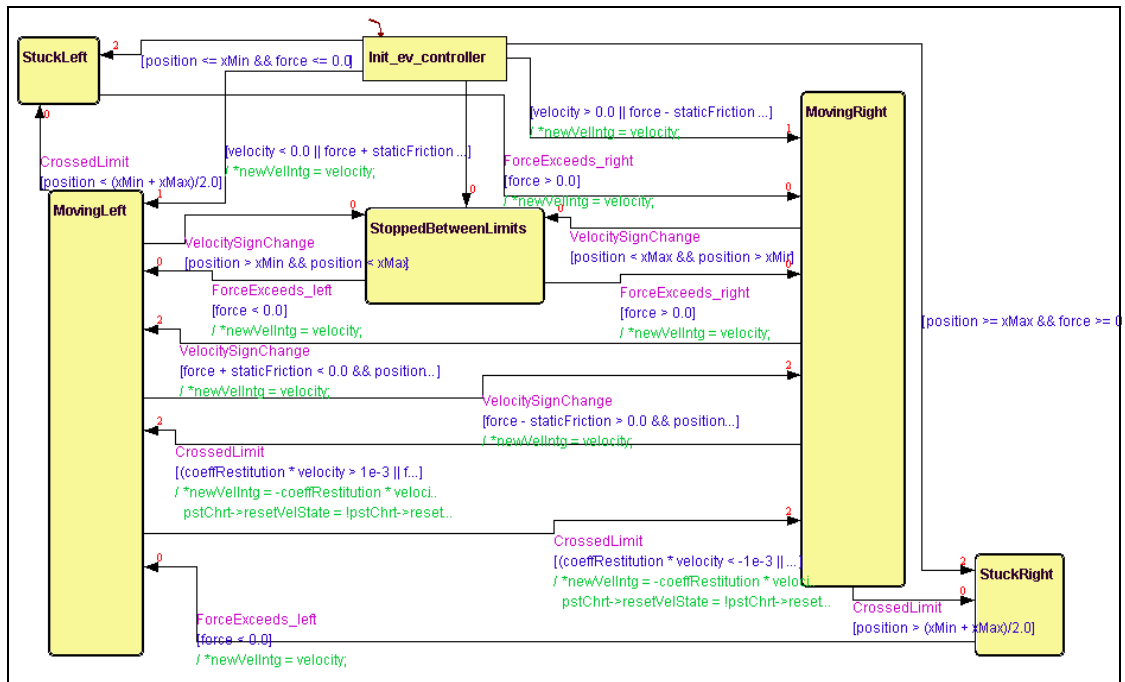
The force, velocity, and position are inputs to `event_controller`, another continuous SuperBlock. The event controller consists of a set of equations that are passed through a ZeroCrossing block to create events when their sign changes; note that passing multiple inputs into one ZeroCrossing block creates the same behavior as passing each of those inputs into a separate ZeroCrossing block. In turn, these events drive the BetterStateChart block that follows the ZeroCrossing block (see [Figure 15-10](#)).

Figure 15-10 event_controller SuperBlock



The BetterState chart consists of one non-resting state, which allows the controller to initialize itself and determine its first resting state based on the transition conditions. Then it moves among its five states—StuckLeft, MovingLeft, StoppedBetweenLimits, MovingRight, and StuckRight. The movements are initiated by the events generated in event_controller, but they also require the satisfaction of conditions on the transitions. The user code is written in C for this example. This statechart appears in Figure 15-11.

Figure 15-11 LimitSliderMain StateChart



Even though the logic in the BetterState chart is somewhat complex, you might find the model easier to construct using this method than using UserCode blocks and writing all the code manually.

For another example of a plant changing its behavior based on an operating point, see the *MATRIX_X* Getting Started for your platform.

15.3 BetterStateChart Blocks in Simulation Models

As we have discussed in [Using BetterStateChart Blocks in SystemBuild: The Basics](#), BetterState charts can have one of two control implementations: event-driven or procedural. Each has restrictions on how you use it. In this topic, we discuss how these different control implementations of the block are handled in simulation models.



WARNING: If you are using BetterState charts created in BetterState 5.2 or before, any variables declared in variables boxes *must* be moved to the Data Dictionary if you plan to use the resume feature of simulation.

A procedural BetterStateChart block represents a *block* in SystemBuild. They are sorted by the analyzer and executed as blocks.

An event-driven BetterStateChart block represents a separate *subsystem* in SystemBuild (see [Dividing Your Model into Subsystems](#) on p.168). It is executed immediately after the continuous subsystem and has a user-selected execution priority. Event signals must come from ZeroCrossing blocks for simulation.



CAUTION: Any time you simulate or generate code for a model that contains BetterStateChart blocks, that model should exist in a separate directory. Otherwise, you risk mixing objects between models because there is only one **simucb** shared library per working directory.

15.4 BetterStateChart Blocks in Models for Which You Use AutoCode

Just as for simulation, there are restrictions for models for which you generate AutoCode based on the control implementation.

Procedural blocks work the same for simulation and AutoCode.

An event-driven BetterStateChart block represents a separate *subsystem* in SystemBuild. Event signals must be external I/O for the code generator. You use the AutoCode template to connect an interrupt service routine (ISR) to each unique event. The ISR schedules each BetterState event-driven chart subsystem

for its event. The order of execution within the ISR for the subsystem is controlled by the execute Priority field.

For more information on AutoCode, see the *AutoCode User's Guide*.

16

Fixed-Point Arithmetic

This chapter describes SystemBuild fixed-point arithmetic. This feature emulates two's complement binary arithmetic, allowing systems running code generated by AutoCode (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 data type.
- Questions of precision and significance arise because of the fixed word sizes and ranges of the data types.

The SystemBuild user interface provides several access points to fixed-point information:

- The SystemBuild Connection Editor reports on the data types of fixed-point signals.
- The Inputs tab of the SuperBlock Properties dialog allows you to specify fixed-point input data types.
- The Outputs tab of the SuperBlock Properties dialog allows you to specify fixed-point output data types.
- The Gain and ScalarGain block dialogs allow you to specify a radix position for the gain parameter.
- User-defined data types (also called UserTypes) provide an aliasing capability for SystemBuild data types. It is possible to assign a name that is relevant to your problem to a data type. Although UserType support is a general SystemBuild feature, it is particularly convenient for fixed-point arithmetic, where you may need to switch data types after each simulation. For more information on UserTypes, see [16.5 User-Defined Data Types \(UserTypes\)](#), p.442 and, for a full treatment of data types, see [5.5.5 Specifying Data Types](#), p.103.
- For simulation or code generation, the SystemBuild Analyzer performs all necessary data type checking, based on data type checking rules tabularized in [16.3 Fixed-point Blocks and I/O Data Type Rules](#), p.424. For operation of the Analyzer, see [8.4.2 Using the analyze\(\) Function](#), p.185.
- The block set for use with fixed-point numbers includes arithmetic, logical, and piecewise linear blocks. For the complete list see [Table 16-1](#), p.424.

AutoCode provides full support for fixed-point arithmetic in generated code. See "Fixed-point Arithmetic" in the *AutoCode User's Guide*.

16.1 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 remainder of this chapter discusses the issues involved in simulating models and generating fixed-point C code from a SystemBuild model with integer data types and scaling and the options available to produce fixed-point code.

16.1.1 Fixed-point Number Representation

Fixed-point arithmetic uses integer data types, 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 *word size*; thus the possible word sizes 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 word size 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 16-1 shows an 8-bit unsigned fixed-point number with radix position of 4, meaning that four bits are for the fractional part and four bits are 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 16-1 shows an 8-bit signed fixed-point number with radix position of 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 $[-2, 1 \frac{63}{64}]$.

Figure 16-1 **A Signed Fixed-point Number with 8 Bits and Radix Position of 6**

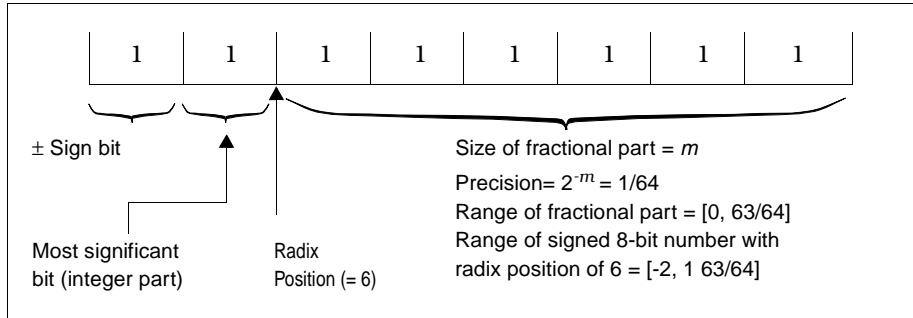
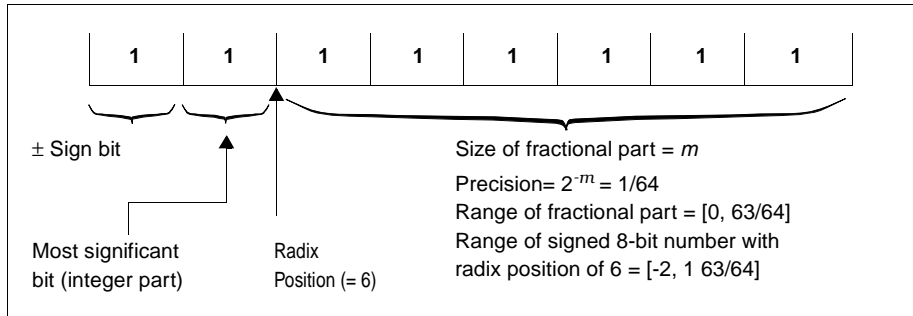


Figure 16-2 **A Signed Fixed-point Number with 8 Bits and Radix Position of 6**



When the radix position is zero, note that the example in Figure 16-1 is an unsigned integer with range $[0, 255]$, and the example in Figure 16-1 is a signed integer with range $[-128, 127]$.

The *scaling* referred to in fixed-point arithmetic is directly related to the radix position, m , of the fixed-point data types. The scale factor is 2^m . As an example, the number 1.25 is stored as 0000101 in an unsigned 8-bit fixed-point data type 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 word size limitations not a constraint, the following rules would provide the result radix position, word length, sign (that is, the result data type), 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 word size 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 can result in an overflow¹ if it causes a significant bit to “fall off” the left (most significant) side of the word boundary. A similar difficulty can 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 word size.

Moreover, you frequently determine a desired data type 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), which would impact the radix position and alignment rules discussed above.

For maximum flexibility, every block that supports fixed-point arithmetic allows you to specify an output data type. In most cases, such output data type stipulation resolves operation data type issues appropriately. Nevertheless, there are cases where more data type 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 is represented as an ordered pair (i, rn) , where i

1. Overflow is defined as loss of significance; that is, losing bits in the integer part of the number. The term underflow is used to mean overflow on a negative number.

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 ($i, rn, sign$).

16.1.2 Conversion Between Fixed-point Numbers

You can convert one fixed-point data type to another using a signal TypeConversion block. Algorithmically, if the word size is not changed, this operation amounts to a change in the radix position.

In software generated by AutoCode, 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 2 by shifting the integer one position to the left. Note that this operation overflows if the result cannot fit in the number of bits available. When this happens, a significant bit “falls off” the left side of the number. (A significant bit is 1 for an unsigned or signed positive and 0 for a signed negative number.)

Similarly, to reduce the radix position of a fixed-point value by one the integer is divided by 2 using a bitwise right shift. In this instance, loss of precision occurs 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 yields 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 16-1](#) shows conversion between fixed point numbers of different radix position.

Example 16-1 Conversion of fixed-point numbers

0010[^]0101 (n1 = (37, r4), decimal value: 2.3125)

([^] indicates the imaginary position of the radix position within the binary data)

Align to a radix position of 6 (that is, shift left n1 by 2):

10[^]010100 (n2 =(148,r6), decimal value: 2.3125)

16.1.3 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 16-2](#) adds the fixed-point number $n1 = (37, r4)$ and fixed-point number $n2 = (65, r1)$ to produce the fixed-point result number $n3$ with radix position 2 using 8-bit unsigned variables.

Example 16-2 Addition of Fixed Point Numbers

In binary representation:

```
0010^0101 (n1 = (37, r4), decimal value: 2.3125)
```

+

```
0100000^1 (n2 = (65, r1), decimal value: 32.5)
```

Align the radix positions of $n1$ and $n2$ to the radix position of the result before adding (that is, shift $n1$ right by 2 bits, and shift $n2$ left by 1 bit). Place the aligned results in $n1'$ and $n2'$; then perform the addition:

```
000010^01 (n1' = (9, r2), decimal value: 2.25)*
```

+

```
100000^10 (n2' = (130, r2), decimal value: 32.50)
```

```
100010^11 (n3 = (139, r2), decimal value: 34.75)
```

* Loss of precision occurred while shifting $n1$ to $n1'$.

16.1.4 Multiplication

Multiplication of fixed-point numbers does not require pre-alignment of radix positions. The radix position of the product is the sum of the radix positions of the operands, and the stored value of the result is the product of the stored values of the operands. The length of the result of a multiplication can be as great as the sum of the lengths of the operands. Because the result of a multiplication may not fit into a variable of the same length as the operands, widening multiplication is performed in SystemBuild and AutoCode for the multiplication operation to produce an internal result that holds the full product. For example, if two 8-bit variables are multiplied, the internal result is a 16-bit variable; if two 16-bit variables are multiplied, the internal result is a 32-bit variable. This internal result is then converted to the desired fixed-point data type with possible loss of precision (clipping of the least significant bits), and/or loss of significance

(clipping of the most significant bits). Use of an extended internal data type avoids overflow of the multiplication and allows overflows to be corrected before the output data type is applied. Because 64-bit data types are not supported in C or Ada, extended internal results are stored using two 32-bit data types.

[Example 16-3](#) 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 a 16-bit intermediate result.

Example 16-3 **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 (that is, shift $n3'$ right by 3 bits). Place the aligned result in $n3$:

```
1001011^00 (n3 = (300, r2), decimal value: 75.0)
```

16.1.5 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 word length 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 16-4](#) illustrates division of fixed-point number $n1=(128, r4)$ by fixed-point number $n2=(224, r5)$ to produce the fixed-point result number with a radix position of 7 using 8-bit unsigned variables and a 16-bit internal variable.

Example 16-4 **Division of Fixed Point Numbers of Different Radix Positions**

1000^0000 (n1 = (128, r4), decimal value: 8.0)

Left shift the 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 the result in an 8-bit unsigned variable *n3*:

1^0010010 (n3 = (146, r7), decimal value: 1.140625)

16.1.6 Relational Operations

For comparing two fixed-point numbers of the same word size, the number with the smaller radix position (that is, 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 1 indicating that the comparison is TRUE and 0 indicating that it is FALSE. [Example 16-5](#) shows greater-than comparison of fixed-point number *n1* = (25, r3) and fixed-point number *n2* = (17, r1) using 8-bit unsigned variables.

Example 16-5 **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 (that is, shift *n2* left by 2 bits). Place the aligned results in *n2'*:

00011^001 (n1 = (25, r3), decimal value: 3.125)

>

10000^100 (n2' = (136, r3), decimal value: 8.5)

FALSE

16.1.7 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 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 Outputs tab of the block's dialog, 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 16-6](#) shows overflow in the context of conversion of fixed-point number (32, r4) to a fixed-point number with radix position of 7.

Example 16-6 **Conversion of Fixed-point Number with Overflow Protection**

```
0010^0000 (n1 = (32, r4), decimal value: 2.0)
```

Align to a radix position of 7 (that is, shift left *n1* by 3):

```
0^0000000 (n2 = [0, r7], decimal value: 0.0)*
```

Correct overflow (that is, use maximum number possible):

```
1^1111111 (n2 = (255, r7), decimal value: 1.9921875)
```

* Overflow has occurred while left shifting *n1*.

Example 16-7 shows addition of the fixed-point number $n1=(249, r4)$ and the fixed-point number $n2=(7, r3)$ to produce a fixed-point result number with a radix position of 4 using 8-bit unsigned variables. Overflow protection is provided.

Example 16-7 Addition with Overflow Protection

1111^1001 ($n1 = (249, r4)$, decimal value: 15.5625)

+

00000^111 ($n2 = (7, r3)$, decimal value: 0.875)

Align the radix position of $n2$ to the radix position of the result before adding (that is, shift $n2$ left by 1 bit). Place the aligned result 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 (that is, use maximum number possible):

1111^1111 ($n3 = (255, r4)$, decimal value: 15.9375)

* Overflow has occurred while adding $n1'$ and $n2'$.

16.2 SystemBuild Fixed-point

16.2.1 User Interface

If you are the user of fixed-point math, you proceed much as you do in the traditional SystemBuild paradigm, selecting, placing, parameterizing, and connecting blocks. Particular differences are the emphasis placed on assigning data types, and its influence on the ranges and precisions of data values. Two places are provided for specifying fixed-point data:

SuperBlock Properties Dialog, Inputs Tab — The Input Data Type field is a combo box that provides a complete list of choices. Click the arrow until the vertical list

appears, and then move the mouse down to select the appropriate data type; release the mouse button. You can add the Input Radix as required.

Block Dialog — The Output Data Type is a combo box. Click the arrow until the vertical list appears, and then move the mouse down to select the appropriate data type; release the mouse button. You can add the Radix as required.

In both these dialogs, you can also type a user-defined data type or UserType. This method is described in [16.5 User-Defined Data Types \(UserTypes\)](#) on p.442.

Note that Float (real floating-point numbers) is the default setting for inputs and outputs of the blocks discussed in this chapter.

16.2.2 Simulator

For simulating from the Xmath command area, use the **fixpt** keyword to invoke fixed-point arithmetic. For simulating from the Simulation Parameters dialog, set the Sim Type field to a fixed-point data type. The **fixpt** keyword or Sim Type field is useful for comparisons and for studying the effect of quantization. Simply run a simulation with **fixpt** off, and then with it on; 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, the **fixpt_round** keyword has no effect.

Note that whenever **fixpt** is set:

- Data type checking is always on.
- External inputs are always rounded.
- Parameters are rounded, and their data types 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.

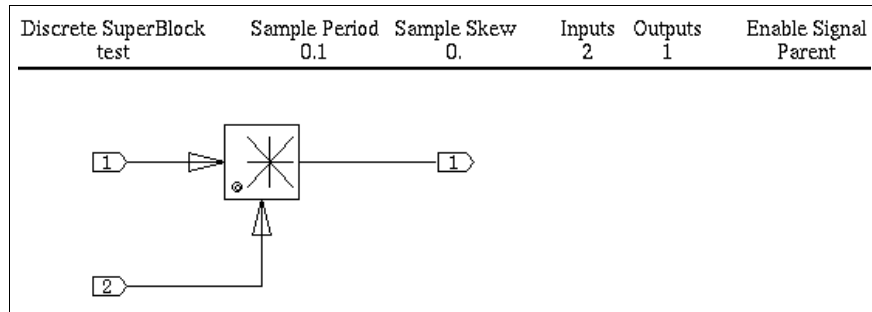
16.2.3 Building a Model and Demonstrating Overflow

When the size of a value becomes too large to fit in the data type, overflow has occurred. (Thus, in the example in [Figure 16-1](#), trying to place a value ≥ 16 in the register that holds the number causes overflow; in [Figure 16-1](#), 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 data type (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 16-8](#) shows this.

Example 16-8 Demonstrating Fixed-point Arithmetic and Overflow

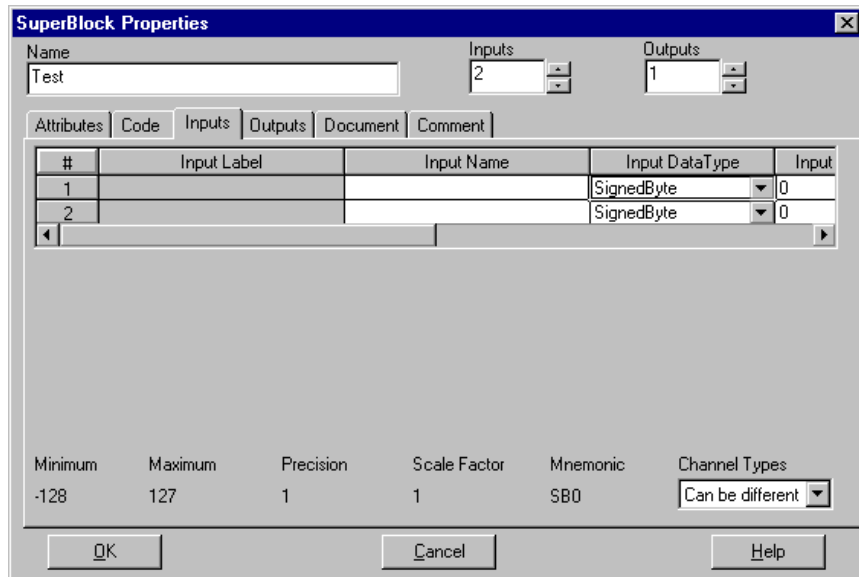
1. First create a discrete free-running SuperBlock; name it Test. Give it two inputs and one output. The exact Sampling Period does not matter; accept the default value of 0.1.
2. Inside the SuperBlock place an ElementProduct block. Connect two external inputs to the inputs of the block, and connect the output to an external output. This model is pictured below.



3. Double-click in the header area of the diagram to invoke the SuperBlock Properties dialog. On the Inputs tab, change the data type of both inputs to Signed Byte.

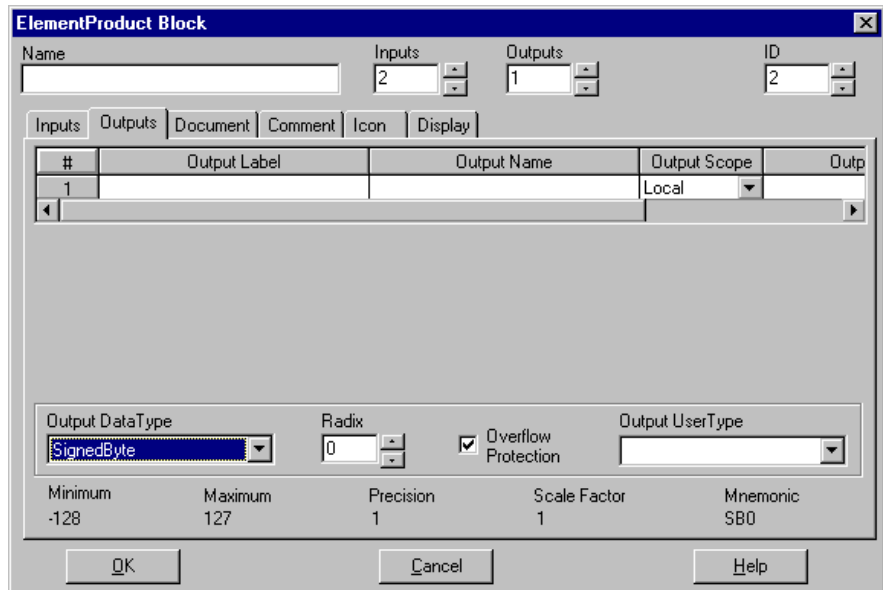
See [Figure 16-3](#). This step is required to change the default input data type (Float) to the needed fixed data type. Observe the bottom line of the dialog, where Minimum, Maximum, Precision, Scale Factor, and Mnemonic (SB0) are shown. Note that the range [minimum, maximum] is [-128, 127].

Figure 16-3 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 [Figure 16-4](#).

Figure 16-4 Outputs tab with Signed Byte Chosen

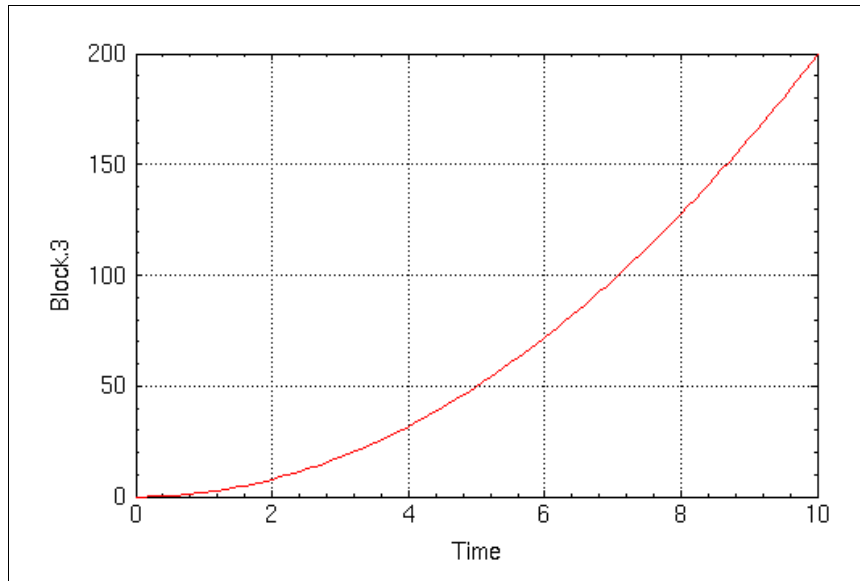


5. Try a simulation without `fixpt` asserted. From the Xmath command area, type:

```
t = [0:0.1:10]';  
u = [2*t,t];  
y = sim("Test",t,u,{graph});
```

See [Figure 16-5](#) 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 16-4](#)), that is outside the range of the output data type, SB0. However, the `fixpt` keyword is not activated, and therefore floating-point arithmetic is performed, without the absolute fixed-point limitations on the output.

Figure 16-5 **Fixed-point Plot without Overflow**

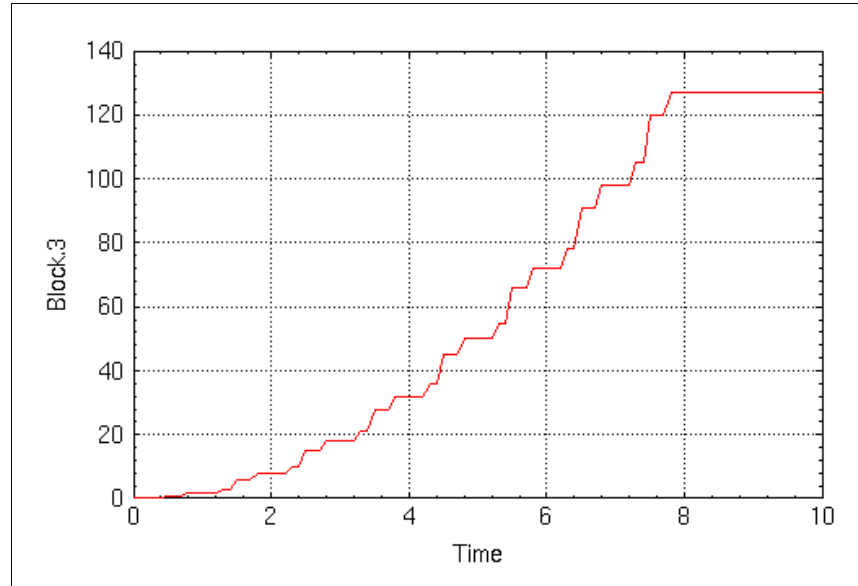


6. Try another simulation, this time with **fixpt** asserted:

```
y = sim("Test",t,u,{graph,fixpt});
```

The plot should look like [Figure 16-6](#).

Figure 16-6 **Fixed-point Plot with Overflow**



Observe what is happening here; the value increases until 127, the maximum number for the SB0 data type, 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 data type as seen in [Figure 16-4](#). 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 data type, the simulation stops returning maximal values.

16.2.4 Comparing Fixed- and Floating-Point Numbers

Example 16-9 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 and then subtract it from a floating point input of the same value.

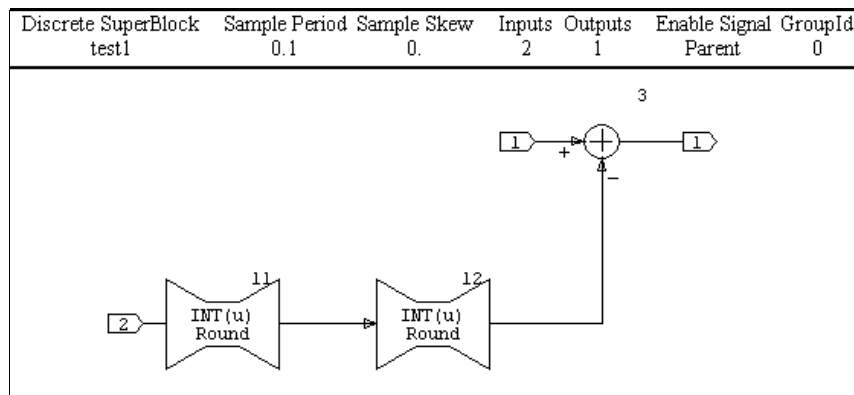
Example 16-9 Comparing Fixed- and Floating-Point Numbers

You are going to build a model that accepts 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 output is plotted, it shows the difference between the floating-point input and its fixed-point form.

To create a model to compare fixed- and floating-point numbers:

1. Create a new SuperBlock. Name it test1. Make it discrete and specify two inputs and one output.
2. From the Algebraic palette of the Palette Browser, drag two TypeConversion blocks and a Summer block into the SuperBlock. Create the model shown in [Figure 16-7](#).

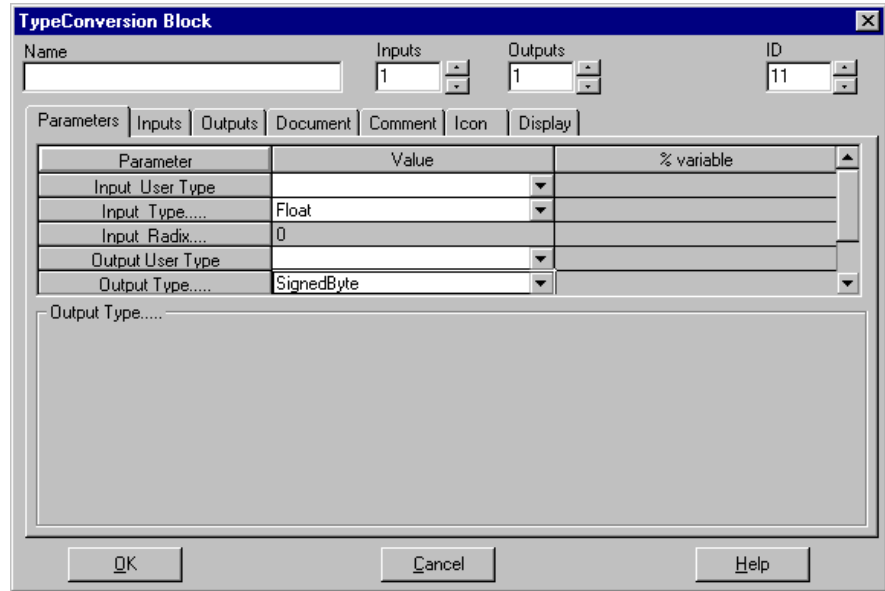
Figure 16-7 **Example Showing How to Compare Fixed and Floating Types; SB0 Case Shown**



3. Raise the TypeConversion Block dialog for the first block. On the Parameters tab, change the Output Type from Integer to Signed Byte.

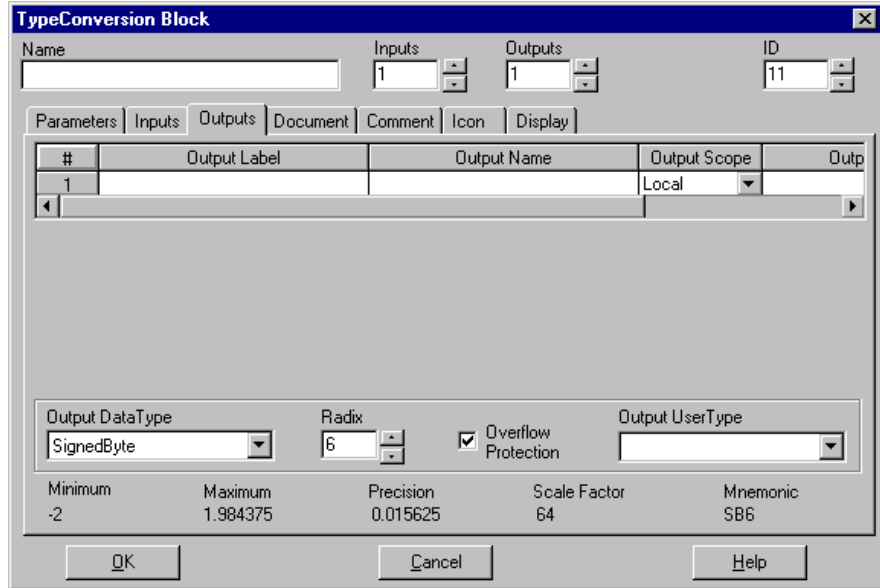
This block accepts a floating-point signal and produces a fixed-point output. See [Figure 16-8](#) for the Parameters tab with these changes made.

Figure 16-8 **Block 1 Parameters Tab**



4. On the Outputs tab, give it a Radix Position of 6 (see).
This is critical to the precision of the output.

Figure 16-9 **Block 1 Outputs Tab**



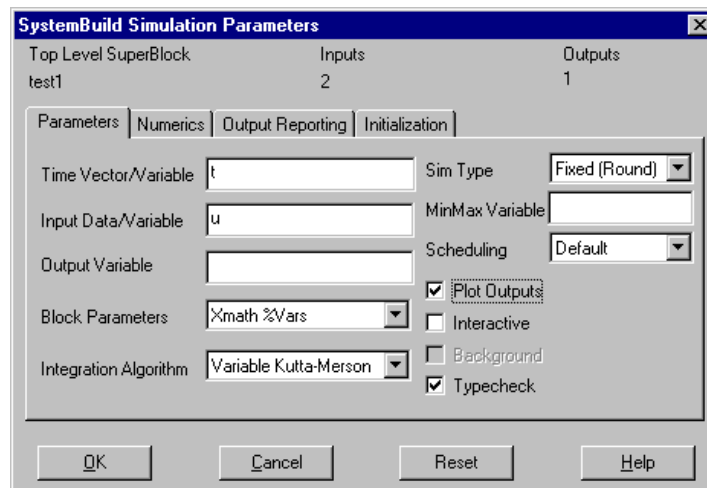
5. From the Parameters tab of the TypeConversion Block dialog for block 2, modify block 2 so that the Input Type is Signed Byte, the Input Radix is 6, and the Output Type is Float.

- To simulate the model, in the Xmath command area, type:

```
t = [0:0.1:10]';  
u = [sin(t),sin(t)];
```

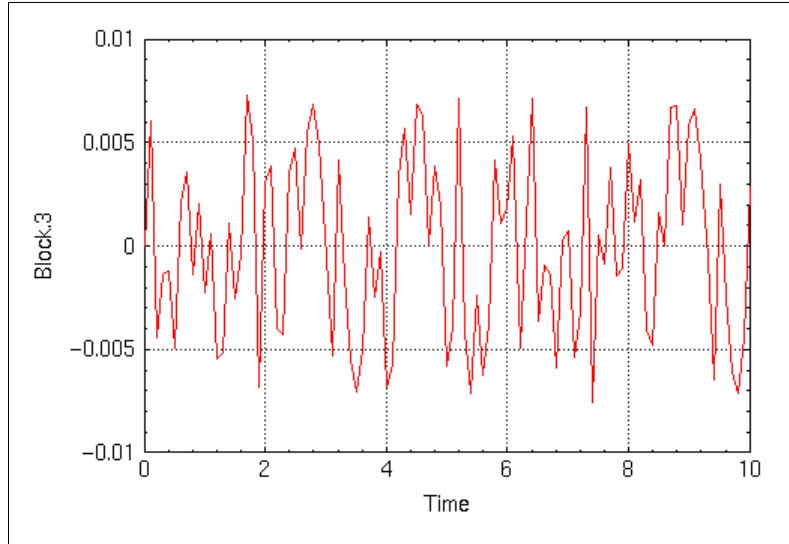
- From the SuperBlock Editor that contains test1, select Tools→Simulate. In the SystemBuild Simulation Parameters dialog, specify t and u for the time vector and the input data variables, respectively. Set the Sim Type to Fixed (Round), and enable Plot Outputs. (See Figure 16-10.) Click OK.

Figure 16-10 SystemBuild Simulation Parameters Dialog, Ready for Simulating Comparison



From the plot (see [Figure 16-11](#)), it appears that, at this radix position, the fixed and floating values of the input sine wave 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 $2^{-6} = 0.015625$.

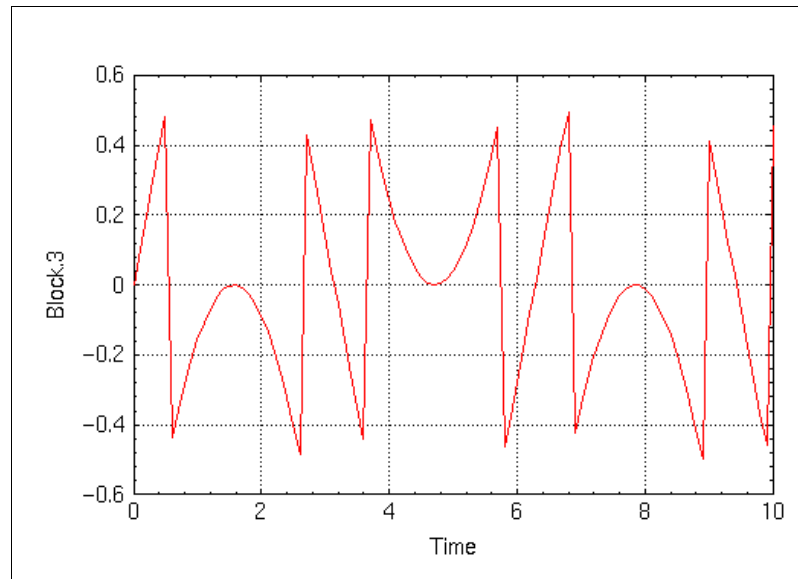
Figure 16-11 **Plot of Comparison with Radix Position 6**



- 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 16-12](#).

Figure 16-12 **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, $\pm 1/2$ the resolution, which is equal to $2^0 = 1$.

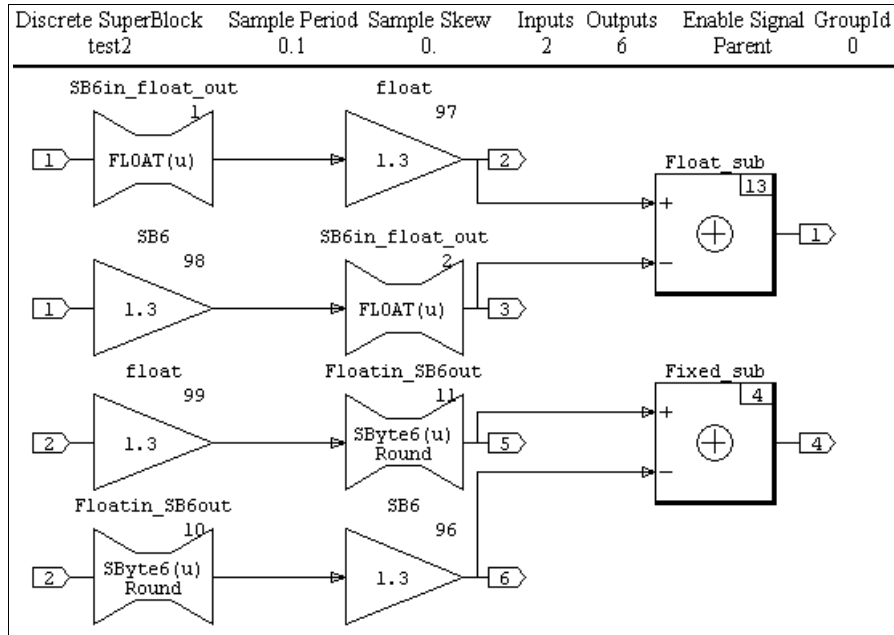
16.2.5 Comparing the Effects of Different Conversion Sequences

The effect of converting between floating and fixed data types for different arithmetic operations varies according to the sequence of operations. This is illustrated in [Example 16-10](#), where fixed- and floating-point inputs are brought into a network of TypeConversion blocks and Gain blocks. Each input is changed to the other data type and multiplied by a constant; it is the sequencing of these operations that is significant.

Example 16-10 **Effect of Data Type Conversion before and after Multiplication**

1. Build up the model shown in [Figure 16-13](#). Give the model a name, test2, and make the SuperBlock discrete. Observe common points of the blocks:

Figure 16-13 **Model for Data Type Conversion before and after Multiplication**



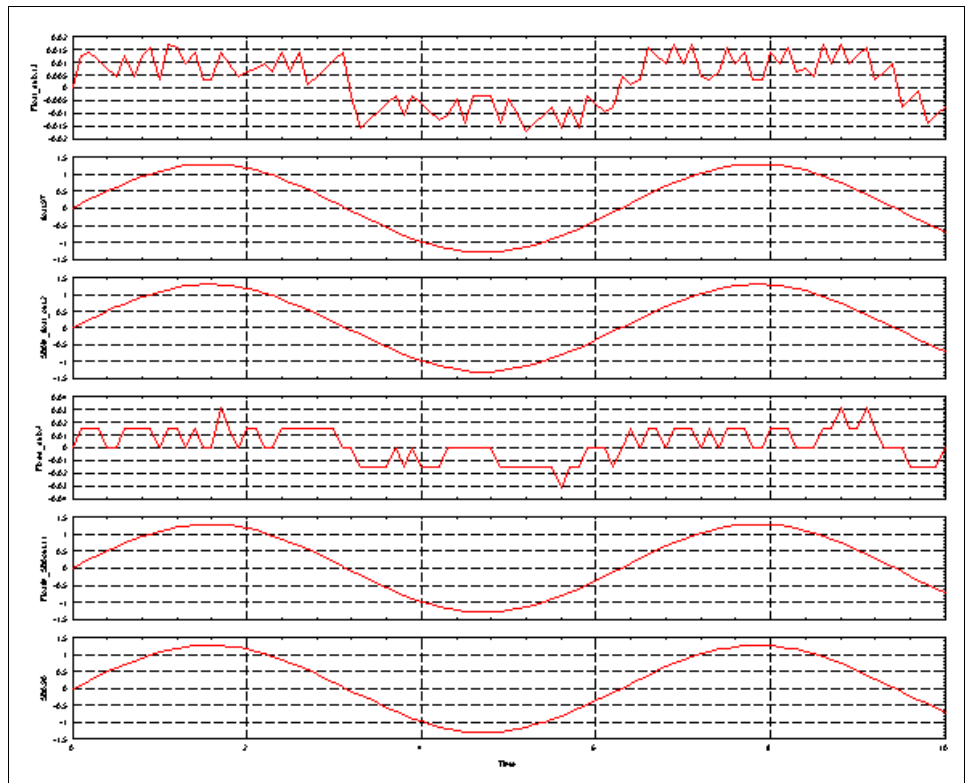
- All the Gain blocks share the same gain value, 1.3. The output data type for the Gain blocks named float is Float; the output data type for those named SB6 is Signed Byte, Radix 6.
- The two TypeConversion blocks named SB6in_float_out have input data types of SB6 and output data types of Float; the other two TypeConversion blocks have input data types of Float and output data types of SB6.
- The upper summing junction, Float_sub, has an output data type of Float; the other summing junction, Fixed_sub, has an output data type of SB6.
- The data type 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 = [0:0.1:10]';  
u = [sin(t), sin(t)];  
y = sim("test2",t,u,{fixpt, graph})
```

The plot should look like [Figure 16-14](#). Observe the many differences between the sequences of operation. You can design many other modes of operation to compare.

Figure 16-14 Plot of the Data Type Conversion before and after Multiplication Example



16.3 Fixed-point Blocks and I/O Data Type Rules

A selected group of SystemBuild blocks have been adapted for use with fixed-point arithmetic. Table 16-1 lists the changed blocks, plus data typing I/O and block parameter rules for using the blocks with fixed-point arithmetic. Wind River software supports four data types: floating point, integer, logical, and fixed. Under fixed, we support 390 sub-types, referred to as fixed-point data types, which constitute the subject matter of this chapter.

Unless otherwise noted, in Table 16-1 the terms *type* and *fixed type* refer to fixed-point data types. Fixed-point types are said to be the same only if the length, sign status (signed or unsigned), and radix position are identical.

Table 16-1 **Blocks Compatible with Fixed Point, with Data Type Rules**

Block Names	Input/Output Data Type 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 data type. ■ All outputs must be the same fixed data type. ■ Output word size must be \geq input word size. ■ See p.434.
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 word size must be \geq input word size.
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 word size may exceed denominator word size

Table 16-1 **Blocks Compatible with Fixed Point, with Data Type 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. <p>Piece-wise Linear</p>
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 word length. <p>Dynamic Blocks</p>
TimeDelay	<ul style="list-style-type: none"> ■ All fixed data types must be the same. ■ Initial output must be the same fixed data type as the output. <p>Logical Blocks</p>
LogicalOperator RelationalOperator	<ul style="list-style-type: none"> ■ All data types are accepted, in any combination.
ShiftRegister	<ul style="list-style-type: none"> ■ All data types 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 16-1 **Blocks Compatible with Fixed Point, with Data Type 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 data type. ■ All outputs must be the same fixed data type. ■ Output word size must be \geq input word size. ■ See p.434.
MatrixTranpose	<ul style="list-style-type: none"> ■ All inputs must be the same fixed data type. ■ All outputs must be the same fixed data type. ■ Output word size must be \geq input word size.
Constant	<ul style="list-style-type: none"> ■ All inputs must be the same fixed data type. ■ All outputs must be the same fixed data type. ■ Output word size must be \geq input word size.

16.3.1 Advanced Simulation Topics

This section provides information for advanced users on topics of Intermediate data types, 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:

- The operation is binary (requires two operands) or unary (requires only one operand).
- The operand types are defined.
- The result data type is defined.

Since the data types of all inputs and outputs are required for any block in a SuperBlock, 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 Summer block with four inputs, formally described by:

$$y = a + b + c + d \quad (16-1)$$

For this discussion assume that the computation order in the above expression is equivalent to:

$$y = (((a + b) + c) + d) \quad (16-2)$$

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 :

$$s1 = a + b \quad (16-3)$$

$$s2 = s1 + c \quad (16-4)$$

$$y = s2 + d \quad (16-5)$$

The question arises as to what the data type 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:

$$y = -a + b \tag{16-6}$$

At first sight, it appears that (16-6) is the same as (16-7):

$$y = b - a \tag{16-7}$$

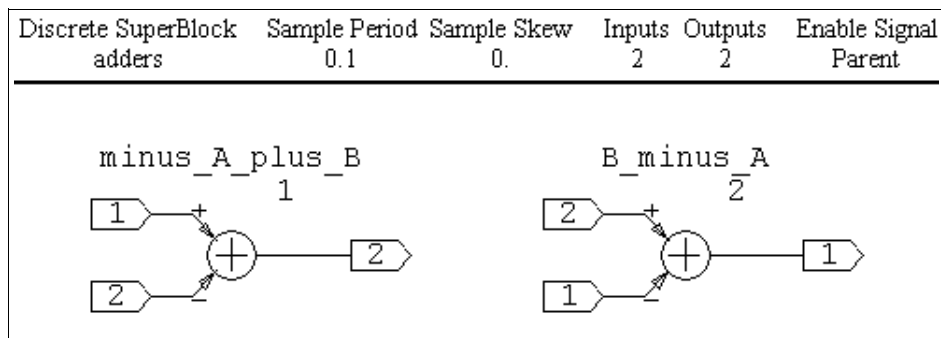
and therefore requires no intermediate variables and hence no ITypes. But closer inspection reveals that (16-6) may be written as:

$$s1 = -a \tag{16-8}$$

$$y = s1 + b \tag{16-9}$$

Thus, (16-6) 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 (16-6) involves an intermediate data type. This implies that (16-6) and (16-7), depicted in Figure 16-15, may lead to results that are different (although numerically close). Using (16-7) 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 16-15 Add and Subtract Sequencing



Wind River's built-in IType rules are based on two goals. Given the operand types and the nature of operation, the ITypes are derived such that:

- 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 [Example 16-11](#), [4b](#) below.)

- Word length 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 data type word length, for example, from 8 bits to 16 bits.) [Example 16-11](#) illustrates this phenomenon.

Example 16-11 Need for Intermediate Types

1. Suppose that $z = ((x1 + x2) + x3)$ with:

Type($x1$):= Signed Byte (8 bits), Radix 3 (SB3)

Type($x2$):= Signed Byte, Radix 4 (SB4)

Then we can use rules **n** and **n** to determine an IType for the sum of $x1$ and $x2$. To apply rule **n** 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 guards against overflow is SB2 (Signed Byte, Radix 2).

2. In [1](#), if the data type of $x2$ were changed such that:

Type($x2$):= Signed Byte, Radix 1 (SB1)

Then the IType based on rules **n** and **n** would be SB0.

3. In [1](#), if the data type of $x2$ were changed such that:

Type($x2$):= Signed Byte, Radix 0 (SB0)

Then in the worst case, no 8-bit signed data type 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, Radix 4 (UB4).

Then, since the negated value is negative, the IType must be signed. Since, in the worst case, data type UB4 might be as large as 15.9375, the appropriate IType is SB3.

- b. Now consider the case where:

Type(a):= Signed Byte, 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 data type (-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 data types 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 data types 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.

Consider a computation with a large number of additions, such as:

$$y = a1 + a2 + a3 + a4 + a5 + a6 + a7 + \dots \quad (16-10)$$

To keep the nesting of intermediate data types from becoming excessive, we have placed a limit on the number of intermediate data type computations that will be performed for any summation or multiplication block. The limit is placed at six; the seventh intermediate data type is set to the result data type and the cycle continues.

You 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 16-15 and 16-16 this means that:

$$(a + b) + c = c + (a + b), \text{ (commutative)} \quad (16-11)$$

whereas

$$(a + b) + c \neq a + (b + c), \text{ (not associative)} \quad (16-12)$$

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 Summer block that adds up three variables a , b , and c perform:

$$(a + b) + c \quad (16-13)$$

$$a + (b + c) \quad (16-14)$$

$$\text{or } (a + c) + b? \quad (16-15)$$

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 block 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 16-16). 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 16-12 below.

Example 16-12 **Possible Implementations of the Expression $a+b+c$**

Figure 16-15 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 Summer block output data type is SB3.

In this setup, it is easy to see how the different implementations might produce different results: assume $a = -14.75$, $b = 6.75$, and $c = 14.75$. Then, using the summation rules for fixed-point data types, the implementation of $a + b + c$ as $(a + b) + c$ yields:

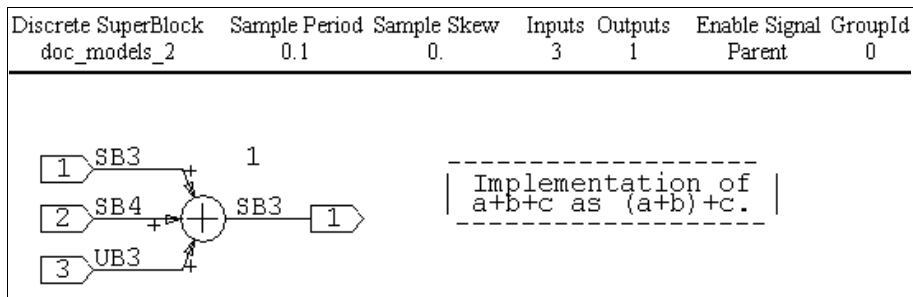
$$\begin{aligned} a + b &= -8 \\ (a + b) + c &= 6.75 \end{aligned}$$

whereas the implementation of $a + b + c$ as $a + (b + c)$ results in:

$$\begin{aligned} b + c &= 15.875 \quad (b + c = 21.5 \text{ but SB3 saturates at } 15.875) \\ \text{Therefore, } a + (b + c) &= 1.125 \end{aligned}$$

Assume that only one Summer block is used to realize the sum. Then the implementation of the equations above is shown in Figure 16-16.

Figure 16-16 **Adding Three Operands**



Now we have

$$\begin{aligned} (a + b) &\text{ has the IType SB2} \\ a + b &= -8, \\ (a + b) + c &= 6.75 \end{aligned}$$

which shows no loss of precision.

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 data types, the result is a value that is clipped at the appropriate limit of that data type.

32-bit multiplication and division are dealt with differently in fixed-point arithmetic. This is explained in *32-bit Operation Issues*.

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 data types are derived from input and/or output types as explained in [Table 16-1](#), p.424. The Gain block exception is explained in *Gain Block: A Special Case*.

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 word size available is only 32 bits and therefore operands cannot be promoted to a higher word size 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 (that is, 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 (that is, 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 Wind River 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 data types with headroom for possible calibration using Run-time Variable editing (see [Example 16-13](#)).

The fixed-point data types are defined by three variables: signed or unsigned, length, and radix position. In setting the data type 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 word size of the parameter is defined by the input data type.
- 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).

16.3.2 Radix Calculations

If you specify an UnsignedShort as the output type of a Gain block, and place the number 3.1 inside it, the assigned radix is 14.

If you specify the output data type of the Gain block as fixed-point, the Parameters tab in the block dialog offers a Define Radix field with a default value of No. If you click to change this value to Yes, a Gain Radix field becomes active, 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 word size and sign status of the gain parameter, which are the same as the word size and the sign of the parameter.

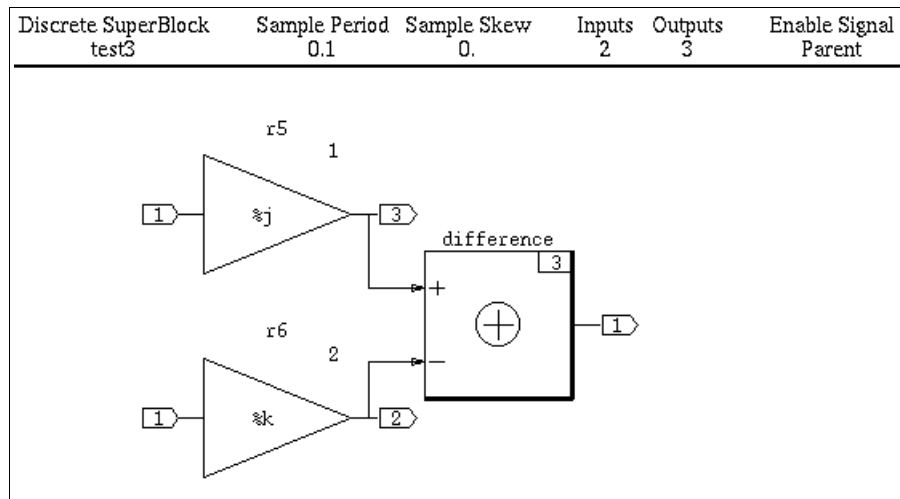
Define Radix position used in conjunction with RVE helps you interactively establish an optimal fixed data type for a given parameterized Gain block. The operation of the Define Radix position feature is illustrated in Examples [16-13](#) and [16-14](#).

Example 16-13 **Building and Exploring Gain Blocks with Define Radix Position**

In this example we build 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 data types later. Start building the model shown in [Figure 16-17](#).

Figure 16-17 **Gain Block Radix Example**



2. Name the first block r5. Make the Gain 1.3, and the %variable equal to 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 is an active field with default No. Change the value to Yes. The Gain Radix field becomes active; 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. From the SuperBlock Properties dialog, change the Inputs to 2. Click the Inputs tab, and change the Input Types to SignedByte. Change the Input Radix to 6. Make all data types the same.

5. Set the Output Data Type of the Summer block to SignedByte with an output Radix of 6. Name the summing junction difference.

6. Prepare for simulation. In the Xmath command area, type:

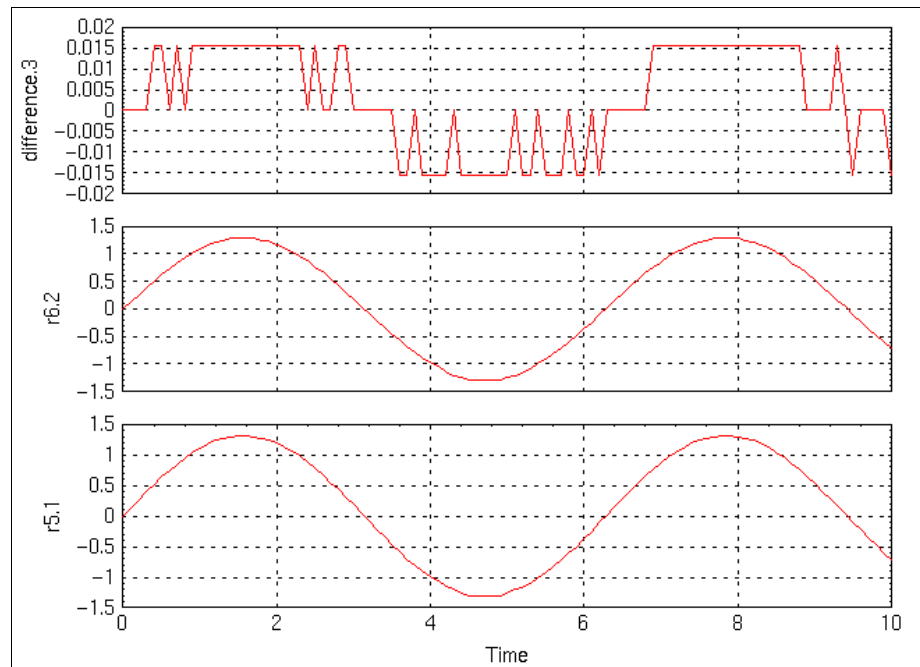
```
t = [0:0.1:10]';  
u = [sin(t), sin(t)];
```

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})
```

The plot should look like [Figure 16-18](#).

Figure 16-18 **Plot Comparing Different Gain Radix 5 vs. 6**



The spikes in the top strip of [Figure 16-18](#) 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 data type fixed-point numbers.

- Both input sine waves were multiplied by 1.3, which was variously expressed as two fixed-point numbers of different data type: 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 16-14 **Overflow Caused by Gain Values**

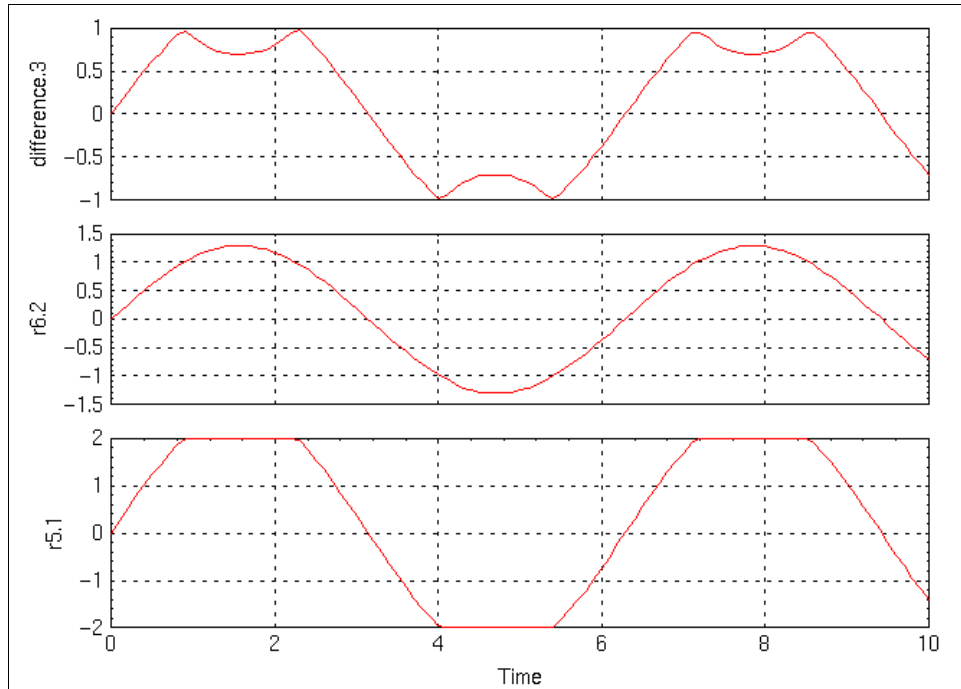
For this example, we use the same model but a different gain value that forces overflow on one of the channels of comparison.

1. Display the model from [Figure 16-17](#).
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](#), p.436 and the `sim()` command from [Step 7](#).

The plot should look like [Figure 16-19](#).

Figure 16-19 Plot of Gain Radix Positions with Overflow



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 data type is SB6 and therefore whose range is $[-2: 1.984375]$. The other block, whose data type 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 16-19](#), and you can see the approximate amount of the overflow in the top strip.

16.4 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 to Xmath and stored in a special MinMax data set. The name and partition location of the data set 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 data set is created it can be viewed with the Minmax Display GUI interface.

Restrictions:

- No continuous SuperBlocks
- Does not work with the **resume** keyword in simulation

16.4.1 Activating MinMax Logging

In the SystemBuild Simulation Parameters dialog, enter a variable name (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 data set), with your specified name.

Simulating with the minmax Keyword

The **sim()** keyword **minmax** takes a string specifying the name of the Xmath data set 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 Data Sets to a File

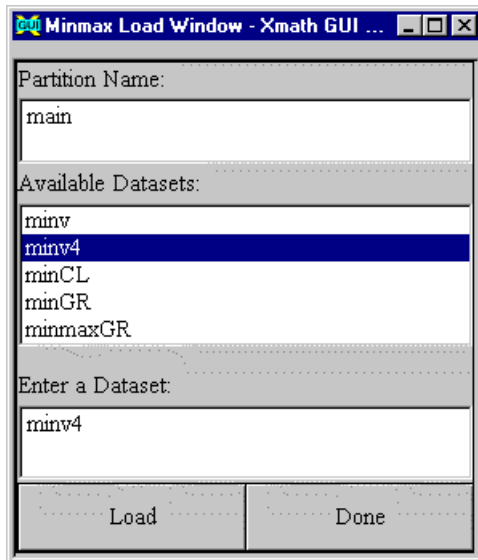
Data sets are stored as Xmath list variables. Like any other Xmath variable, they can be saved to a file with the **save** command. If you are saving a catalog object from the Catalog Browser, be sure to include Xmath data if you want to preserve your data sets.

16.4.2 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 data set for display, select Special→Load. The Load window displays the name of the current partition and all data sets available within it. The partition can be changed. Select a data set; then click Load. Click Done when you are finished.



The MinMax Tool is shown in [Figure 16-20](#).

Figure 16-20 **MinMax Display, Simple Overflow (Figure 16-19 Example Shown)**

The screenshot shows a window titled "Minmax Display - Xmath GUI Tool" with a menu bar containing "Special", "Select", and "Help". The main area is divided into two sections. The top section displays a table of output data for the dataset "main.minv". The bottom section displays a table of SuperBlocks.

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

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 that it was disabled, and an empty column indicates that it was enabled. The output information for the block you select is displayed in the top area.

For each output, the top area displays the selected block's name, data type, the initial minimum and maximum values, and the times they occurred.

Display Options

Select Output Display from the Select menu 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 is raised whenever it is selected in the MinMax dialog.

16.5 User-Defined Data Types (UserTypes)

As a convenience in situations where multiple data types are required, you can assign your own meaningful names to data types, called user-defined data types or UserTypes. You can use these names in all the block data type information dialogs. A special editor is provided to let you change the meaning of a custom data type.

16.5.1 UserType Editor

The UserType Editor provides an interactive interface for UserType creation, modification, and deletion. This tool is launched from the Xmath Commands 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:

In the Xmath Commands window, type

UserType

The Windows implementation of the UserType editor is shown in [Figure 16-21](#); 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.

Figure 16-21 **UserType Editor**

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	

To create a new UserType — First type the UserType name in the Name field and press Return. Then choose the data type for the new UserType. When you are satisfied with the UserType name and data type, 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 data type, 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.

16.5.2 UserType MathScript Commands

A set of Xmath commands provide the same functionality as the UserType Editor. See online Help for more information on each command. Try typing the commands in [Example 16-15](#).

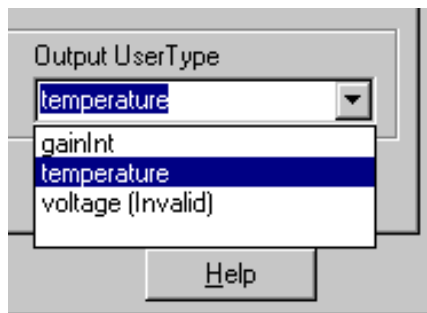
createusertype	Creates a new UserType definition.
modifyusertype	Changes the data type of a UserType.
deleteusertype	Deletes a UserType.
listusertype	Lists out all defined UserTypes in the Xmath Commands window.

Example 16-15 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
```

16.5.3 Using UserTypes in SystemBuild

UserType information is located wherever data type 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 are listed in the UserType combo box. If a UserType is illegal in the current context, it is marked (Invalid), as shown above).
- Once a UserType is entered, other data type items become read only.
- If a UserType is deleted but still referenced in a block, the last value of the UserType is used.

Example 16-16 **Creating and Using a UserType**

1. Create a UserType named **test1**.
2. Create a Gain block in a model.
3. Open the Gain Block dialog. Go to the Outputs tab. Enter the UserType name **test1** in the field named Output UserType (bottom of tab). Click OK to close the dialog.
4. Go the UserType Editor. Change the data type of **test1**.
5. Open the Gain block again. Inspect the output data type. It should be updated to the new UserType definition.

16.5.4 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.

16.6 SystemBuild Functions in Fixed-Point

16.6.1 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 (that is, parameters defined in the block forms of each block in the model) are quantized according to their fixed-point data types.
2. The operating point inputs and state (that is, 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 (that is, 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 [10.5 Multirate Linearization](#), p.230.

16.6.2 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 data types.

However, for the calculation of \dot{x} , the computation is done in floating-point. \dot{x} is the “pseudo-rate”, which is computed from

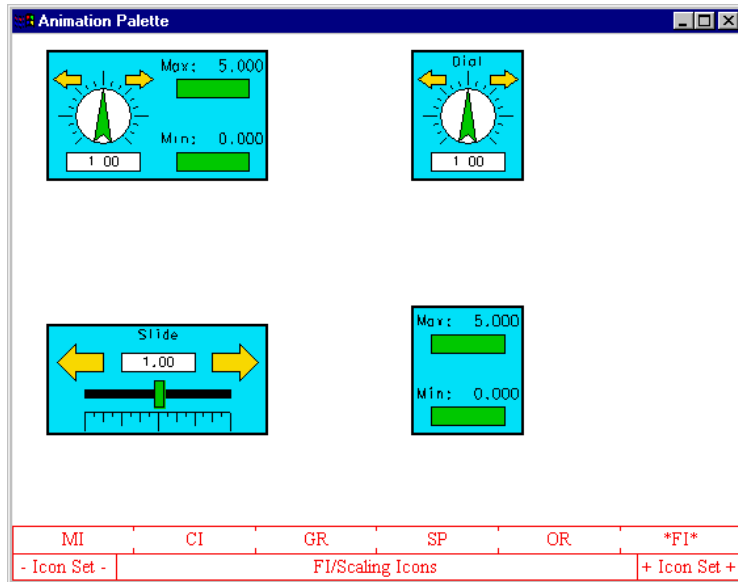
$$\frac{x[k + 1] - x(k)}{\Delta T} \quad (16-16)$$

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.

16.7 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 16-22](#)).

Figure 16-22 **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.

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 Wind River 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 has 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. [17.3.1 Restrictions on Component SystemBuild Hierarchies](#) 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 occurs.

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 [17.3.7 Creating Components Using the Component Wizard](#).

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 SuperBlock 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 [17.2.3 Controlling Component Parameters](#) and in [17.2.4 Loading Component Parameter Sets](#).

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 [17.2.5 Changing Scope into a Component Catalog](#).

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 determines 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. You are 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. You can

simulate and generate documentation for models that reference encrypted components, but the model output does not expose the internal details of the encrypted component.

- You cannot generate code for models that contain references to encrypted components.
- You cannot load encrypted component files into the SuperBlock 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 [17.3 Creating Components](#).

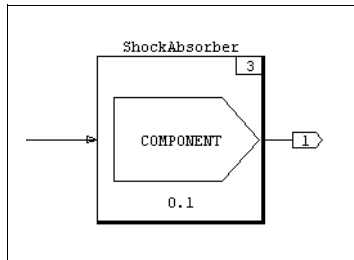
References to components are similar to references to “built-in” Wind River blocks (for example, a Gain block). They can be used in the same manner as Wind River blocks.

17.2.1 Viewing Components

This section briefly describes how to view components in SystemBuild’s Catalog Browser and editor windows. See online Help for Catalog Browser for more details.

[Figure 17-1](#) shows a component reference in the SuperBlock Editor. The representation of component references is similar to that of SuperBlock references.

Figure 17-1 **Default Component Icon**



Component references are displayed in the Contents view of the Catalog Browser, just as with other block references. If a model contains regular components, then the component’s definition is listed in the Components folder of the Catalog view.

17.2.2 Creating References to Components

You can create references to components 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.

You can use the following methods 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 view in the Catalog Browser, and click a SuperBlock that references a component. The Contents view lists 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 the Libraries folder, and select a library catalog. A list of components available in the library catalog is displayed. Drag and drop a component from the Catalog Browser 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 Wind River block (see [Chapter 20](#)).
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 appears 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 on the Parameters tab of the Component Block dialog.

- 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 the **Psets_Load** command. See [17.4 Creating and Using Parameter Sets](#).

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 are 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.

[17.4 Creating and Using Parameter Sets](#) contains more details on PSETs.

17.2.5 Changing Scope into a Component Catalog

Typically, you do 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 Shortcut menu and select Component Catalog. Note, not every component allows you to scope into its catalog (see [Table 17-1](#), p.453).

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, select View→Main Catalog.



NOTE: If any changes are made within a component catalog, the component needs to be re-componentized. See [17.3.8 Modifying Components](#) 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 is 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 [17.3.7 Creating Components Using the Component Wizard](#).

17.3.1 Restrictions on Component SystemBuild Hierarchies

SystemBuild components encapsulate SuperBlock hierarchies. Therefore, any modeling element that breaks this encapsulation is not allowed in a component hierarchy. These elements cannot be included in components:

- DataStores
- 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 fails.

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 assumes 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 reports an error at the time of component creation (see section [17.3.7 Creating Components Using the Component Wizard](#)).

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 assume the value specified by the parent component, if any. Otherwise, they 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 you.

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 [17.3.1 Restrictions on Component SystemBuild Hierarchies](#), 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 that, 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 at every time step during simulation but only at the start of the simulation or if the mapping variable is changed during RVE (run-time variable editing). Therefore, mapping is 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 might 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 online Help), although the interface is not used when you specify an MSF. The event string is "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 [17.6 Distributing SystemBuild Components](#).
- 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, and 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?
- Does this component use mapping? If so, gather the mapping parameters for this component in a single Xmath partition. The Component Wizard uses the designated partition to obtain the parameter dimensions. The parameter values become the default values for the component's mapping parameters. The new component no longer needs the original Xmath partition or its contents.

Make sure that you prepare the equations you need 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.

- 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 so, gather the variables into a single Xmath partition. The Component Wizard uses the values of the variables found in the specified partition. Once created, the component no longer needs or refers to the partition that contains the defaults for the exported variables.

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:

1. In to the Catalog browser, select the component, and 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.

2. Specify the new interface, and re-componentize the component.

To modify the internal details of a component, change scope to the component's catalog as discussed in [17.2.5 Changing Scope into a Component Catalog](#), and make the necessary changes. When the scope is changed back to the component's parent catalog, the Component Wizard is invoked 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 [17.3 Creating Components](#).

17.3.9 Unmaking a Component

To reduce a non-encrypted component to a SuperBlock hierarchy in its parent catalog, select the component, and then select Tools→Unmake Component. This replaces the component in the parent catalog with the SuperBlocks within the component. If there are any name conflicts, you are 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

This example creates, saves, and loads a parameter set called **MonsterShocks** for a **shock_absorber** component that has exported the parameters *spring_rate* and *setting*.

1. 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** is only 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 issuing the **Psets_AddToList** command.

```
Psets_AddToList pset1
```

The component interface expects a PSET to be the only object in a single file.

3. 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, it can be distributed along with a component as a custom block on a custom palette; see [20.2.1 What Kinds of Blocks Can Be Customized?](#), p.521.

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 combo box on the Parameters tab of the Component Block dialog 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 the 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 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 you 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 or documentation file. For a detailed description of distributing a component as a custom block, refer to [20.2 Custom Blocks](#), p.521.

Encrypting and Licensing Components

Components can be encrypted using the **ENCRYPT** command described in 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, and then select File→SaveAs. In the Save dialog's SuperBlocks field, choose Selected, and then press OK. The saved component is 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 do 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. With the SuperBlock `proto` selected in the Catalog Browser, select Tools→Make Component.

The component wizard appears.

3. Click Finish to accept the default settings.

In the Main Catalog view, observe that the SuperBlock `proto` has moved from the SuperBlocks folder.

4. In the Catalog view, click the Components folder to display the components in the Contents view.

5. Open the component `proto`.

6. Navigate into the SuperBlock `test`. Change the Gain block value to **99**. Press OK.

7. Click the Parent toolbar button to return to the `proto` component; then click the Parent toolbar button again to change the scope back to the Main catalog.

Since the component has been changed, the Component Creation Wizard appears as you leave the component's scope.

8. Click Finish to re-package the component.
9. From the Catalog Browser, open the SuperBlock `ex1`, and navigate into the test SuperBlock.

10. Inspect the gain value in SuperBlock test.

The value is still the original default value of 1.0. Changing the SuperBlock 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`, and 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 the SuperBlocks folder, 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.

The exporting definition screen appears.

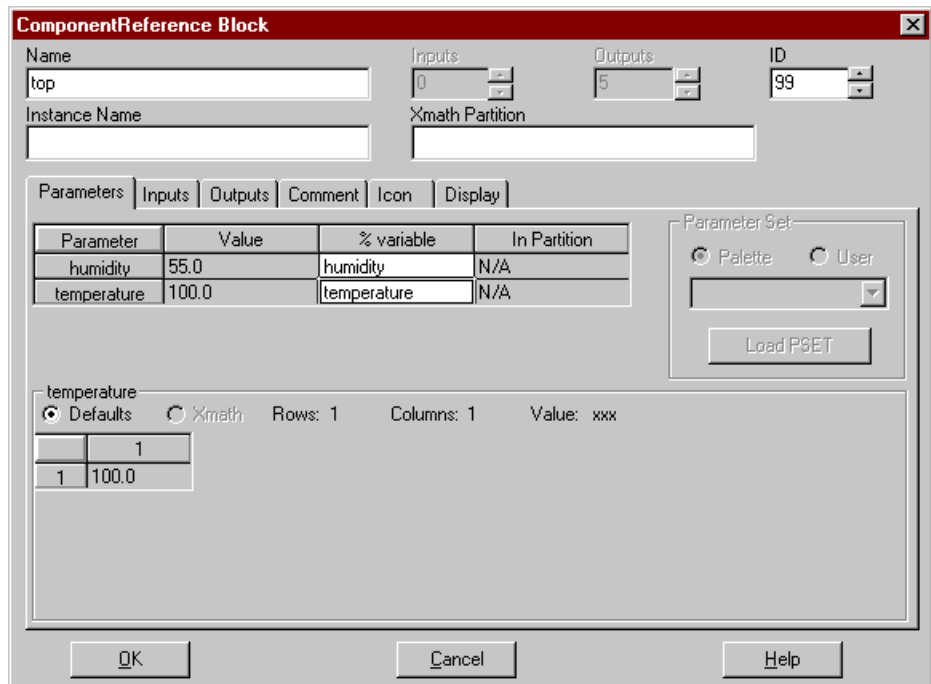
4. Export the %Variables *temperature* and *humidity*, and the variable block `varblock2`. At the bottom of the form, enable Replace. Type `ex2` in the partition name field. Click Finish.



5. In the Catalog Browser, open the SuperBlock ex2, and then open the block dialog for the component reference top.

Note that the exported %Variables are visible (see [Figure 17-2](#)). Click 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 uses parameter sets (PSETs) with the component top created in [17.7.2 Exporting Component Parameters.](#)

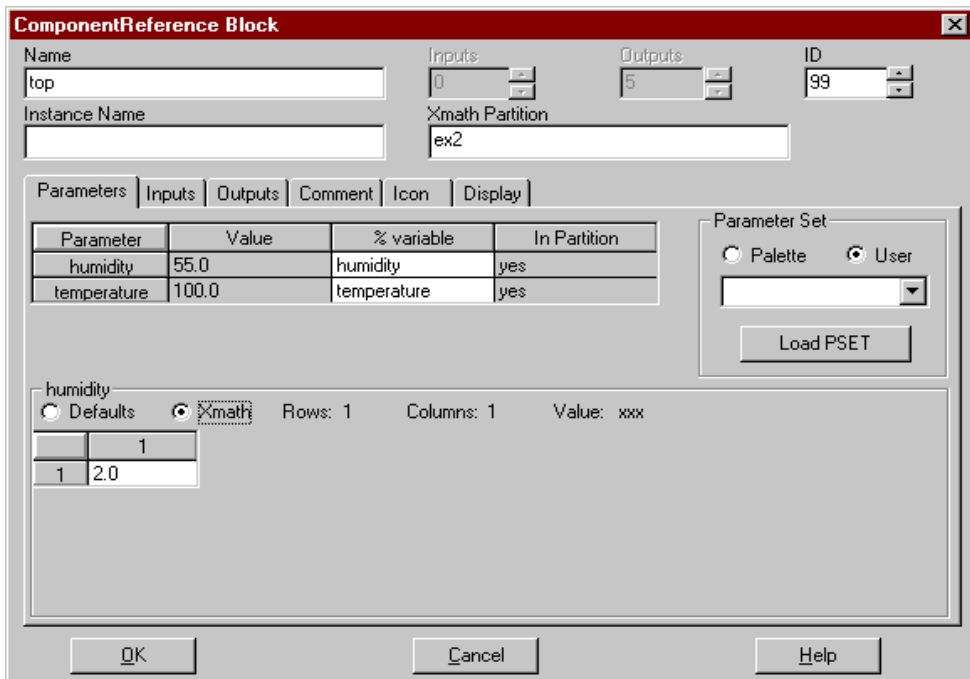
1. In the Xmath command area, make sure you are in partition ex2 , and then define PSETs for the component top:

```
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});
```

2. 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 do not activate until you tab out of the Xmath Partition field.)
- In the Parameter Set area, select User.
The combo box displays the PSETs defined above.
- Select a PSET, and then click Load PSET.
PSET values are now available in the block dialog.
- Select a parameter (either *humidity* or *temperature*). In the display area below, enable Defaults to view the original %Var, or enable 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 uses 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.

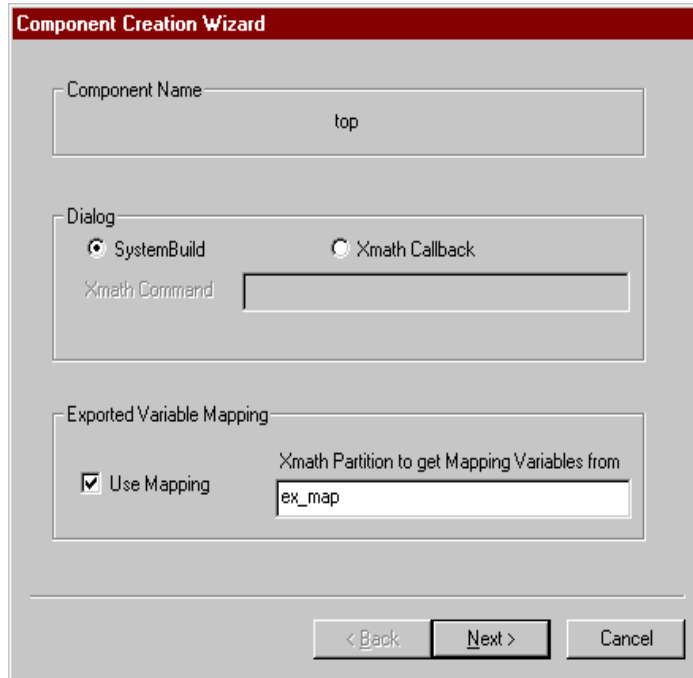
1. Load the example data:

```
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, enable Use Mapping. In the field Xmath Partition to Get Mapping Variables from, enter **ex_map** as the partition from which to get mapping variables. Click Next.



The screenshot shows the 'Component Creation Wizard' dialog box. It has a red title bar and a grey background. The dialog is divided into three main sections:

- Component Name:** A text field containing the text 'top'.
- Dialog:** Two radio buttons are present: 'SystemBuild' (which is selected) and 'Xmath Callback'. Below these is a text field labeled 'Xmath Command' which is currently empty.
- Exported Variable Mapping:** A checkbox labeled 'Use Mapping' is checked. To its right is a text field labeled 'Xmath Partition to get Mapping Variables from' containing the text 'ex_map'.

At the bottom of the dialog, there are three buttons: '< Back', 'Next >', and 'Cancel'. The 'Next >' button is highlighted with a black border.

4. In the next dialog, export the parameter *temperature* only. Click Next. 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:

$$\text{temperature} = (9/5) * c + 32;$$

Component Creation Wizard (Xmath Parameter Definition)

Mapping

Exported Vars [Reference Only]

temperature

Xmath Mapping Partition: ex_map

Parameter

c

Add

Remove

c (1.1)

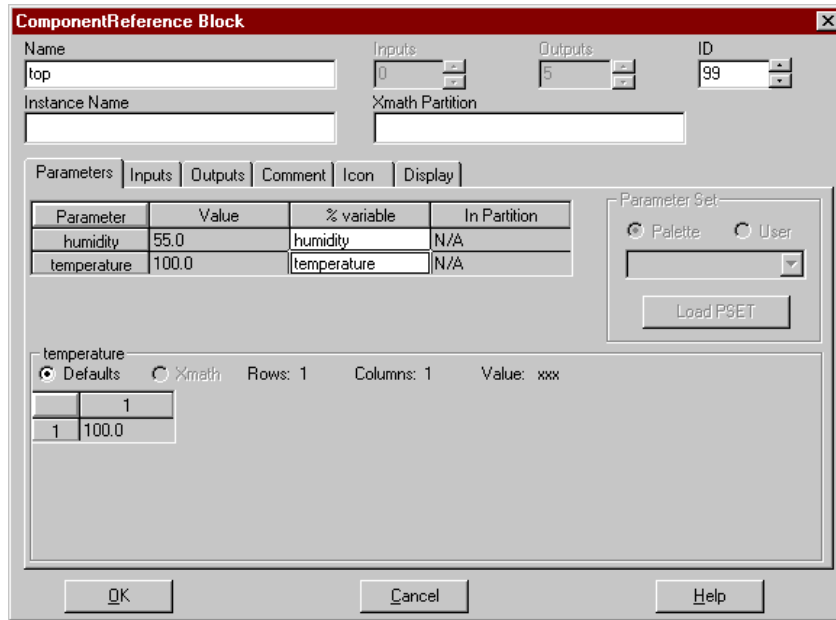
Mapping Expression: Exported Vars = f (Parameters)

temperature = (9/5) * c + 32;

< Back Finish Cancel

- Click Finish.

7. Open the block dialog form for the component reference, and note that the Parameters tab displays only the %Variable *c* with an initial value of 8.



17.7.5 Using a Custom Dialog

In this example, the Xmath dialog functions `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.

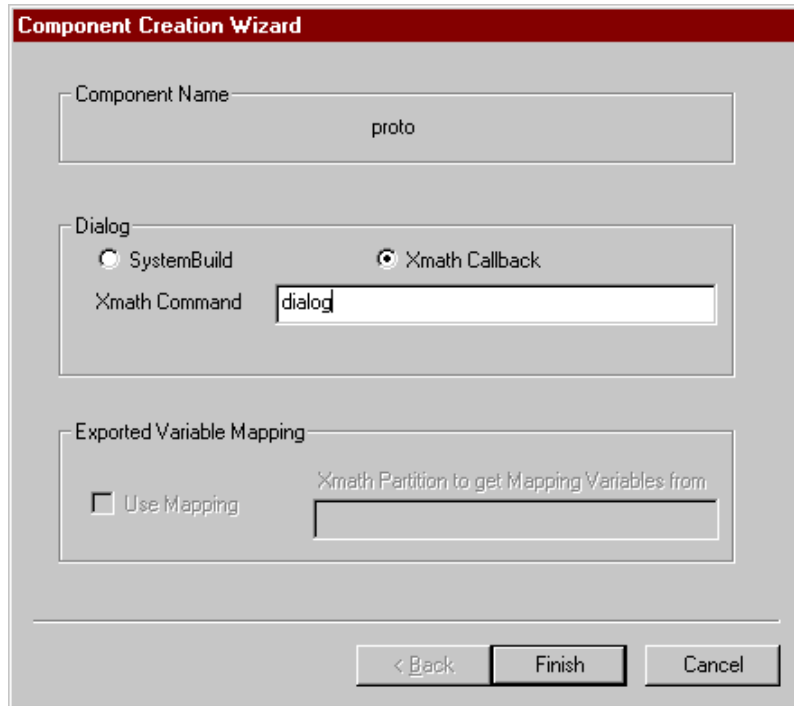
1. Copy the sample MathScript function to your local directory.

```
copyfile "$SYSBLD/examples/components/dialog.msf"
```

2. Load a simple demonstration model:

```
load "$SYSBLD/examples/components/encap.dat"
```

3. In the SuperBlock ex1, make a component out of the SuperBlock proto. In the Component Creation Wizard, enable Xmath Callback. In the Xmath Command field, type **dialog**.



The screenshot shows the 'Component Creation Wizard' dialog box. It has a title bar with the text 'Component Creation Wizard' in white on a dark red background. The main area is light gray and contains three sections: 1. 'Component Name' with a text box containing 'proto'. 2. 'Dialog' with two radio buttons: 'SystemBuild' (unselected) and 'Xmath Callback' (selected). Below the radio buttons is a text box labeled 'Xmath Command' containing the text 'dialog'. 3. 'Exported Variable Mapping' with a checkbox labeled 'Use Mapping' (unchecked) and a text box labeled 'Xmath Partition to get Mapping Variables from' which is currently empty. At the bottom of the dialog are three buttons: '< Back', 'Finish', and 'Cancel'.

4. Click Finish.
5. In the Catalog Browser, open ex1 for editing.
6. Open the component proto block reference.

A single text entry box appears from the Xmath Commands 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

SystemBuild Customization

This chapter details how to customize SystemBuild with the user initialization file (**user.ini**) and the resource file (**Sysbld**).

This chapter contains the following main topics:

- The *User Initialization File* allows you to customize such items as editors and menus.
- The *SystemBuild Resource File (UNIX)* allows you to customize colors and size and placement of windows.

18.1 User Initialization File

The SystemBuild user initialization file (**SYSBLD/etc/sysbld.ini**) defines the default printing and editor settings, the 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 **user.ini** file is placed in **SYSBLD/etc** (by someone with root or administrator privileges), all users see the customization. SystemBuild also reads this file if it is in your startup directory.

18.1.1 File Format

The basic `user.ini` file format is shown in [Example 18-1](#). Use Xmath's `copyfile` command to copy this file to your local startup 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 MATRIX_X.



NOTE: Be sure to make a copy of the sample file before you edit it for your purposes.

Example 18-1 **Sample user.ini File**

```
#####  
#                               user.ini                               #  
#####  
#                               SystemBuild Configuration file for User Menus #  
#-----  
# 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
```

18.1.2 Printer Settings (UNIX)

User initialization file (**user.ini**) printer settings are ignored on Windows platforms; they are used 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 18-1](#). 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.

18.1.3 Default Text Editor

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



NOTE: The comment editors (see [Comment Editor](#)) 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). 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:

In the **OPTIONS** section, alter the **TextEditor** definition (see [Example 18-1](#)).
For example:

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

To change the default text editor from your operating system command line, type the appropriate command:

```
setenv EDIT_COMMENT editor_name #UNIX
set EDIT_COMMENT=editor_name    #Windows
```

where *editor_name* is the name of your editor program (for example, **xemacs** or **Wordpad**). The change takes effect after you close and reopen SystemBuild.

18.1.4 Comment Editor

Text that you enter into the Comment 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 18-1](#). 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. Although the comments document is stored with the model file and cannot be accessed from outside SystemBuild, you can 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 pertains to one instance of the SuperBlock—that is, 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 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.

18.1.5 Custom Menus

You can define and add one or more menus to the menu bar for the SuperBlock Editor from the user initialization file; they appear before the standard Help menu in the order that they are defined.



NOTE: You cannot alter the standard SystemBuild menus or their contents, and you cannot change the Catalog Browser and Palette Browser menu bars.

Your menus might invoke MathScript functions or commands, send a command to your operating system, or execute a shell script.

Define your custom menus in the platform-independent ASCII file, **user.ini**:

- To create a menu, first specify **menuItem=Pulldown**. Menu Items that appear on the pulldown menu can be **PushButton** (a normal menu entry) or **Separator** (a dividing line).
- The **Label** is the text that appears 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 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 you specify **System**, 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.

You must restart SystemBuild whenever you modify or introduce a new **user.ini**.

18.1.6 A Typical Template for User Menus

A typical user initialization file for menu item declaration is shown in [Example 18-2](#); you can copy this file from **SYSBLD/examples/sbmenu**. The lines that start with the pound sign (#) are comments. The template is structured into sections and subsections.



NOTE: This template contains UNIX-specific system calls that you can comment out or delete as necessary.

Example 18-2 Typical User Initialization File 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: dYMMDDtHHMMSSsave.asc
FuncType = Xmath
Function = FORCEDSAVE
```

18.1.7 Using the Sample User Initialization File that Calls MSCs

`SYSBLD/examples/sbmenu` contains the file `user.ini` and two MathScripts that are called by the custom menus specified in the initialization file. [Example 18-3](#) has you load and try files that call MSCs; it creates two custom menus in your SuperBlock Editor: the Global Diagram Changes menu and Miscellaneous menu.

Example 18-3 Using a Sample Initialization File That Calls MSCs

To try a sample file that calls MSCs:

1. Copy a sample `user.ini` file and two sample MSCs from the SystemBuild distribution:

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

`COPYFILE` copies the files to your current working directory.

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

18.2 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 SuperBlock Editor, 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 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, and finally in `SYSBLD/etc`.



CAUTION: Release 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, and then delete the obsolete file.

18.2.1 Controlling Colors

You can adjust colors, sizes, and positions of the SystemBuild editors and Interactive Simulation (ISIM) window by creating a SystemBuild defaults file named `Sysbld`.

Foreground and Background

Foreground and background colors in the SystemBuild editors and Interactive Simulation window are each controlled by two variables in the `Sysbld` file. The variables are listed in [Table 18-1](#).

Table 18-1 **SystemBuild and ISIM Color Defaults**

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

The selection of colors that you can use in the screen specifications on UNIX platforms is 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, and 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 editors, and `isim.color1` through `isim.colorE` for interactive simulation. The color names are the same as for the foreground and background colors.

18.2.2 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 corner and the window's width and height. The numbers are percentages of the screen's width and height, as appropriate. [Table 18-2](#) lists the variables and their permissible ranges.

Table 18-2 **Display Sizing and Positioning Variables**

Variable	Range	Default
Editors		
<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.

19

Custom Icons

This chapter teaches you how to build custom icons for SystemBuild blocks and for the Interactive Animation palette.

19.1 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 SuperBlock Editor. You can use the IA language to create a custom block icon.
- The Interactive Animation module 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.

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 SuperBlock Editor; click the IA toolbar button in the editor's toolbar. This action provides access to a set of five default palettes.

19.1.1 Adding a Custom Icon to a Block Diagram

There are several ways to add an icon to a block diagram:

- Type icon source code into the Icon tab and then set the block icon type to Custom to see the new icon ([19.2.2 Creating or Attaching an IA Source Icon](#)).
- Reference or import an external bitmap ([19.2.1 Importing or Referencing an External Bitmap](#)).
- 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 (see [Example 19-2](#)).
- 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 (see [Example 19-3](#)).

19.1.2 Sample Icon Source

To view examples of icon source code, look at the source files that create the five main palettes delivered with IA:

moni.src	Monitor Animation Icons
coni.src	Controller Animation Icons
graf.src	Graphic Shapes Icons
spec.src	Special Animation Icons
orig.src	Original Animation Icons

Each file contains all icon definitions for an existing IA palette. These files are typical of the resources found in **SYSBLD/src**.

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** 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** defines the IA Builder Control Panel. In **SYSBLD/src**, the file **control.src** is provided to allow you to rearrange the control panel. Make a copy of **control.src**, and proceed with caution when rearranging the control panel.

19.2 Defining Custom SystemBuild Icons

You can use the Icon tab in the dialog of any primitive block to define your own icon design, reference an external bitmap, or import an external bitmap.

19.2.1 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. You can use any size bitmap. It's up to the user to size the block in a way that best displays the bitmap. You can reference a bitmap as an external file or import a bitmap 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:

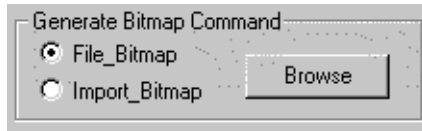
```
FILE_BITMAP 'path_to_file' [xloc yloc xsiz ysiz]
```

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*, and *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. On the Icon tab, go to the Generate Bitmap Command area.



Enable either File or Import, and then click the Browse button. Locate the desired file, and then double-click it. When the file is found, the proper IA source is generated in the Custom Icon text field.

On the Display tab, change the Icon Type to Custom to view the new icon.

19.2.2 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 displays the image you have defined. This image is 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 [19.3 Icon Source File](#) 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 19-1](#)).

Example 19-1 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:
2. Open the SuperBlock continuous_automobile.
3. When the car model appears, select the engine block (a LimitedIntegrator block), and press Return to open the dialog; select the Icon tab.
4. 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'
```



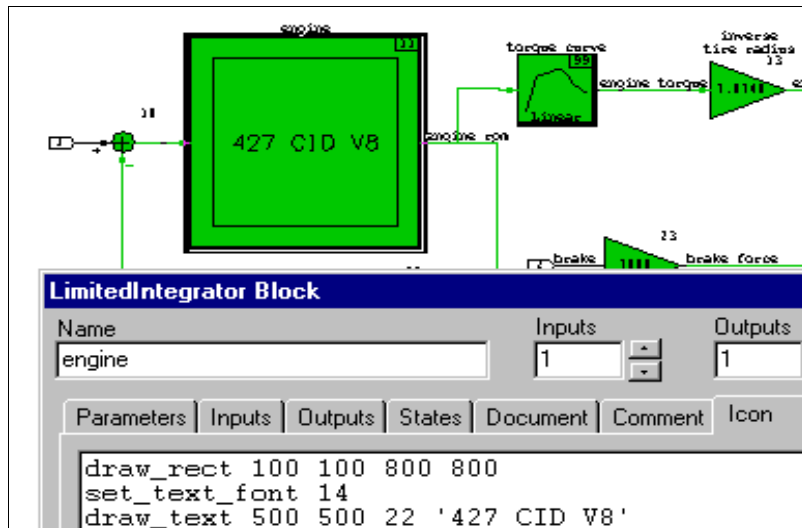
NOTE: By default these commands are in the static graphics section of the icon definition.

5. Click OK to release the block and compile the icon.
6. Place the cursor over the block and press the s key until the icon type changes to Custom.

See [Figure 19-1](#) 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 19-1 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 19-2](#) for attaching a custom icon.

Example 19-2 **Attaching a Precompiled .sog File Icon by Name and Filename**

In this example, we attach a custom icon (strip chart) supplied by Wind River 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, open the accelerometer block's dialog.

This is the Gain block with ID 6.

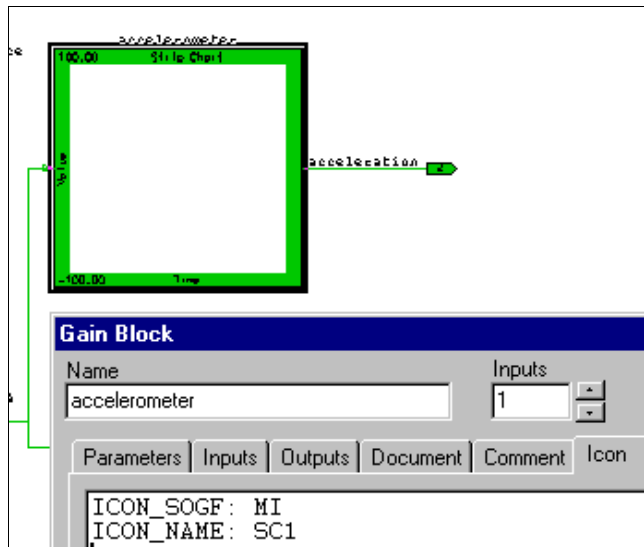
2. In the Icon field, type:

```
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 can be seen.

See [Figure 19-2](#) for a view of this icon and the dialog box with the .sog file and icon name.

Figure 19-2 **Strip Chart Icon Attached to a Primitive Block**

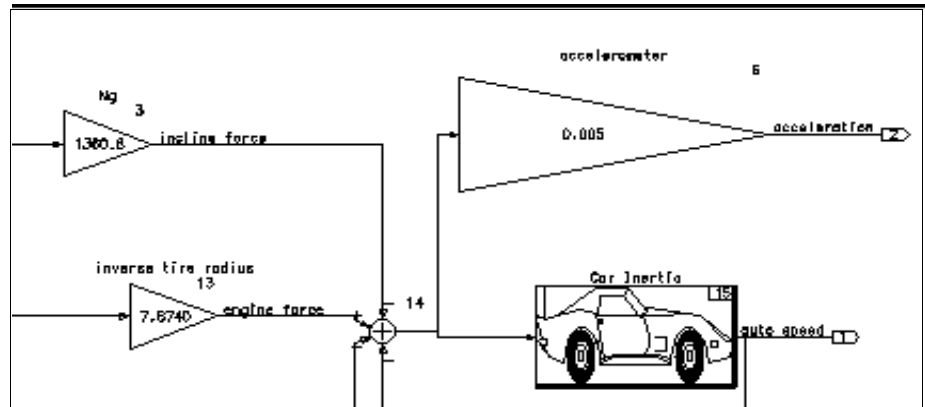


A third method for attaching a custom icon to a primitive block is by drag-and-drop. [Example 19-3](#) illustrates how this is done.

Example 19-3 Attaching a Custom Icon Using Drag-and-Drop

The custom icon has no functionality; it merely changes how the model looks.

1. With the `continuous_automobile` model displayed, click the IA icon in the editor's toolbar.
2. When the palettes become available, click SP to obtain the Special Animation Icons palette.
3. Select the Car Inertia block so that you can see the exact block outlines. Drag the automobile icon until it is just centered over the Car Inertia block.



As you can see from the figure above, the icon, once dropped in place, may appear distorted.

4. 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, click the Icon tab, and observe the code produced by this action.

19.3 Icon Source File

The format for a source file containing a single icon appears in [Example 19-4](#) below. Keywords are specified in **ALLCAPS**. 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 is zero. Comments are shown in the normal text format.

The **IDENTIFICATION** shown in the third line of [Example 19-4](#) furnishes an abbreviation for the palette name ("MY" in the example), followed by a slash (/) separator and 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 ❸ in [Example 19-4](#)) are discussed in [19.3.1 Icon Identification](#). Sections are delimited with **BEGIN** and **END**; you need not specify all sections. Section syntax and parameter explanations are discussed in [19.3.6 Animation Statements](#).

Example 19-4 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/endo;
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 USRIAL interface (USRIAL
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

```

19.3.1 Icon Identification

This section lists the possible parameters for keywords in the identification portion, which immediately follows the **BEGIN_ICON** keyword (see ❶ on p.494).

Table 19-1 Parameters for the IDENTIFICATION Portion of the Icon Source File

Keyword	Parameter	Loc	Comments
ICON_TYPE	number	❶	The icon identification number for each icon. This number is used in the PALETTE_DEFINITION section. If you have changed the IDENTIFICATION field (p.494), and thus changed the name of the palette, you may use sequential integers (1...n) for your icons.
ICON_PRIVILEGE	privilege	❷	Normally set to 0.
ICON_WIDTH	width	❸	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.
ICON_HEIGHT	height	❹	The icon box height in pixels (100 ~ 1 cm).
XXX_VARIABLE	initial value	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 (see 19.3.2 Types).
XXX_VARIABLE	Form Prompt	❻	Initial value can be '##', where ## is the number of characters.

When defining your own icon, *do not use tabs to indent any portion of your code. Use spaces only.*

19.3.2 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.

Typically variables are declared at the top of the icon source file (see [Example 19-4](#)); each separate type is numbered sequentially starting with 1. For example, to define a string variable numbered 17, specify `S[17]`.

Hardcoded Integer Type

`I[#]` Integer variable

Hardcoded Real Types

`R[#]` Real variable

`A[#]` Animation pointer (to input vector)

`O[#]` Output pointer (to output vector)

`T[#]` Time & hold information (pointer to t-vector)

Hardcoded String Type

`S[#]` String variable

19.3.3 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 19-2 Control and Calculation Keywords

Syntax	Example
<pre>CALCULATE v1ptr = fun v2ptr v3ptr CALCULATE v1ptr = v2ptr</pre>	<pre>CALCULATE I[3] = I[4] + I[5] CALCULATE I[3] = I[4]</pre>
<pre>MATH_FUNCTION v1ptr = fun v2ptr v3ptr</pre>	<pre>MATH_FUNCTION R[1] = ATAN2 [R[2] R[3]</pre>
<pre>IF value relation value THEN ELSE ENDIF</pre>	<pre>IF I[3] < 10 THEN your_statements ELSE your_statements ENDIF</pre>
<pre>IF value relation value THEN RETURN ENDIF</pre>	<pre>IF I[2] EQ 1[5] THEN RETURN ENDIF</pre>
<pre>DO variable start end inc RETURN ENDDO</pre>	<pre>DO I[2] 1 5 1 your_statement ENDDO</pre>

CALCULATE functions use the pointers v1ptr, v2ptr, and v3ptr. Pointers can be reals, integers, or strings. You can use CALCULATE functions 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
```

MATH_FUNCTIONS are:

SIN	COS	TAN	SQRT	ABS
ASIN	ACOS	ATAN	ATAN2	

The ATAN2 function uses **v3ptr**.

```
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.

19.3.4 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 [19.3.5 General Graphic Characteristic Statements](#)). 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 [19-5](#) through [19-16](#).

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 19-5 **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 19-6 **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 [p.504](#).) The text is aligned on that point according to the **justify** parameter, a two-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 is drawn centered on the starting point and top-aligned, because a 2 is in the tens column and a 1 was in the ones column.

Example 19-7 **Graphics Erase Text Statement**

```
ERASE_TEXT x y justify strings
```

Erases a string created by **DRAW_TEXT** when given exactly the same parameters.

Example 19-8 **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 two-digit code value used for **DRAW_TEXT justify**).

Example 19-9 **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 19-10 **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 [p.504](#)) determines the fill pattern for the arc.

Example 19-11 **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 19-12 **Graphics Rotate Line Statement**

```
ROTATE_LINE ptr xc yc npoints x1 y1 ... xn yn
```

Performs the **DRAW_LINE** function and adds the ability to rotate the line centered on the point specified by *xc*. *yc*. *ptr* is the number of degrees of rotation counterclockwise from the horizon (that is, the positive part of the x-axis).

Example 19-13 **Graphics Relative Position Line Statement**

```
RELATIVE_POSITION_LINE xd yd npoints x1 y1 x2 y2 ...
```

Performs the **DRAW_LINE** function and adds the ability to place the line relative to the point specified by *xd* and *yd* (the coordinate displacement in IA graphical units).

Example 19-14 **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 [19.3.5 General Graphic Characteristic Statements](#).

Example 19-15 **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*; it 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 19-16 **Graphics Variable Position Line Statement**

```
VARIABLE_POSITION_LINE dir iptr x1 y1 v1p x2 y2 v2p
```

Performs the **VARIABLE_SIZE_BOX** function but draws a line in the defined area rather than filling a box.

19.3.5 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

Additional details follow.

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 dash
3 = dashed line	7 = close dots	10 = dotted
4 = close dash		

SET_LINE_WIDTH <i>width</i>	Specifies line width in pixels; the width of a normal line is 1 pixel.
SET_FILL_PATTERN <i>pattern</i>	Specifies the pixel density of a filled area. If 0 is supplied, there is no fill. If 2 is supplied the fill is 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 SET_FILL_PATTERN 55 , expect a density of 8 pixels.
SOUND_BELL <i>loudness</i>	Sets bell volume to an integer between 0 and 8, where 0 is silent and 8 is the loudest.
SOUND_KEY_CLICK <i>loudness</i>	Sets key click volume to a number between 0 and 8, where 0 is silent and 8 is the loudest
SET_ARC_TYPE <i>type</i>	Specifies fill type as 0, 1, or 2. 0 indicates empty, 1 indicates a filled arc, and 2 indicates pie fill.
SET_TEXT_FONT <i>font type</i>	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.
SET_TEXT_SLOPE <i>angle</i>	Sets the text slope with a number indicating the angle of the text in degrees counterclockwise from horizontal. The <i>angle</i> is a number representing the degrees of rotation.
SET_LINE_DISPLACEMENT <i>xd yd</i>	Moves a line drawn by GENERAL_LINE . This keyword displaces all points in the x and y directions by the number of IA graphical Units in <i>xd</i> and <i>yd</i> (default=0).

SET_LINE_ANGLE *angle x y*

Specifies the angle of a line drawn with **GENERAL_LINE**. 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.

SET_LINE_SCALE *xsc ysc xs ys*

Expands or shrinks a **GENERAL_LINE** 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, and so forth. *xsc* and *ysc* indicate the location of the scale center point; the default is 0 (centered).

19.3.6 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* and 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 is made a complementary color; if it is 1, the area flashes 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.

19.3.7 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 identifies the hot spot being pointed to. *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 ($H1-x2, H1-y2$).

INQUIRE ptr prompt	Creates a dialog box containing a <i>prompt</i> string you specify; <i>ptr</i> accesses what the user types into the dialog.
LOAD string	Loads a .pic file without bothering to keep track of previous .pic files
CHAIN_DOWN string	Loads a .pic file, keeping track of its position in a hierarchical stack.
CHAIN_UP	Reloads the last .pic file.

19.3.8 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 p.494)
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 <i>vars_diff</i> , 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.

19.4 Animation Configuration File

You can modify the supplied configuration file *SYSBLD/etc/animation.cfg* (shown in [Example 19-17](#)) 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 19-17 Typical Animation Configuration 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 last items listed are the only changes from the Release 6.0 distribution version of the file.

19.4.1 Important Animation Configuration Keywords for Customized Icons

This section describes the `animation.cfg` keywords you might need to alter:

ICON_DATA_FILE: 'myicons.sog'

You supply the compiled and translated source file that defines which icons are available in IA.

PROCESS_PICTURES: 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.

ICON_SOURCE_FILE: 'myicons.src'

Supply the name of the source file that 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.

BUILD_LOAD_PICTURE: 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 saves time if this is the primary icon you use.

SYSTEM_BUILD_RTF_FILE:

Give the default file you want supplied when you select **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.

19.4.2 Icon Source File for Customized Icons and New Palettes

The icon source file (identified by **ICON_SOURCE_FILE**) contains the keyword **IDENTIFICATION:**

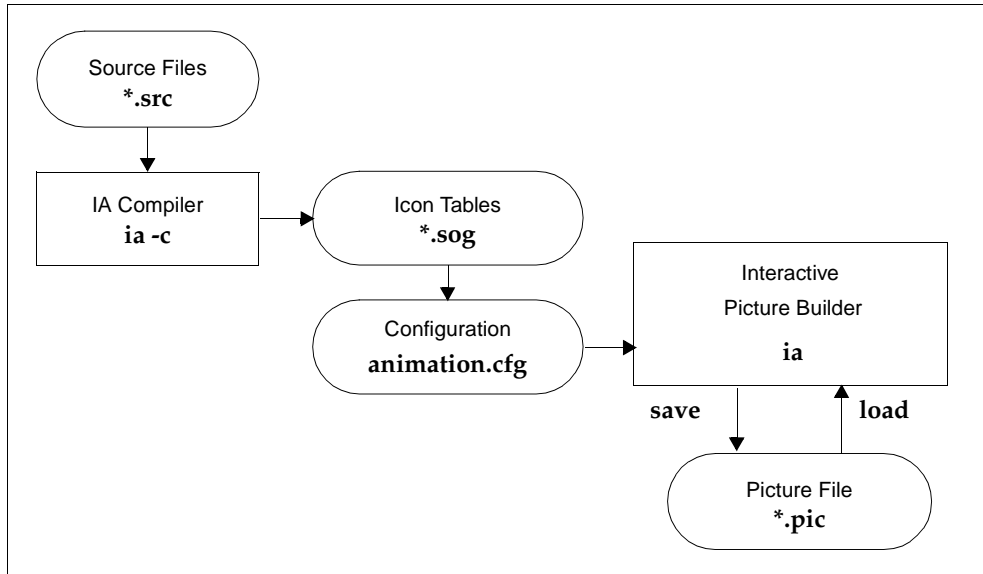
IDENTIFICATION: 'up to 30 Characters '

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 might encounter errors that are the result of conflicting **ICON_TYPE** numbers (see ❶ in [19.3.1 Icon Identification](#)).

19.5 Building Your Own IA Custom Icons

Figure 19-3 illustrates the steps for building custom IA icons. This process is summarized below; see the *Interactive Animation User's Guide* for a detailed explanation.

Figure 19-3 Building Custom IA Icons



To build custom IA icons:

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; for example, **infile.src**. The identification is the name of the new palette whose icons' source statements are contained in **infile.src**.
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. Create and debug the compilation of a single icon in a separate file before adding it to the palette **.src** file.

4. Add the icon to the palette `.src` file, and specify its position on a palette page using the **PALETTE DEFINITION** keyword at the end of the `.src` file.
5. 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*.

6. 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`.
7. 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.

20

Custom Palettes and 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 blocks predefined by Wind River, customized versions of predefined 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, SuperBlocks, components, and DataStores can become custom blocks.

Custom Palettes shows how to create and open a palette that uses predefined blocks. *Custom Blocks* shows how to create a custom block, including how to add special startup, Help, and bitmap files. Examples show how to incorporate custom blocks into custom palettes and then open them in the Palette Browser. *Supporting Commands and Functions* discusses SBA support for custom blocks and palettes.

20.1 Custom Palettes

By default, the Palette Browser displays all predefined 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, and so forth). You can create your own customized palette file that contains blocks or palettes organized as you see fit. You can add or delete customized palettes 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.

20.1.1 Creating Palette Files

A palette file is composed of lines of palette items. Each palette item points to a built-in block, a custom block, or another palette file. [Example 20-1](#) shows how to create a simple custom palette file.

Example 20-1 Trivial Custom Palette Example

To create a new palette that shows only the Dynamic blocks palette and the Gain block:

1. Create a **palettes** directory, and set your current working directory to it.
2. In the **palettes** directory, create a text file named **example.pal** that contains the following two lines:

```
blockdirectory="ISI_Gain" title = "My Gain"  
palettefile="$SYSBLD/palettes/dyn.pal" title = "My Dynamic"
```

Save the file.

3. Open the Palette Browser. Select File→Open. Select **example.pal**. Click OK.

The new palette appears under the heading **example**; click **example**, and the Wind River Gain block appears to the right.

4. Double-click the label **example**.

The Dynamic palette appears in the left pane.

5. Click **My Dynamic**.

Its blocks appear in the right pane.

6. Highlight **example**, and then select File→Close to close the palette.

The example's tree structure is very simple. A base node, *example*, has one node, *My Dynamic*, attached to it. [Example 20-2](#) shows how to create multiple levels in the Palette Browser hierarchy.

Example 20-2 Trivial Custom Palette Example with Nesting

To create multiple levels in the Palette Browser hierarchy:

1. In the current working directory (**palettes**), 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"
```

You created **example.pal** in [Example 20-1](#).

3. Open the Palette Browser.
 - a. Select File→Open. Select **levelone.pal**. Click OK.
The new palette appears with the heading *levelone*.
 - b. Double-click *levelone*.
A node with label *leveltwo* appears.
 - c. Double-click *leveltwo*.
The label *example* appears as a child of *leveltwo*.
 - d. Click *example*, and the Wind River Gain block appears to the right.
 - e. Double-click *example* (or click the + next to the label *example*) to see the Dynamic palette as a node below.
 - f. Click *My Dynamic*, and the dynamic palette blocks appear to the right.

You can call palette files recursively; for example, *leveltwo* can contain an entry for *levelone*. When recursion is detected, the Palette Browser displays a single level of recursion.

4. Highlight *levelone*, and then select File→Close to close the palette *example*.



NOTE: Close can only remove entire palettes; there is no way to close off *leveltwo* and keep *levelone*.

20.1.2 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 is:

```
blockdirectory = "BlockDirectory" title="block_title_on_palette"  
                help = "pointer_to_html_file" icon="pointer_to_bitmap_file"
```

```
palettefile = "pointer_to_palette_file" 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 [p.526](#).
- Block and palette statements may be interspersed in the file.
- Comments are not supported in custom palette files.
- If the help and icon files are specified in the creation of the custom block, the path is not required in the palette file. (See also [Step 2: Add a Custom Block to a Custom Palette File](#).)

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 = "trg.pal"
```

PaletteFile

palettefile points to a palette file. You can reference built-in palettes with the following **palettefile** values:

sysbld.pal	Includes all of the default SystemBuild palettes
sup.pal	SuperBlock
alg.pal	Algebraic
pwl.pal	Piece-wise Linear

dyn.pal	Dynamic
imp.pal	Implicit
trg.pal	Trigonometric
pel.pal	Power Exponential Logarithmic
trn.pal	Coordinate Transformation
sgn.pal	Signal Generator
log.pal	Logical
usr.pal	User Programmed
kbb.pal	Artificial Intelligence
ntp.pal	Interpolation
sc.pal	Software Constructs
mtx.pal	Matrix Equations
bst.pal	BetterState
iai.pal	Interactive Animation (requires separate license)
arc.pal	Archived (obsolete blocks)

In addition to Wind River palettes, **palettefile** can also be the path to a palette file. The path can be in any of the following formats:

1. An absolute path, such as **c:\palettes\my.pal** (Windows), or **/homes/SunPlatform/users/usr1/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 **PALETTE_PATH** is the path to a palette file.



NOTE: The UNIX style works inside SystemBuild for Windows, as well. If you are pointing to Wind River palette files, **PALETTE_PATH** is **SYSBLD**.

BlockDirectory

blockdirectory is used to specify native Wind River blocks and custom blocks (see [Custom Blocks](#).) To specify a default block, simply prepend **ISI_** onto the name of the block. See [Using Relative Paths for Icon Files](#) for hints on using graphics in a custom palette.

20.1.3 Defining the Default SystemBuild Palette

Instead of opening a custom palette file every time SystemBuild is started, you can define a file named **startup.pal** that is 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. The current working directory.
2. **%HOME%\xmath** on Windows or **\$HOME/xmath** 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. However, **\$SYSBLD** is predefined but is only available in the product.

4. The default SystemBuild palette directory, **%SYSBLD%\palettes** on Windows, or **\$SYSBLD/palettes** on UNIX.

[Example 20-3](#) shows you how to create a new startup palette file.

Example 20-3 Startup Palette File

To create a new startup palette file:

1. Copy the file **example.pal** (created in [Example 20-1](#)) to **startup.pal**.
2. Exit SystemBuild, and then restart it.

You can see that the label Main now contains only the blocks you defined in **startup.pal**. The default Wind River palette is not loaded.


3. If you want to add the default Wind River palette, add the following line to **startup.pal**, and then repeat [Step 2](#):

```
palettefile="$SYSBLD/palettes/sysbld.pal"
```


20.1.4 Closing and Reloading the Default Palette

You can remove and then reload the default palette at any time.

To close the default palette:

1. Highlight the label Main.
2. Select File→Close, or click the Close Palette toolbar button  to remove the palette.

To reload the default palette:

Select File→Load SystemBuild Palette.

20.2 Custom Blocks

This section describes how to create and instantiate custom blocks. A custom block can start as:

- One of the predefined blocks provided with SystemBuild
- A SuperBlock or component created from predefined 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.

20.2.1 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 can define any combination of labels, parameter values, data types 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 contains 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.

20.2.2 Creating a Basic Custom Block

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, open and display the custom palette file in the Palette Browser.

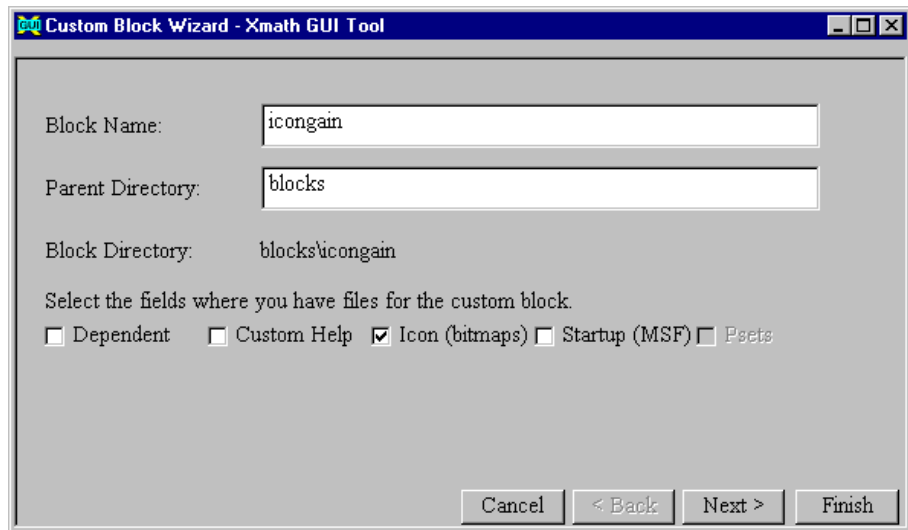
This section describes the procedure to create and use a basic custom block. Subsequent sections show how to create custom blocks with additional features.

To create and use a custom block:

1. In the SuperBlock Editor, create or select a block that has the properties you want to reuse.
2. Create a custom block using the Custom Block Wizard.
 - a. Select Edit→New Custom Block to bring up the Custom Block Wizard.

The Custom Block Wizard displays multiple panels to specify information about the files related to the custom block.

Figure 20-1 Custom Block Wizard Fields



By default, the Block Name field of the Custom Block Wizard shows the name of the block selected in the editor.

- b. Accept or change the Block Name field.

If no name is specified, the wizard displays the block type in the name field. The block name is used to name a subdirectory in the directory specified in Parent Directory.

- c. In the Parent Directory field, type in the location where the new custom block subdirectory is to be created. If you want your files to be portable, specify a relative path (as discussed in [Using Relative Paths for Icon Files](#) and as shown in [Figure 20-1](#)). Press Return to update the Block Directory field.

The default is the current Xmath working directory.

- d. Enable one or more checkboxes to indicate that you have auxiliary files you want included in the block.

- e. Click Next if you have auxiliary files; otherwise click the Finish button.

The Custom Block Wizard creates a new directory to contain any custom block files.

3. Add your custom block to a custom palette file (a text file with a **.pal** extension):

Using the syntax discussed in *Palette File Syntax*, define a single line in the custom block palette file for each new block; for example:

```
blockdirectory = "blocks/icongain" icon = "icongain.bmp"
```

4. Open the custom palette file in the Palette Browser.

[Example 20-4](#) shows you how to create a custom Gain block for your first custom block.

Example 20-4 Simple Custom Block Example

To create a custom Gain block:

1. Set your current working directory to the **palettes** directory that you created in a previous section.
2. 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.
3. Create a custom block using the Custom Block Wizard.

- a. Select the Gain block; then select Edit→New Custom Block.

The Custom Block Wizard appears.

- b. Deselect any of the checkboxes currently enabled in the files area.
- c. Enter **mygain** for the Block Name.
- d. Enter **blocks** for the Parent Directory.

If you do not have this directory, allow SystemBuild to create it for you.

- e. Click Finish.

The new custom block directories are created within the parent directory: **palettes/blocks/mygain**.

4. Create the palette file **example2.pal** that contains the following lines:

```
blockdirectory="blocks/mygain"  
blockdirectory="blocks/mygain" title = "gain100"
```

5. Start the Palette Browser, and open the file **example2.pal**.

Your new custom block palette appears under the label **example2**.

The custom block appears twice in the palette: the first block displays the name of the directory (**mygain**) because no title was specified. The second block displays the string specified with the **title** keyword (**gain100**).

6. Drag the custom block off the palette into the SuperBlock Editor, and verify that it has the same attributes as the block from which you made the custom block.

20.2.3 Creating More Sophisticated Custom Blocks

In addition to the basic features listed above, each custom block may optionally have defined a dependent file, a custom Help file, an icon that appears in the Palette Browser, an icon that appears in the SuperBlock Editor, a startup file, and a parameter set. You define all of these files, with the exception of the bitmap that appears in the editor window, in the Custom Block Wizard. See [Defining Custom SystemBuild Icons, p.489](#) and [Using Relative Paths for Icon Files, p.531](#) for specifying a custom icon for the SuperBlock Editor.

The general procedure for defining custom blocks is provided below.

Step 1: Create a Custom Block Using the Custom Block Wizard

1. Specify the Block Name and Parent Directory in the Custom Block Wizard. Enable the checkboxes in the lower section of the window that indicate the files that are associated with the custom block you are creating (Dependent, Custom Help, Icon (bitmaps), Startup (MSF), Psets). Click **Next >** to access the page for the first file type.
2. On the appropriate file list page, click the **Browse** button. This brings up the file dialog and allows you to select the file to be added to the list. Alternatively, type in the full path to the file directly in the Custom Block Wizard. Click the **Add** button to add this selected file to the list. Once you select the file to be added to the list and click **Add**, the full path name of the file is displayed in the Custom Block Wizard.

To delete a file from the list, single-click the file entry, and 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 = Path_BlockName [[keyword=value] [keyword=value]...]
```

The valid keywords are:

- title** Specifies the title of the block to be displayed in the Palette Browser.
- 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 custom 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 Browser. This file can be platform dependent. If the filename is specified as *name.platform*, then the name is automatically interpreted to be *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"
```

Use a relative path if portability is required (see [Using Relative Paths for Icon Files](#) on p.531).

Step 3: Open the Custom Palette File

Open the custom palette file in the Palette Browser.

20.2.4 Including a Startup MathScript File with the Custom Block

When a custom block is instantiated, an optional MathScript function defined for the custom block in the Custom Block Wizard can execute. This MathScript function (MSF) file must have the same name as the custom block. For example, if the custom block is called `csi2`, then this MSF file must be called `csi2.msf`. The MSF provides 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, block_id)
    ....function body.....
endfunction
```

See [SysbldEvent](#).

We'll now extend the simple gain custom block to include a startup file via [Example 20-5](#); the startup file simply changes the color of the block based on user input.



NOTE: The startup file and the custom block must have the same root name.

Example 20-5 Custom Block Example with Startup MathScript File

To include a startup MathScript file:

1. In your local directory, create a text file named `startgain.msf` that contains the following lines:

```
function [result] = startgain(sbname, block_id)
    color = getchoice("Choose a color:", ["red", "green"]);
    modifyblock block_id, {color = color};
endfunction
```

This function raises a dialog to prompt you for a block color whenever the `startgain` custom block is dragged from 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 **200**.
3. Select the Gain block, and then select Edit→New Custom Block.

The Custom Block Wizard appears.

- a. Enter `startgain` for the Block Name. Enter a valid name for the Parent Directory (for example, `blocks`).
- b. Enable the Startup (MSF) checkbox and disable all other file checkboxes.

- c. Click Next.
 - d. In the Startup Files dialog, use the browser or enter the path directly to the file **startgain.msf**. Select Add to complete the selection.
The full path name of the file appears in the lower window.
 - e. Click Finish.
The new custom block directories are created within the parent directory.
The file **startgain.msf** is also copied to the custom block directory.
4. Edit an existing palette file (**example.pal**) to include startgain:

```
blockdirectory = "blocks/startgain" title = "gain200"
```
 5. Start the Palette Browser, and open the palette file **example.pal**.
Your new custom block file appears under the label example.
 6. Click example, and drag the custom block off the palette into the SuperBlock Editor.
A small Xmath dialog appears.
 7. Enter one of the options, and click OK.
The Gain block with the appropriate color is drawn in the editor window.

20.2.5 Including a Custom Help File with the Custom Block

HTML files that form the custom Help for the custom block can be specified in the Custom Block Wizard. Since these files are platform-independent, the Platform combination box is not 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](#).

We'll now extend the simple gain custom block to include a Help file via [Example 20-6](#).

Example 20-6 **Simple Custom Block with Help**

To include a Help file with the custom block:

1. In your current directory, create 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 must be **.html** because **.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: Name = helpgain, Gain = 99.
3. Select the Gain block, and then select Edit→New Custom Block.

The Custom Block Wizard appears.

- a. Enter **helpgain** for the Block Name. Enter a valid name for the Parent Directory (for example, **blocks**).
- b. Enable the Custom Help checkbox and disable all other file checkboxes.
- c. Click Next.
- d. In the Startup Files dialog, enter the path to the file **helpgain.html** (or use the browser). Select Add to complete the selection.
- e. Click Finish.

The new custom block directories are created within the parent directory. Your simple Help file is copied to the **Help** subdirectory in the custom block directory.

4. Modify or create the palette file **example.pal**. Add the following line to the file:

```
blockdirectory = "blocks/helpgain" title = "gain99"
```

5. Start the Palette Browser, and open the file **example.pal**.
Your new custom block appears under the label example.

6. Click `example`, and drag the custom block off the palette into the SuperBlock Editor.
7. Open the Block dialog for the custom block. Click `Help`.

Your HTML Help text is displayed in a local Help window.

20.2.6 Additional Custom Block Features

As mentioned above, each custom block may optionally have defined a dependent file, a custom Help file, a bitmap that appears in the Palette Browser, a bitmap that appears in the editor, a startup file, and a parameter set. We have provided you examples of creating a simple startup file and a simple custom Help file. The process for providing the remaining files in the Custom Block Wizard is the same; therefore, we simply define the other files that you can define in the wizard below.

Dependent File

The files that are associated with a UserCode block (UCB), a BlockScript block, and so forth can be specified in the Custom Block Wizard. You can specify the platform dependency by using the Platform combination box to specify the appropriate platform for the selected files.

Icon for the Palette Browser

You specify the icon file(s) used for the custom block while it is in the Palette Browser in the palette file with the `icon` keyword (see [Palette File Syntax](#)). You can use the `Icon` option in the Custom Block Wizard to copy the correct icon file(s) to the custom block directory. When you reference this icon in the palette file with the `icon` keyword, no path is necessary because it is copied to the correct location.

The Platform combo box in the Custom Block Wizard is disabled for icon files. Files with the extension `.bmp` are used in the Windows environment, while files with the extension `.xpm` are used on all UNIX platforms. If the file has the extension `.platform`, then SystemBuild automatically interprets the extension as `.bmp` on Windows platforms and `.xpm` on UNIX platforms.

Thus, if you are using SystemBuild on more than one platform, supply both `myicon.bmp` and `myicon.xpm` in the wizard. Then use `icon = "myicon.platform"` in the palette file.



NOTE: Custom icons created with IA do not display on the palette.

Parameter Sets

One or more files containing values pertinent to the block. Parameter sets allow you to select different prespecified values for the block. A custom block can have multiple parameter sets.

20.2.7 Using Relative Paths for Icon Files

We begin this section with a caution, which defines the reason for this section:

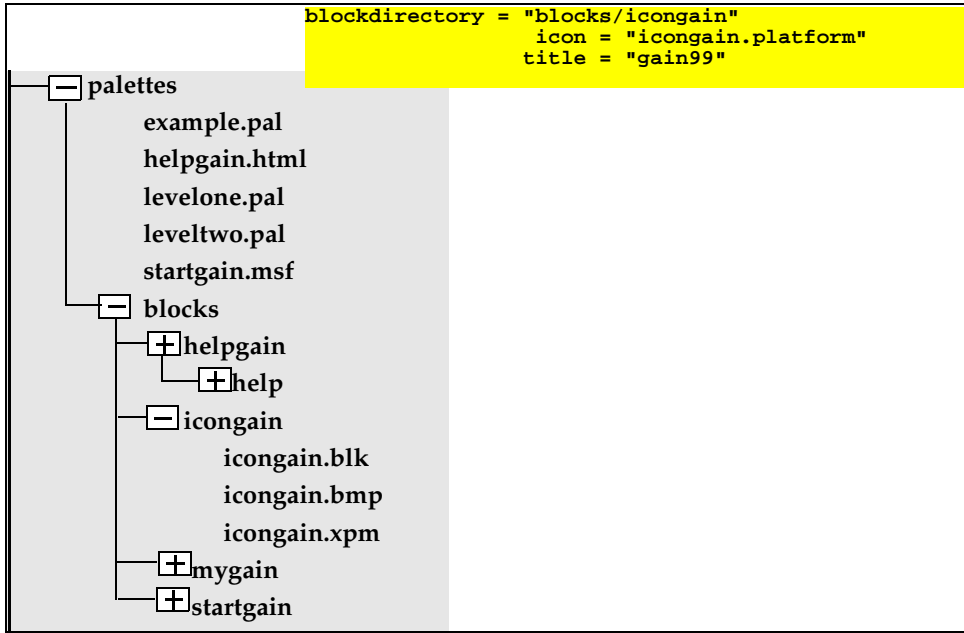


CAUTION: Although the SuperBlock Editor displays all supported bitmaps on any SystemBuild platform, the Palette Browser does not. Furthermore, icon file paths in palettes are platform dependent. If you have an incompatible path in a palette file, you will not be able to instantiate your block in the SuperBlock Editor because the icon file cannot be found.

In *Icon for the Palette Browser*, we discuss providing icons for both UNIX and Windows in the Custom Block Wizard and then using the **.platform** extension in the palette file to specify the icon so that the same palette file can be used on multiple platforms. You can use the same icon file for display in the editor and the palette as long as relative paths are used. (Forward slashes are acceptable for either Windows or UNIX in this context.)

For example, look at the directory structure in [Figure 20-2](#).

Figure 20-2 Sample Directory Structure with Accompanying Palette File



The current working directory is **/palettes**, which you set in Xmath. The palette file (**example.pal**) is in the current directory, and a custom block resides in a subdirectory. Both the custom block and the palette file (shown in the yellow overlay) can use a relative path to refer to the icon. This hierarchy works across platforms whenever the top of the hierarchy (in this case **/palettes**) is the current working directory.



NOTE: Once you create the custom block, the bitmap file is copied to the block directory and that image is used in the Palette Browser. If you no longer need the original file, you can delete it.

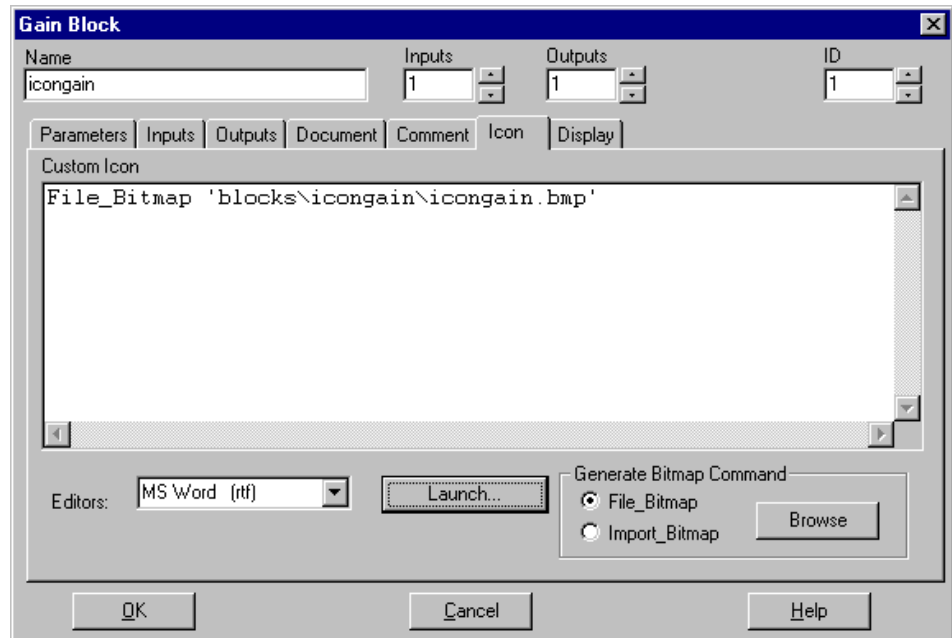
The icon that appears for the block in the SuperBlock Editor is defined in the block dialog for the block; you can define a custom icon on the Icon tab and set the Icon Type to Custom Icon on the Display tab. If you want to use the same image in the Palette Browser and in the SuperBlock Editor, you can use the same file. If the custom block imports a bitmap image, there is no cross-platform restriction; SystemBuild displays any of the accepted bitmaps (a **.gif**, **.jpg**, **.bmp**, or **.xpm** file)

on any platform. Therefore, you can use either bitmap file for the editor; by using a relative path, you can use the same custom block on any platform. Figure 20-3 shows the bitmap icon being specified for the SuperBlock Editor using a relative path. You need to set these attributes before you create the custom block.



NOTE: If you use the browser to find the file, your path is provided as an absolute path. You then have to edit the path to get a relative path for use on all platforms.

Figure 20-3 Sample Custom Icon Specification for the SuperBlock Editor



20.3 Supporting Commands and Functions

You can also manipulate custom blocks with commands and functions issued from the Xmath command area.

20.3.1 SystemBuild Access Support

The following SBA commands accommodate custom blocks and palettes. For a complete description of each command, see 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. |

20.3.2 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.

Inputs are as follows:

- | | |
|--------------|--|
| event | The SystemBuild event to be associated with this action; 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 <i>event</i> parameter; 0 disables the event. If you are enabling an event and you have not specified a function to handle the event, then the <i>MSFName</i> parameter is required.
MSFName	A string containing the name of an Xmath MSF that handles the specified <i>event</i> . The Xmath MSF must accept two parameters: <i>block_id</i> and <i>event</i> , where <i>block_id</i> is the block identification number of the block selected by the editor and <i>event</i> is the same event specified by the calling SysbldEvent() function. 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 **UserBlockOpen()** MSF, you can use SBA to query or modify the model. Then, if the MSF returns 0, the editor displays the dialog. If the MSF returns 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. You can also use **SysbldRelease()** if the user interface becomes locked and does not release during **SysbldEvent()** operations.

Bibliography

- [A80] Amit, N., "Optimal Control of Multirate Digital Control Systems," Ph.D. thesis, Stanford University, SUDAAR #523, July 1980.
- [BCP89] 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.
- [BG73] R.L. Brown and 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.
- [BK93] 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.
- [CdeB80] Conte, S.D., and C. deBoor, *Elementary Numerical Analysis*, McGraw- Hill, New York, 1980.
- [DFT92] John C. Doyle, Bruce A. Francis, and Allen R. Tannenbaum, *Feedback Control Theory*, Chap 3, MacMillan Publishing Company, New York, 1992.
- [F62] Fox, L., *Numerical Solution of Ordinary and Partial Differential Equations*, Pergamon Press, 1962.
- [FL91] Führer, C., and B. J. Leimkuhler, "Numerical solution of differential-algebraic equations for constrained mechanical motion," *Numerische Mathematik*, Vol 59, pp. 55-69, 1991.
- [G71a] Gear, C. W., *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice Hall, Englewood Cliffs, N.J., 1971.

- [G71b] Gear, C. W., "Simultaneous Numerical Solutions of Differential-Algebraic Equations", IEEE Transactions on Circuit Theory, CT-18, January 1971.
- [G83] Glasson, D.P., "Development and Application of Multirate Digital Control," Control Systems Magazine, November 1983.
- [H73] Hamming, R.W., *Numerical Methods for Scientists and Engineers*, McGraw Hill, New York, 1973.
- [KMN89] Kahaner, D., C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice Hall, Englewood Cliffs, N.J., 1989.
- [McL50] McLachlan, N.W., *Ordinary Non-Linear Differential Equations in Engineering and Physical Science*, Oxford University Press, Glasgow, 1950.
- [M70] Milne, E.M., *Numerical Solution of Differential Equations*, Dover Publications, New York, 1970.
- [P83] Petzold, L.R., "A Description of DASSL: A Differential/ Algebraic System Solver," in *Scientific Computing*, pp. 65-68, eds. R.S. Stepleman et al, North Holland, Amsterdam, 1983.
- [PFTV89] 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.
- [SG79] 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.

Index

Symbols

`$(SYSBLD)/etc` directory 488
`%Variables` 176, 178

A

actiming 173
Adams-Bashforth-Moulton integration 292
advanced load 9
algebraic loop 182, 280
 and trim function 242
 detected 185
 initial conditions for 185
 integration algorithms for 183
analyze function 185
 Editor tool 188
 from SystemBuild editor tool 188
 items displayed 186
 running by simulator 186
 sbInfo 186
 with implicit outputs 274
animation.cfg
 BUILD_LOAD_PICTURE 511
 ICON_DATA_FILE 510
 ICON_SOURCE_FILE 510
 PROCESS_PICTURES 510
 SYSTEM_BUILD_RTF_FILE 511

animation.cfg file, IA 508
asynchronous trigger SuperBlock 136
AutoCode
 BetterStateChart blocks 396
AutoSave capability 20
auxiliary constraints 273

B

background
 procedure SuperBlock 129
 simulation 194
basic time period 230
BetterState block
 example usage 380
 procedural
 conditional logic example 383
 types of models for use 380
BetterState chart
 actions, using procedure SuperBlocks 377
 create new 25
 creating reference to 374
 open 22
 using SystemBuild variables in 377
 See also *BetterStateChart* block
BetterStateChart block 373
 AutoCode
 event-driven chart 396

- procedural chart 396
- control implementation 374
- simulation
 - event-driven chart 396
 - procedural chart 396
- usage in SuperBlocks 374
- blocks
 - BetterStateChart 373
 - Code tab 85
 - comment editor 481
 - create (Palette Browser) 72
 - creating 72
 - defining 72
 - described in online help 70
 - icon
 - colors 92
 - types 93
 - ID
 - determined by location 164
 - number 74
 - inputs 74
 - name 74
 - outputs
 - labels 87
 - names 87
 - type name, block dialogs 87
 - parameters, specifying 85
 - special behaviors 70
 - states field 74
 - supported in RVE 219
 - tabs 86
 - unsupported in RVE 220
 - UserCode 323
 - using matrix editor 95
- BOOLEAN data type 105
- Break block 71
- build 3
- building IA custom icons 512

C

- catalog 7
 - browsing 14

- Catalog Browser
 - Catalog view 14, 17, 29
 - Contents view 15, 16
 - drag copies 29
 - sort 17
 - creating SuperBlock reference 59
 - dragging and dropping objects 29
 - modifying catalog 27
 - navigating 14
 - Quick Access menu 21
 - saving selected data 19
 - Shortcut menu 21
 - tools 31
 - transform SuperBlock tool 150
 - using advanced load 8
- chart animation, using in BetterState 386
- chart, BetterState
 - See also *BetterStateChart* block
 - See *BetterState* chart
- circled
 - bar, chart animation 386
- classical analysis tools 245
- color assignment (UNIX) 485
- colors, block icons 92
- comment editor 481
- compilers 357
- component 13, 449
 - %Variables 458
 - creating 457, 461
 - encrypted 452
 - encryption 466
 - exported variables 460
 - interface 451
 - licensing 453
 - mapping parameters 471
 - modifying 462
 - open 452
 - parameter mapping 460
 - parameters 455
 - PSET 463
 - references 451
 - restrictions 458
 - SBLIBS 465
 - scope 459
 - unmake 463

- computational attributes 223
- Condition block 70, 126
 - can contain Standard procedure 127
 - procedures referenced from 126
- Connection Editor 79
 - Add button 79
 - Cancel button 81
 - Del button 80
 - Done button 81
 - routing 113
- connections 75
 - display 81
 - rules 75
- constrained DAEs 295
- constraints
 - auxiliary 273
 - in implicit models 273
 - required 273
- continuous subsystem 168
 - linearizing 225
 - TimeDelay block, linearizing 228
- continuous SuperBlocks 123
- control.sog, IA control panel file 489
- copy
 - SuperBlock (copy and paste) 52
 - SuperBlock (rename) 65
 - SuperBlock (via SuperBlock properties) 53
- CREATERTF 188
- CREATEUSERTYPE 444
- Ctrl-p 176
- custom
 - block 521
 - add to custom palette file 524, 526
 - creating 522
 - SBA support 534
 - specifying associated files 525
 - using Custom Block Wizard 522
 - dialog, component 474
 - icons
 - adding to block diagram 488
 - building 512
 - keywords 510
 - palette 516
- custom block 521
- customizing SystemBuild 478

D

- DASSL 183, 290, 291
- data types 57, 87, 103
 - classes of data 104
 - floating point 105
 - input/output data mismatch 107
 - integer 105
 - logical 105
 - RT_BOOLEAN 105
 - RT_FLOAT 105
 - rules 108
 - type-checking issues 107
- DataPathSwitch block 71
- DataStores
 - in catalog 12
 - timing 144
 - timing features 144
 - using 131
- DELETEUSERTYPE 444
- demo, running 4
- Differential Algebraic Equations (DAEs) 284, 325
- differential equation solvers 179
- direct terms 183
 - with UCBs 335
- discrete
 - subsystems 124
 - linearizing 227
 - types 124
 - SuperBlocks 124
- drag and drop, to attach an IA icon to a block 488

E

- EDIT_COMMENT environment variable 480
- Editor
 - adding custom menus 481
 - block label names 102
 - changing font size 114
 - classical analysis tools 245
 - color assignment (UNIX) 485
 - connections 76
 - creating
 - blocks 72



- custom menus 481
 - SuperBlock reference 61
- default window position 486
- Display toolbar 112
- force catalog update 30
- modifying block diagram 113
- resizing and repositioning the display 486
- scope 23
- shortcuts 112
- SuperBlock transform tool 150
- enabled periodic subsystem 168
- event-based controller
 - BetterStateChart block example 387
- event-driven BetterState chart 374
- exact linearization 228
- example
 - BetterState chart event-based controller 387
 - BetterState chart MegaTest2 383
 - limited slider model 392
 - plant, piecewise-linear behavior 392
 - using **typecheck** 107
- exiting SystemBuild 4
- explicit
 - models 272
 - states 273
 - UCB 324

F

- File SuperBlocks 66
- finite difference linearization 228
- fixed-point
 - 32-bit operations 433
 - arithmetic 399
 - addition and subtraction 405
 - division 406
 - comparing with floating point 416
 - compatible blocks 399
 - with data types 424
 - data type
 - conversion 404
 - rules 424
 - Gain block radix position 434
 - linearization 446

- multiplication 406
- number representations 401
- overflow 408, 411
- relational operations 407
- simulation, ITypes 427, 429
- with simout 446
- fixpt 106
- floating point
 - data type 105
 - greatest common divisor 170
- functional blocks 48
- FuzzyLogic block, linearizing 229

G

- Gear's Method (GEARS) 301
- Group ID 169

H

- hierarchy of SuperBlocks 48

I

- IA 512
 - Builder
 - icon source code syntax 512
 - palette files 488
 - compiler 513
 - icon source file 494
- icon definition
 - ANIMATION_GRAPHICS 495
 - ANIMATION_POINTER 494
 - BACKGROUND_SECTION 495
 - DO 499
 - DRAW_ARC 501
 - DRAW_LINE 501
 - DRAW_RECT 500
 - DRAW_TEXT 500
 - ERASE_VALUE 501
 - FORM_DEFINITION 495

- GENERAL_LINE 502
 - ICON_HEIGHT 494
 - ICON_PRIVILEGE 494
 - ICON_WIDTH 494
 - IF/THEN 499
 - INITIALIZATION_SECTION 495
 - INTEGER_VARIABLE 494
 - MATH_FUNCTION 499
 - OUTPUT_POINTER 494
 - PALETTE_DEFINITION 495
 - REAL_VARIABLE 494
 - RELATIVE_POSITION_LINE 502
 - ROTATE_LINE 502
 - STATE_POINTER 494
 - STATIC_GRAPHICS section 495
 - STRING_VARIABLE 494
 - types 497
 - icon source file, IA 494
 - ICON_SOURCE_FILE keyword definition 510
 - IconScript keywords 499
 - ID number, block 74
 - IfThenElse block 70
 - implicit
 - models 272
 - outputs 273, 274
 - states 273
 - UCB 325
 - initial condition transformations 152
 - inline procedure SuperBlocks 128
 - input
 - data types, how set 104
 - names, block dialog 86
 - signals, block dialog 86
 - vector, u 193
 - integer data type 105
 - integration algorithm 285
 - Adams-Bashforth-Moulton 304
 - Adams-Moulton 292
 - DASSL
 - estimating err 303
 - infinity norm 303
 - solving DAE 295
 - default 285
 - Euler 286
 - fixed-step Kutta-Merson 288
 - GEARS 301
 - solving DAE 295
 - ODASSL 294
 - estimating err 303
 - solving DAE 295
 - QuickSim 292
 - Runge-Kutta 287, 288
 - selecting 179
 - stiff system solver (DASSL) 290
 - operating point 290
 - variable-step Kutta-Merson 289
 - Integrator block functionality 315
 - interactive
 - root locus tool 260
 - simulation. See *ISIM*
 - Interactive Animation (IA) 487
 - interrupt procedure SuperBlock 129
 - IS_Jacobian 369
 - ISIM 201
 - and RVE 213
 - chart animation 386
 - color settings 485
 - compared to IA 202
 - debugging 209
 - default
 - icon palette 203
 - window position 486
 - example 204
 - icon update times 210
 - monitoring block outputs 210
 - pause 209
 - resume 209
 - run in background 218
 - scaling aid blocks 447
 - terminating 196
 - view output labels 210
 - window 208
 - isim.sog, icon file 489
 - iusr01 template 329
- ## K
- Kalman-Bertram State-Space Method 223

L

labels

- creating 97
- defined at source 97
- initializing matrix 100
- matricizing 100
- propagating 92
- shortcuts for editing 102
- showing 91, 92
- SuperBlock external inputs 98
- vectoring 99
- with matrices 100

Lagrange Multipliers 296

limited slider model 392

linearization

- at a non-zero time 230
- continuous time delay 228
- exact vs. finite difference 228
- FuzzyLogic Block 229
- implicit form 224
- in fixed-point mode 446
- Kalman-Bertram state-space method 230
- multirate 230
- procedure SuperBlocks 127, 229, 245
- resettable integrator 229
- single rate 225
 - continuous 225
 - discrete 227
 - explicit 225
 - implicit 226
- State Transition Diagram 229
- subsystem method 231
- UserCode block 229

linksim 355

LISTUSERTYPE 444

LOAD 8

load, advanced 9

logical data type 105

M

macro procedure SuperBlock 127

matrix editor, SystemBuild 95

menus, custom 481

MIL-STD-2167A, comments documents 481

MinMax

dataset, save 440

display 440

logging 439

restrictions 439

MODIFYUSERTYPE 444

multirate linearization 230

N

name

block 74

SuperBlock 53

names

creating 97

initializing matrix 100

matricizing 100

vectoring 99

with matrices 100

namespace 450

O

ODAE 284

ODASSL 183, 294

ODASSL with implicit UCB 295

ODE 284

open-loop frequency response tool 248

operating point

DASSL 344

Jacobian matrix computation 345

ODASSL 344

output data types, how set 104

overdetermined stiff system solver (ODASSL) 294

P

Palette Browser, using 72

palette file 516

- block directory 520
- built-in 518
- default 520, 521
- IA 488, 511
- multiple level 517
- syntax 518
- palette, custom 516
- PALETTE_PATH 520
- panning the display 112
- parameter
 - root locus tool 266
 - set. See *PSET*
 - variable scoping 177
- parameterized variables 176
- plant, piecewise-linear behavior
 - BetterStateChart block example 392
- point-to-point frequency response tool 255
- printer settings 480
- procedural BetterState chart 374
- procedure
 - subsystem 168
 - SuperBlocks 126
 - background 129
 - in linearization 229
 - inline 128
 - interrupt 129
 - linearization 127, 245
 - macro 127
 - standard 127
 - startup 129
- procedure SuperBlock
 - navigating to from BetterState 380
- propagate labels 92
- PSET 463
 - example 469
- Psets_AddToList 464
- Psets_Load 464
- Psets_Save 464

Q

Quick Access menu. See *Shortcut menu*

R

- ReadVariable block 127
- reference 48
 - component 454
- renaming, SuperBlock 64
- repositioning the display 486
- resettable integrator, linearization 229
- resume, simulation
 - warning, BetterState charts 396
- root locus tool 258
- RT_BOOLEAN data type 105
- RT_FLOAT data type 105
- RT_INTEGER data type 105
- RTF (real-time file), creating 188
- running simulations 167
- Run-Time Variable Editor. See *RVE*
- RVE 211, 214
 - commands and functions 218
 - invoking 214
 - supported blocks 219
 - unsupported blocks 220

S

- SAVE 18
 - AutoSave feature 20
 - usertype keyword 445
- sb_uext reserved variable 247
- SBA 153
 - command syntax 155
 - error handling 161
 - function syntax 156
 - keyword formats 162
 - sample scripts 159
 - specifying mimo names 162
 - updating Editor display 158
- SBLIBS 66, 67
- sbsim command 196
- scheduler 170
 - minor cycle 145
- Sequencer block 71
- SETSBDEFAULT 191, 285
 - autosavefile 20

- autosavetime 20
- Shortcut menu, Catalog Browser 21
- shortcuts 84
- SHOWSBDEFAULT 285
- signal 97
- sim function 192
 - bg 194
 - function syntax 192
 - iahold 208
 - initmode 280
 - interact 207
 - PDM outputs 193
 - sbsim 196
 - sbview 207
 - ucbcode loc 356
 - vars 176
- SIMAPI 361
 - debug example 370
 - debugging simulation 366
 - source files 362
 - UCB reference information 362
 - variable access 363
- simAPI.h 362
- SIMAPI_FlushVars 365
- SIMAPI_GetBlockId 363
- SIMAPI_GetBlockInputLabel 362
- SIMAPI_GetBlockInputType 362
- SIMAPI_GetBlockName 362
- SIMAPI_GetBlockOutputLabel 362
- SIMAPI_GetBlockOutputType 362
- SIMAPI_GetDefaultOutputLabel 362
- SIMAPI_GetExternalInputDimension 367
- SIMAPI_GetExternalInputName 367
- SIMAPI_GetExternalInputValue 368
- SIMAPI_GetExternalOutputDimension 367
- SIMAPI_GetExternalOutputName 367
- SIMAPI_GetExternalOutputValue 368
- SIMAPI_GetImplicitOutputDimension 367
- SIMAPI_GetImplicitOutputName 367
- SIMAPI_GetImplicitOutputValue 368
- SIMAPI_GetImplicitSolverJacobian 368
- SIMAPI_GetNumVars 364
- SIMAPI_GetOperatingPointJacobian 368
- SIMAPI_GetSBName 363
- SIMAPI_GetSimStatus 369
- SIMAPI_GetStateDerivativeValue 368
- SIMAPI_GetStateDimension 367
- SIMAPI_GetStateName 367
- SIMAPI_GetStateValue 368
- SIMAPI_GetUCBBlockInfo 362
- SIMAPI_GetVarData 365
- SIMAPI_GetVarDatatypeName 364
- SIMAPI_GetVarDimension 364
- SIMAPI_GetVarIndexByName 365
- SIMAPI_GetVarName 364
- SIMAPI_GetVarPartition 364
- SIMAPI_GetVarStorageSize 364
- SIMAPI_GetVarUsertypeName 364
- SIMAPI_InitializeUserDebug 367
- SIMAPI_IsVarEditable 364
- SIMAPI_PutVarData 365
- SIMAPI_ResetVar 365
- SIMAPI_TerminateUserDebug 367
- simout function 189, 295
- simout function, with fixed-point 446
- simulation 167
 - abort 197
 - API. See *SIMAPI*
 - background 194
 - BetterStateChart blocks 396
 - DataStores in 132
 - errors 197
 - extracting values (simout) 189
 - fixed-point intermediate types 427
 - from Editor 191
 - from the OS command line 195, 196
 - initial conditions 282
 - initialization mode 280
 - input vector (u) 193
 - interactive 201
 - maximum integration step size 305
 - operating point 280
 - changing 282
 - continuous subsystem 280
 - discrete subsystem 280
 - resume
 - warning, BetterState charts 396
 - scheduler 170
 - See *sim function*
 - states 273

- stopping background job 194
 - termination 196
 - time vector (t) 193
 - timing properties 132
 - with ISIM 211
- single-rate systems, linearizing 225
- source palette files, IA 488
- standalone simulation (sbsim) 196
- starting SystemBuild 3
- startup custom palette file 520
- startup procedure SuperBlock 129
- startup.pal 520
- state event
 - continuous system 313
 - UCB 338
 - continuous 318
 - ZeroCrossing block 314
- State Transition Diagrams 12
 - linearizing 229
- statechart. See *BetterState chart*
- stdwrt 359
- stiff system solver
 - DASSL 290
 - ODASSL 294
- Stop block 71
- subsystem
 - continuous 123
 - discrete 124
 - enabled periodic 172
 - free-running periodic 172
 - trigger 134, 173
 - asynchronous 147
 - priority 171
 - priorities 171
 - procedure 168
 - processor
 - Group ID 169
 - pseudo-rate sample interval 170
- SuperBlock 11
 - Attributes tab 55
 - Block Properties dialog 61
 - Code tab 56
 - Comment tab 58
 - continuous 123
 - copy, paste, and rename 27
 - create 23, 25, 50
 - from existing blocks 50
 - Document tab 57
 - documentation generation 481
 - Editor coordinates 163
 - Editor. See *Editor*
 - expand 51
 - File 66
 - Inputs tab 56
 - instances 48
 - label names 98
 - editing 102
 - vectoring 99
 - label, propagate 63
 - name 53
 - open 22
 - output labels 57
 - output names 57
 - procedure 126
 - background 128
 - execution sequence 141
 - inline 128
 - interrupt 129
 - macro 127
 - navigating to from BetterState 380
 - standard 127
 - startup 129
 - use in BetterState chart 377
 - properties 53
 - defining 53
 - Properties dialog
 - SuperBlock transformation from 151
 - reference 48, 53, 61
 - in catalog browser 49
 - in Editor 61
 - rename 64
 - top-level 48, 53
 - transforming 31, 149
 - undoing 152
 - triggered 125
 - syntax rules, IA icons source code 512
 - SysbldEvent 534
 - SysbldRelease 535
 - SystemBuild
 - Access. See *SBA*

- demo, running 4
- exiting 4
- launching 3
- resource file 484
- starting 3

T

- text editor, changing default 480
- time response tool 252
- time vector (t) 193
- timing attributes, UCB 340
- top-level SuperBlock 53
- transforming
 - gain block 149
 - integrators 149
 - PID controller 149
- triggered subsystem 168
- trim 237, 238, 240
 - for systems with algebraic loop 242
 - free integrators 242
- typecheck**, usage in example 107
- TypeConversion block, usage 105

U

- UCB 323
 - code location 356
 - compile and link 356
 - compilers, supported 357
 - continuous 318
 - debug example 370
 - debugging 366
 - user code 357
 - direct terms 335
 - error messages, how to generate 359
 - execution order 339
 - explicit 324
 - implicit 325
 - with sim, lin, or simout 344
 - in SystemBuild vs. AutoCode 323
 - initialization 340

- interface 324
- linearization 229
- makefile 355
- modes
 - EVENT 328
 - INIT 327
 - LAST 329
 - LIN 328
 - MONIT 328
 - OUTPUT 327
 - STATE 327
- multi-use 353
- numerical integration algorithm 342
- programming considerations 353
- See *variable interface UCB* 346
- specifying source code 355
- state events 318, 338
- templates 329
- timing attributes 340
- variable names in 353
- updating Catalog Browser 30
- user
 - initialization file 478
 - parameters 89
- user.ini 478
- UserCode block. see *UCB*
- UserCode function 325
 - arguments 329
 - IINFO vector 331
 - mode parameters 333
 - RINFO vector 332
- user-defined data types. See *UserType*
- UserType 400
 - create 442
 - delete 443
 - editor 442
 - modify 443
- usr01 template 329

V

- variable
 - blocks 127
 - component mapping 460

- global, using in BetterState chart [377](#)
- interface UCB [346](#)
 - C wrapper required [347](#)
 - C wrapper, writing [347](#)
 - specifying interface [350](#)
- parameterized [176](#)
- scoping [177](#)
- variable, global, using in SystemBuild [377](#)
- variable-step integration [292](#)
 - Adams-Moulton method [292, 304](#)
 - Kutta-Merson method [303](#)

W

- While block [71](#)
- WriteVariable block [127](#)

