MATRIXx™

SystemBuild™ BlockScript User Guide

NATIONAL INSTRUMENTS™

April 2004 Edition
Part Number 370862A-01

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

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

Worldwide Offices

Australia 1800 300 800, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599, Canada (Calgary) 403 274 9391, Canada (Ottawa) 613 233 5949, Canada (Québec) 450 510 3055, Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530, China 86 21 6555 7838, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, Greece 30 2 10 42 96 427, India 91 80 51190000, Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400, Malaysia 603 9131 0918, Mexico 001 800 010 0793, Netherlands 31 0 348 433 466, New Zealand 0800 553 322, Norway 47 0 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210, Russia 7 095 783 68 51, Singapore 65 6226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 2 2528 7227, Thailand 662 992 7519, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on the documentation, send email to techpubs@ni.com.

Important Information

Warranty

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

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

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

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

Copyright

Trademarks

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

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

Patents

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

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

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

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

Conventions

The following conventions are used in this manual:

» — The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.

This icon denotes a note, which alerts you to important information.

This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.

bold — Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.

italic — Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.

`monospace` — Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.

`monospace italic` — Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

Contents

Chapter 1
Introduction

SystemBuild Block Paradigm 1-2

Chapter 2
Using BlockScript in SystemBuild

BlockScript Program Structure in a BlockScript Block 2-1
General Program Structure 2-1
Variable Name Definitions 2-2
Data Type Definitions 2-2
Update Equations 2-2
Simple Example 2-2
Using BlockScript with Simulator and AutoCode Code Phases 2-2
BlockScript Variables 2-4
Block Variable Declarations 2-5
Data Types and Dimensions 2-7
Wildcard Dimensions and Dialog Imported Information 2-9
Method for Implicit Data Typing 2-10
BlockScript Data Types and Code Generation 2-11
Environment Variables 2-11

Chapter 3
BlockScript Language

Operators and Precedence 3-1
Assignment Statements and Expressions 3-3
Arithmetic Expressions 3-3
Relational Expressions 3-3
Logical Expressions 3-4
Range Expressions 3-4
Set Expressions 3-4
Looping and Decision-Making Constructs 3-5
For Loop 3-5
While Loop 3-5
If Clause 3-5
Select Clause 3-6
Exit Statement 3-7
Iterate Statement 3-7

Functions 3-8
var.rows, var.columns, var.size 3-8
integer(a), float(a), and var.type(a) 3-8
abort(n) 3-8
abs(a) 3-8
acos(a) and asin(a) 3-8
atan(a) and atan2(y,x) 3-9
bSet(a,b), bClear(a,b), bTest(a,b) and bToggle(a,b) 3-9
bitLshift(a,b) and bitRshift(a,b) 3-9
bitNot(a), bitOr(a,b) and bitAnd(a,b) 3-9
bound(a,b,c) 3-9
exp(a) 3-9
log(a) and log10(a) 3-10
max(a,b) and min(a,b) 3-10
mod(a,b) 3-10
quad(a, w, x, y, z) 3-10
round(a), truncate(a), floor(a) and ceiling(a) 3-11
sign(a) 3-11
sin(a), cos(a), and tan(a) 3-12
sinh(a), cosh(a) and tanh(a) 3-12
sqrt(a) 3-12
swap(a,b) 3-12
trg(a, x, y, z) 3-13
uRand(s,v), nRand(s,v), ouRand(s, ouLast, timeInterval, timeConst,v) 3-14

Chapter 4
BlockScript Examples

SystemBuild Model Usage 4-1
SystemBuild Examples 4-1
Bessel Equation BlockScript Block 4-1
Discrete PID Controller BlockScript Block 4-2
Three-Cycle Delay BlockScript Block 4-5
Linear Interpolation Algorithm BlockScript Block 4-6
Hysteresis BlockScript Block 4-7
Generating a Series of Pulses 4-8
Implementing a Pulse Width, Pulse Frequency Modulator 4-11
Debugging Tip 4-17
Converting BlockScript Blocks to UCBs for Faster Simulations 4-17

Appendix A
Technical Support and Professional Services

Index

1 Introduction

BlockScript is a proprietary programming language owned by National Instruments. You can use BlockScript in SystemBuild.

BlockScript provides a generalized programming capability for:

- Defining SystemBuild BlockScript blocks

  A BlockScript program defines block inputs, outputs, and parameters, specifies their data types and dimensions, and provides the update equations that process the inputs and parameters to produce the outputs. The BlockScript block extends the concepts used in the AlgebraicExpression and LogicalExpression blocks provided by SystemBuild.

- Conditions and actions in a BetterState chart

  When you specify BlockScript for user code, BetterState can generate either C or Ada code. Thus, you can change the output language without having to change your statechart.

This document contains the following additional chapters:

- Chapter 2, *Using BlockScript in SystemBuild*, discusses the use of BlockScript in SystemBuild.
- Chapter 3, *BlockScript Language*, provides the details of the language that are independent of the application.
- Chapter 4, *BlockScript Examples*, provides a number of examples using BlockScript.

SystemBuild Block Paradigm

This section explains the SystemBuild BlockScript block paradigm, shown in Figure 1-1, and shows how the structure of the BlockScript program supports it.

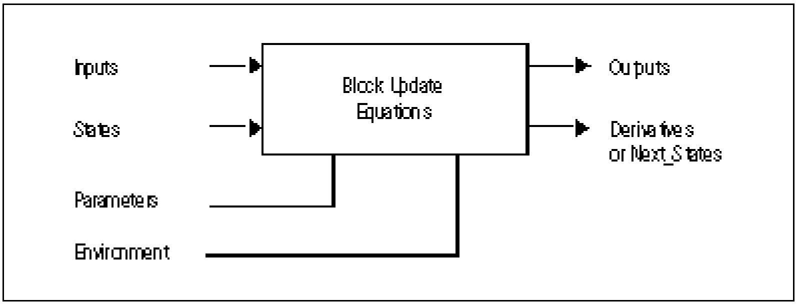

Figure 1-1. BlockScript Block Paradigm

The block update equations are programmed to accept:

- Block inputs
- States (information from the previous cycle)
- Parameters (information from the block dialog)
- Environment information, such as time and certain universal and platform-dependent constants

The block update equations produce two types of outputs:

- Block outputs
- State derivatives (continuous SuperBlock) or Next_States (discrete SuperBlock)

2

Using BlockScript in SystemBuild

You can provide user code in the BlockScript language in the BlockScript block, which is located on the User Programmed palette of the Palette Browser. This chapter focuses on the following topics:

- *BlockScript Program Structure in a BlockScript Block*
- *BlockScript Variables*

Refer to Chapter 4, *BlockScript Examples*, for the language specification.

BlockScript Program Structure in a BlockScript Block

This section presents the general structure of a BlockScript program and discusses the phases of the SystemBuild simulator and AutoCode generator with respect to how you structure your BlockScript code.

General Program Structure

The general structure of a BlockScript program in a BlockScript block is as follows:

```
# Block variable names
# Data type and dimensions definitions
# Block output update equations
```

The three sections must be presented in the order shown. The sections are defined by the formats of the statements in context, such that the first data type and dimension definition marks the end of the block variable names, and the first statement with the format of a block output update equation marks the end of the data type and dimension definitions.

Variable Name Definitions

The structure of a block variable name definition is:

```
Category: (Var1, Var2, …);
```

Categories are reserved words in BlockScript. A complete list of the supported categories is shown in the *Block Variable Declarations* section.

Data Type Definitions

BlockScript supports three data types: `Integer`, `Float`, and `Logical`. The format of a data type and dimension definition is:

```
Type Var (Dimension)
```

Block variables must receive a definition in this section. You also may assign a data type and dimension definition to any local variables.

Update Equations

The format of block output update equation statements might be simple code, or it might make use of pre-defined environment variables that define program phases. Refer to the *Using BlockScript with Simulator and AutoCode Code Phases* section.

Simple Example

In the following example, a simple addition block is programmed with two input variables, `A` and `B`, and one output variable, `C`. Notice that the order of the input list corresponds to the block input pin assignments for the block defined with this code: `A` is the first input, and `B` is the second input.

```
Inputs: (A, B);
Outputs: C;
C = A + B;
```

Using BlockScript with Simulator and AutoCode Code Phases

The SystemBuild simulator, as well as the AutoCode generator, executes the blocks in a subsystem in dataflow order for an *output* phase and then a *state* phase. The output phase creates the block and subsystem outputs. After this phase is complete, the blocks are again exercised in a state phase. In this phase, all of the dynamic blocks or blocks with states are executed. The blocks in the state phase do not have to be executed in any particular order.

This section shows you how to structure the phases of your program using the environment variables. The specific language constructs appear later in this document.

Use the environment variables `OUTPUT` and `STATE` in BlockScript to identify the code for these two phases. They are defined in BlockScript as `Logical` data type variables and are read-only user variables. For example:

```
inputs:  U;
outputs: Y;
states:  X;
derivatives: XDOT;
parameters: (A, B, C, D);
environment: (INIT, STATE, OUTPUT);

if OUTPUT then
   Y = C*X + D*U;
endif;

if STATE then
   XDOT = A*X + B*U;
endif;
```

Both the simulator and the code generator execute this code with either the `OUTPUT` variable equal to TRUE or the `STATE` variable equal to TRUE; they are never both TRUE at the same time. Therefore, you should not nest the output and state clauses as shown.

```
if STATE then             # Code that will NOT work
   if OUTPUT then
      …
   endif;
endif;
```

If you have code to be executed for both the output and state phases, place this code outside the `if` statements as shown:

```
PHI = U**2;                 # Common code segment

if OUTPUT then
   Y = C*X + D*PHI;
endif;

if STATE then
   XDOT = A*X + B*PHI
endif;
```

The statements are executed in the order that you provide in your BlockScript code. In the output phase, the assignment statement for `PHI` is executed first, followed by the assignment to `Y`. In the state phase, the assignment statement for `PHI` is executed, followed by the assignment to `XDOT`.

`INIT` is another environment variable. It is a `Logical` data type variable that is set to TRUE on the first execution of the block code and set to FALSE on all subsequent executions. It is used to initialize variables that cannot be initialized in the catalog data structures. The `INIT` variable does not represent a separate phase as do the `OUTPUT` and `STATE` variables. Instead, it is a `Logical` variable that is TRUE only on the first execution of the block; it can be nested in if statements that use the `OUTPUT` and `STATE` variables. For example:

```
parameters: (ALPHA, BETA);

if INIT then            # Common INIT code section which gets
   ALPHA = 2*U;         # exercised in both OUTPUT and STATE phases
endif;

if OUTPUT then
   if INIT then         # INIT used to modify the initial state, X,
      X = BETA*X + U;   # with parameter BETA and the initial input, U.
   endif;
   Y = C*X + D*ALPHA;
endif;

if STATE then           # INIT used in decision making logic in the
   if INIT then         # STATE phase. This X gets its value from
      XDOT = A*X;       # the 'if INIT' clause in the OUTPUT phase.
   else
      XDOT = A*X + B*ALPHA;
   endif;
endif;
```

BlockScript Variables

BlockScript programs employ two kinds of variables:

- Block variables correspond to data flow and parameters in the block dialog for the block being defined. These are the inputs and outputs of the block update equations illustrated in Figure 1-1, *BlockScript Block Paradigm*.
- Local variables take their data typing and meaning from the program context in which they are defined.

Note Local variables cannot be used to pass data between output and state program phases. You have several choices: recompute the data, use a parameter to store the data, use a state variable, or use a block output.

All the inputs and outputs in Figure 1-1, *BlockScript Block Paradigm*, are defined using lists of block variables. Updating of the outputs and derivatives is performed using the equations in the BlockScript program. The design of the BlockScript program lets you choose variable names that are descriptive in the context of the block equations.

Notice the following:

- Language operators, function names, and keywords are case insensitive.
- Variable names are case sensitive.
- Environment variables must be fully capitalized.

Block Variable Declarations

Block variables are declared with the following list construct:

`Category: (Var1, Var2, …);`

- `Category` can be one of the predefined list category names in Table 2-1.
- If there is only one variable in the list, parentheses are not required. If there are no variables in the list, then the parentheses are required but contain nothing.
- For all lists, order is significant. The first variable maps to the first input/output/state, and the last variable maps to the last element. Notice that SystemBuild redefines the number of `Inputs`, `Outputs`, and `States` based on the number of these elements that you define in the code.

You can dispense with the name list mechanism altogether if you are willing to accept the default names for each category, as listed in Table 2-1.

Table 2-1. Default Variable Names

List Category Name	Default Variable Name	Definition
`Inputs`	`u`	A list of input variable names.
`Outputs`	`y`	A list of output variable names.

Table 2-1. Default Variable Names (Continued)

List Category Name	Default Variable Name	Definition
`States`	(none)	A list of state names.
`Derivatives`	(none)	A list of state derivatives. This declaration is only valid for continuous dynamic blocks. The dimension of this list must agree with the dimension of the `States` list.
`Next_States`	(none)	A list of next-state variable names. This declaration is only valid for discrete dynamic blocks. The dimension of this list must agree with the dimension of the `States` list.
`Parameters`	(none)	A list of parameter names. This list implies order. If AutoCode maps the variables into `Rpar` and `Ipar` vectors, mapping duplicates the order in the `Parameters` list. If a list of parameters is supplied, additional fields are added to the block dialog in the order specified in the `Parameters` list.
`Environment`	Refer to the *Environment Variables* section.	The SystemBuild and AutoCode environment provides predefined variables that can be imported into the BlockScript code through this `Environment` list. The variables in the *Environment Variables* section are available regardless of the environment (simulation or generated code).

For example, a simple signal generator might be coded as follows:

```
Inputs: ();
Outputs: y;

Parameters (Phi, Theta, Psi);
Environment: TIME;

y= Sin(TIME);
```

Note `TIME` is an environment variable defined in the *Environment Variables* section.

You can enter hard default values for these parameters in the dialog. Also, you can provide a **%variable** name for each parameter. The maximum number of parameterized variables for a BlockScript block is 10.

Data Types and Dimensions

BlockScript supports three data types: `Float`, `Integer`, and `Logical`. Data typing is performed according to the rules in Table 2-2.

Table 2-2. Data Typing Rules

Category	Default Type	OK to assign data type to variable?
`Inputs`	`Float`	Yes
`Outputs`	`Float`	Yes
`States`	`Float`	Yes
`Derivatives`	`Float`	Yes; type must agree with `States`
`Next_States`	`Float`	Yes; type must agree with `States`
`Parameters`	user-defined	Yes, required
`Environment`	predefined	No

If you do not explicitly assign a data type to a local variable, then it is defined as a scalar variable whose data type agrees in context with the first statement that defines it in the BlockScript code.

Parameters and local variables can be scalars, vectors, or matrices. `Inputs`, `Outputs`, and `States` can use names that are scalars or vectors. For vector variables, you can use *`var`*`.size` to obtain its current dimension. For matrix variables, you can use the variables *`var`*`.rows` and *`var`*`.columns` to obtain a dimension.

`var``.size` has special meanings for different variable shapes. For scalars, it is 1; for vectors, it is the dimension specified (wildcarded or not); and for matrices, it is the product of the two specified dimensions.

Consider Example 2-1, which defines a nonlinear Breakpoints block.

Example 2-1 BlockScript for Nonlinear Breakpoints Block

```
Inputs: U;
Outputs: Y;
Parameters: (UBrk, YBrk);
Float U, Y(:);
Float UBrk(:), YBrk(Y.size,:);
J = 1;
K = UBrk.size;
Uval = U;

While J < K-1 Do
   M = (J + K)/2;
   If Uval < UBrk(M) Then
      K = M;
   Else
      J = M;
   EndIf;
EndWhile;

Alpha = (Uval - UBrk(J)) / (UBrk(K)- UBrk(J));
For I = 1:Y.size Do
   Y(I) = (1.0-Alpha)*YBrk(I,J) + Alpha*YBrk(I,K);
EndFor;
```

- The input, `U`, is a scalar.
- The output, `Y`, is a vector that has a wildcard dimension (the colon operator); refer to the *Wildcard Dimensions and Dialog Imported Information* section. Its dimension is imported from the **Outputs** field in the block dialog. The breakpoints are specified as parameters with two variables, `UBrk` and `YBrk`. Parameters also can be given wildcard characters for their dimensions, so that they can be determined from user inputs in the block dialog.
- Variable size can be used as a dimension in any other variable dimension except `Environment`, because environment variables have predefined sizes. Notice the use of the compile time variable `Y.size` to obtain the current dimension of a vector variable in Example 2-1.

Wildcard Dimensions and Dialog Imported Information

The colon (`:`) wildcard character can be used for any parameter dimension. This is possible because the dimensions are not constrained in the block dialog. For example:

```
Parameters: (F,G,H);
Float F(:), G(:,:), H(:,:);
```

You also can use the colon wildcard character for dimensioning `Inputs`, `Outputs`, `States`, `Derivatives`, and `Next_States`.

You can use a colon wildcard with a signal name only if there is just one name in the list because the block dialog provides only the total number of signal values and does not accommodate a list of names.

```
Inputs: U;
Float U(:);
```

Any variable's total size, (*var*`.size`), number of rows, (*var*`.rows`), or number of columns, (*var*`.columns`), can be used as a dimension for any other variable, excluding itself. The size of a variable can be used before the size and data type are defined. For example,

```
Inputs: U;
Float WorkSpace(U.size, Pivot.size);
Float U(:), Pivot(5);
```

This constrains the specified dimension to follow the dimension of *var*. Any constrained dimension, either hardcoded or described by *var*`.size`, is not free—that is, it cannot be changed in the block dialog.

You can change dimensions that are specified with the wildcard character later from the BlockScript block dialog. If you decrease the dimension, information referenced outside that dimension is discarded. If you increase the dimension, the last value of the vector is repeated to extend the vector. If you extend a matrix, the extended area is filled with zeros.

You can use the variables, *var*`.size`, *var*`.rows`, *var*`.columns`, as well as a generic casting function, *var*`.type( )`, in the code.

```
For I = 1:A.rows Do
   For J = 1:A.columns Do
      Y(I) = Y.type(A(I,J)*U(J));
   EndFor;
EndFor;
```

Method for Implicit Data Typing

Not all data types must be explicitly specified. In Example 2-1, the variable `J` is an `Integer` because it is assigned the integer literal `1`. The decimal point and/or `E` in scientific notation are used to specify float literals. `K` is also an `Integer` because `UBrk.size` returns an integer value. `M` is an `Integer` because $(J + K)/2$ evaluates to an integer. `Uval` and `Alpha` are float variables because they are evaluated with float expressions. `I` is an `Integer` because `1:Y.size` is an integer range expression. It is possible to code floating point For loop ranges such as
`For Angle = –Pi : Pi/10.0 : Pi Do`.

Mixing data types within an expression results in a promotion of the intermediate computation. The promotion consists of converting the integer computation to a float computation, which results in a `Float` data type. The `Float` data type is then propagated through the outer expressions. Consider the following equation:

```
Integer I, J, K;
I = (J + K)*3.14 + 255 / (L + M);
```

Both $(J + K)$ and $255/(L + M)$ are evaluated as integer expressions. Furthermore since integer division causes truncation towards zero, $255/(L + M)$ contains that truncated value. Next, multiplication by 3.14 makes (J + K) * 3.14 a float. When added to the integer expression $255/(L + M)$, the resulting right-hand side (RHS) of the equation becomes a float expression. Since `I` is an integer, the RHS float expression is again truncated towards zero before storing the result in variable `I`. The only difference between integer and float expressions is the implied truncation towards zero when dividing two integers or when assigning a float expression to an `Integer` variable. Mixing arithmetic with the explicit casting functions `Integer( )` and `Float( )` is preferred.

Although `Logical` is a special form of the `Integer` data type, and the C language treats them the same in its syntax, other languages, such as Ada and Java, do not. In this regard, BlockScript was designed to deal with Logical variables the way Boolean variables are treated in Ada and Java. Therefore, you must declare and use Logical variables when they are intended to hold logical results. Refer to Example 2-2.

Example 2-2 Declaring Logical Variables

```
Logical Negative, InRange, OK;
Negative = A < 0.0;
InRange= A > B & A < C;
OK = InRange & ! Negative;
If OK Then
…
EndIf;
```

BlockScript Data Types and Code Generation

In most situations, if the **Typecheck** checkbox (SystemBuild Simulation Parameters dialog) is disabled, all signals are forced to be `Float`. The BlockScript block is an exception. The **Typecheck** option does not affect how the BlockScript block is simulated or how code is generated for it. Therefore, if you plan to use a BlockScript block in a model in which you are not enabling the **Typecheck** checkbox, make your inputs and outputs `Float` to be compatible with other signals in your model.

Environment Variables

Environment variable names must be all upper case. They are read-only values. In particular, two of these values—`OUTPUT` and `STATE`—are controlled by SystemBuild to identify program phases. Refer to the *Using BlockScript with Simulator and AutoCode Code Phases* section. `INIT` is set to TRUE by SystemBuild the first time BlockScript is called during simulation or code generation.

ABSTOL

`ABSTOL` is the absolute tolerance specified in the `sim({abstol=value})` function call. It is a floating point value.

EPSILON

`EPSILON` is the smallest floating point value that can be added to unity and change its value. This value is machine dependent.

INIT

`INIT`, a `Logical` variable, is set to TRUE the first time the BlockScript program is called during simulation or code generation. It is set to FALSE at all other times. Refer to the *Using BlockScript with Simulator and AutoCode Code Phases* section.

OUTPUT

`OUTPUT`, a `Logical` variable, is set to TRUE to request the BlockScript program to perform output update computations.

PI

`PI` (3.14159…) is the circumference of a circle divided by its diameter.

RELTOL

`RELTOL`, a `Float` variable, is the relative tolerance specified in the `sim(…,{reltol=value})` function call.

STATE

`STATE`, a `Logical` variable, is set to TRUE to request the BlockScript program to perform state update computations.

TIME

`TIME` is the current value for time. It is a floating point scalar.

TSAMP

`TSAMP` is a floating point value that is the sample period of the parent discrete SuperBlock. If the parent is a triggered SuperBlock, `TSAMP` is defined to be 1.0.

TSTART

`TSTART` is set to zero for the initial `sim( )` call. It is set to the final time from the previous `sim( )` call when you resume a simulation.

3

BlockScript Language

This chapter describes the BlockScript language. You can use the information found in this chapter for a SystemBuild BlockScript block.

The major topics in the chapter are as follows:

- *Operators and Precedence*
- *Assignment Statements and Expressions*
- *Looping and Decision-Making Constructs*
- *Functions*

Operators and Precedence

BlockScript's precedence of operators is similar to those in the C language with the following differences:

- In C, the `Logical` data type is an integer, and therefore logical operators combine integer values. In BlockScript, logical and integer data are different.
- BlockScript makes a distinction between numeric equivalence, `==`, and logical equivalence, `~`, but places them next to each other in the table to provide the same precedence as in C.
- C puts the precedence for bitwise XOR, `^`, between `AND` and `OR`. `XOR` also is NEQV, `!~`; BlockScript places it with EQV, `~`.

Table 3-1 illustrates the BlockScript operators and precedence.

Table 3-1. Operator Precedence

<table>
<tr><th>Operator Type</th><th>Operators</th><th>Operator Names and Meanings</th><th>Alias Operator</th><th>Associativity</th><th>Precedence</th></tr>
<tr><td>Primary</td><td><code>( ) ,</code></td><td>Subexpressions, functions, arrays</td><td>—</td><td>Left to right</td><td>Highest</td></tr>
<tr><td>Power</td><td><code>^</code> or <code>**</code></td><td>Power</td><td>—</td><td>Right to left</td><td></td></tr>
<tr><td>Multiplicative</td><td><code>*</code>
<code>/</code></td><td>Multiply
Divide</td><td>—</td><td>Left to right</td><td></td></tr>
<tr><td>Unary</td><td><code>+</code>
<code>–</code>
<code>!</code></td><td>Unary plus
Unary minus
Not, Complement</td><td>—</td><td>Right to left</td><td></td></tr>
<tr><td>Additive</td><td><code>+</code>
<code>–</code></td><td>Plus
Minus</td><td>—</td><td>Left to right</td><td></td></tr>
<tr><td>Shift</td><td><code><<</code>
<code>>></code></td><td>Shift left
Shift right</td><td><code>LSHIFT</code>
<code>RSHIFT</code></td><td>Left to right</td><td></td></tr>
<tr><td>Range</td><td><code>:</code></td><td>Define range</td><td>—</td><td>Right to left</td><td></td></tr>
<tr><td>Relational</td><td><code><</code>
<code><=</code>
<code>></code>
<code>>=</code>
<code><></code>
<code>==</code></td><td>Less than
Less than or equal
Greater than
Greater than or equal
Not equal
Equal</td><td><code>LT</code>
<code>LE</code>
<code>GT</code>
<code>GE</code>
<code>NE</code>
<code>EQ</code></td><td>Left to right</td><td></td></tr>
<tr><td>Logical equivalence</td><td><code>~</code>
<code>!~</code></td><td>Equivalence, Eqv, nXOR
Not Equiv, Neqv, XOR</td><td><code>EQV,</code>
<code>NXOR</code>
<code>NEQV,</code>
<code>XOR</code></td><td>Left to right</td><td></td></tr>
<tr><td>Logical AND</td><td><code>&</code>
<code>!&</code></td><td>AND, Intersection
NAND</td><td><code>AND</code>
<code>NAND</code></td><td>Left to right</td><td></td></tr>
<tr><td>Logical OR</td><td><code>|</code>
<code>!|</code></td><td>OR, Union
NOR</td><td><code>OR</code>
<code>NOR</code></td><td>Left to right</td><td></td></tr>
<tr><td>Assignment</td><td><code>=</code></td><td>Variable assignment</td><td>—</td><td>Right to left</td><td>Lowest</td></tr>
</table>

Note BlockScript has a set of standard operators, such as `+`, `*`, `<`, and also a set of alias operator names that you can use if you prefer or if your keyboard does not contain all the standard symbols.

Assignment Statements and Expressions

You can assign a value to a variable using an expression to the right of the assignment operator. By default, all block variables are floating point scalar data. If you do not explicitly assign a data type to a local variable, then the local variable is automatically assigned the data type of the right-side expression that first defines it within its function body. After the data type is assigned, you can assign integer variables to floating point expressions and vice versa. You can only assign relational or logical expressions to logical variables. There are five kinds of expressions: arithmetic, relational, logical, range, and set.

Arithmetic Expressions

Arithmetic expressions typically use only arithmetic operators (`**`, `*`, `/`, `+`, `–`). The bitwise operators, which only take integer expressions for their operands, use the same symbols and their precedence as the logical operators. The results are arithmetic.

For example,

```
a = 5;
b = 6;
x = a & b;
y = a + b;
```

The value of `x` is 4. The value of `y` is 11.

Relational Expressions

Relational expressions compare two arithmetic expressions to form a logical result. Relational expressions use the following operators:

```
<     <=     >     >=     <>     ==
```

Logical Expressions

Logical expressions combine logical expressions and/or relational expressions with logical operators to produce logical results. Logical operators are as follows:

primary:	`( )`	
unary:	`!`	
logical EQV:	`~`	`!~`
logical AND:	`&`	`!&`
logical OR:	`|`	`!|`

Range Expressions

Range expressions combine arithmetic values or expressions with the define range operator (`:`) to specify a set of values. A range expression is defined as follows:

```
Range := Start : Increment : End
```

If the increment is omitted, then its value is 1. Ranges may be either integer or floating point.

Set Expressions

Set expressions combine range expressions with the union operator (|) to define sets of values. If ranges are used with the `Float` data type, sets are composed with a discrete number of continuous regions of values.

The syntax for a Set expression is shown below:

```
Set := Region | Region | Region | …
Region := { Range | Value }
```

The vertical bar enclosed in braces in the syntax represents a choice between the enclosed identifiers. The vertical bar used in the set expression is the union operator (`|`) and is required in the syntax. The intersection operator (`&`) and parentheses `( )` are not used in set expressions. All identifiers in a set expression must be the same data type. The set expressions are used in the `Select` clause.

Looping and Decision-Making Constructs

BlockScript provides four constructs for looping and decision making: `For`, `While`, `If`, and `Select`. The `Iterate` and `Exit` statements, which are described at the end of this section, support these constructs.

For Loop

You can use the `For` loop when the body of the loop should be executed a known number of times. The loop counter is a range expression. Its values can be either integer or floating point but should be consistent. The following is the syntax of the `For` loop.

```
For LoopVar = LoopRange Do
   LoopBody;
EndFor;
```

The *LoopBody* is any number of BlockScript statements. The *LoopRange* is in either of the following two formats:

```
Start : End
Start : Increment : End
```

The default *Increment* is 1.

While Loop

The `While` loop is used when the loop body should be executed until some condition is met. The following is the syntax of the `While` loop.

```
While LogicCondition Do
   LoopBody;
EndWhile;
```

LoopBody is any number of BlockScript statements. The *LogicCondition* is any valid scalar logical expression.

Note The *LogicCondition* may use any number of previously defined variables. However, in BlockScript, all input variables used in the loop body must be scalars or subscripted with literals.

If Clause

The `If` clause is used to conditionally execute one of several bodies of code depending on a TRUE evaluation of its condition. The following is the syntax of the `If` clause.

```
If LogicCondition Then
   ConditionBody;
ElseIf LogicCondition Then
   ConditionBody;
Else
   ConditionBody;
EndIf;
```

There may be any number of `ElseIf` clauses. *`ConditionBody`* can consist of any number of BlockScript statements. *`LogicCondition`* is any valid scalar logical expression. It may use any number of previously defined variables. You can omit both the `ElseIf` and `Else` clauses.

Select Clause

The `Select` clause is used to conditionally execute one or more bodies of code depending on a variable whose value matches the values in the corresponding sets specified with `Case` statements.

Note The `Select` clause must contain at least one `Case` statement. The optional `Otherwise` case is executed only if no cases match. The `Otherwise` case can be omitted.

Following is the syntax for the `Select` clause.

```
Select ChoiceVar ClauseForm
Case ConstSet
   CaseBody;
Case ConstSet
   CaseBody;
Otherwise
   CaseBody;
EndSelect;
```

where:

`ChoiceVar` is any integer or floating point variable previously defined.

`ClauseForm` is either `OneOf` or `AllOf`.

- The `OneOf` keyword instructs BlockScript to execute only the first `Case` that matches.
- The `AllOf` keyword allows BlockScript to execute all cases that match.

`CaseBody` is any number of BlockScript statements to be executed for this case.

Note At the end of each *CaseBody*, there is an automatic break to the next `Case` that matches (if `ClauseForm` is `AllOf`) or `EndSelect` (if `ClauseForm` is `OneOf` or `Otherwise` is being executed).

ConstSet is a set of values specified by scalar constant values and constant ranges.

A vertical bar (|) represents a choice between one or more identifiers. In the case of *ConstSet*, it functions as the union operator.

- The data types for all *ConstSets* must agree within the set and must be the same type as *ChoiceVar*.
- If the *ChoiceVar* is type float, you cannot use a range. For example 1.0:3.0 is not accepted. To achieve the same thing, specify 1.0|2.0|3.0.

The *ConstSet* syntax that follows is recursive such that subsets within *ConstSet* can be ranges or values specified as floating points or integers, if appropriate.

ConstSet	*Subset* | *Subset* | *Subset* …
Subset	*Range* | *Value*
Range	*StartValue* : *EndValue*
Value	*IntegerValue* | *FloatValue*

Exit Statement

The `Exit` statement is used to break out of loops. Execution resumes just after the matching `End` keyword.

Note Unlike the C language's `break` statement, `Exit` cannot be used to break out of `Case` statements.

Iterate Statement

Use the `Iterate` statement to invoke the next iteration of the corresponding current `For` or `While` loop. Execution resumes where the loop variable is incremented in `For` loops or where the logical condition is tested in `While` loops.

Functions

This section describes all intrinsic BlockScript functions.

var.rows, var.columns, var.size

These functions return the size of a variable. Use *var*`.rows` and *var*`.columns` for matrix variables and *var*`.size( )` for vectors. In the matrix case, *var*`.size` returns the product of row and column size. These functions return integer values.

integer(a), float(a), and var.type(a)

These functions provide explicit casting operations for converting `Float` to `Integer` and vice versa. The `integer( )` casting function truncates the value towards zero as is the case for Fortran, C, and Ada. *var*`.type( )` is a general casting function that produces a resulting data type that agrees with *var*. If the *var* data type is `Integer`, then integer truncation occurs.

abort(n)

`abort( )` is a `void` function. Its output cannot be assigned to a variable, but it can be used as a procedure call. It must be passed an integer literal value that encodes a severity level and a message index. Its purpose is to stop or raise an exception during the simulation or running of generated code. The values for the integer are the same error message variables as those defined for UserCode blocks. These are negative values. Refer to the *SystemBuild User Guide* for details.

abs(a)

This function takes the absolute value of its argument. The resultant data type is the same as that of the argument.

acos(a) and asin(a)

`acos( )` and `asin( )` return the arc cosine and arc sine, respectively, of the argument. The argument must be floating point. If the argument is larger than 1 or less than –1, a run-time error occurs. `acos( )` returns a floating point value in the range 0 to π. `asin( )` returns a floating point value in the range $-\pi/2$ to $\pi/2$.

atan(a) and atan2(y,x)

Both of these functions return the arc tangent of their input argument(s). `atan( )` returns a floating point value in the range $-\pi/2$ to $\pi/2$. `atan2(y,x)` returns the arc tangent of (y/x), which is a floating point value in the range of $-\pi$ to π depending upon which quadrant (x,y) maps in the Cartesian coordinate frame. If `atan2( )` is passed two zero values, a run-time error occurs.

bSet(a,b), bClear(a,b), bTest(a,b) and bToggle(a,b)

These functions set, clear, test, or toggle bit `b` in integer word `a`. The bit position, `b`, is 0 for the low-order bit. `bTest( )` returns a `Logical` result.

bitLshift(a,b) and bitRshift(a,b)

`bitLshift(a,b)` shifts integer word `a` left `b` bits whereas `bitRshift(a,b)` shifts integer word `a` right `b` bits. The output type is `Integer`.

bitNot(a), bitOr(a,b) and bitAnd(a,b)

`bitNot( )` performs a bitwise complement of integer word `a`. `bitOr( )` and `bitAnd( )` perform bitwise AND and OR operations, respectively, for their input arguments. The output type is `Integer`.

bound(a,b,c)

`bound( )` returns:

`b`, if `a` is less than `b`
`c`, if `a` is greater than `c`
`a`, otherwise

The arguments must be all floating point or all integer. The returned value is the same data type as the arguments to `bound( )`.

exp(a)

`exp( )` returns the value `e` raised to the power `a`, where `e` is the natural number (2.7183…). `a` must be floating point, and the returned value is floating point.

log(a) and log10(a)

`log( )` returns the base `e` logarithm of its input argument whereas `log10( )` returns the base 10 logarithm. Both functions require a floating point input argument and produce a floating point result. If the input is negative, a run-time error occurs.

max(a,b) and min(a,b)

`max( )` returns the larger of the two arguments whereas `min( )` returns the smaller of the two. Both arguments must agree in data type. The returned value has the same data type.

mod(a,b)

This function takes two arguments. It performs the operation:

`a - b*integer(a/b)`

Both `a` and `b` must be the same data type. The resultant data type is the same as that of its arguments.

quad(a, w, x, y, z)

`quad( )` accepts floating point arguments and returns a floating point result. The function is evaluated as follows.

If *a* is	Then the function evaluates to
In the interval [*x*,*y*]	1.0
Less than or equal to *w* OR greater than or equal to *z*	0.0
In the interval (*w*,*x*)	An interpolated value between 0.0 and 1.0
In the interval (*y*,*z*)	An interpolated value between 1.0 and 0.0

The values *w*, *x*, *y*, and *z* must be increasing. Notice that *w* may be equal to *x* and/or *y* may be equal to *z*. Figure 3-1 shows these results graphically.

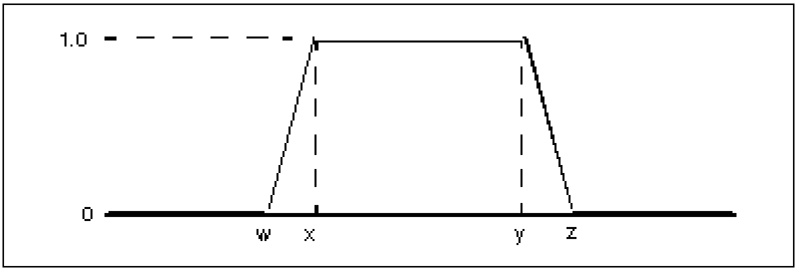

Figure 3-1. Graphical Evaluation of quad( )

round(a), truncate(a), floor(a) and ceiling(a)

These functions accept a floating point input and produce a floating point output whose value is equal to an integer:

`round( )`	Nearest integer
`truncate( )`	Nearest integer in a direction towards zero
`floor( )`	Nearest integer whose value is less than or equal to *a*
`ceiling( )`	Nearest integer whose value is greater than or equal to *a*

sign(a)

`sign( )` computes the signum function of its input. It is defined as follows and shown graphically in Figure 3-2.

If *a* is	Then the function evaluates to
< 0	–1
== 0	0
> 0	+1

The resulting data type is the same as that of its argument.

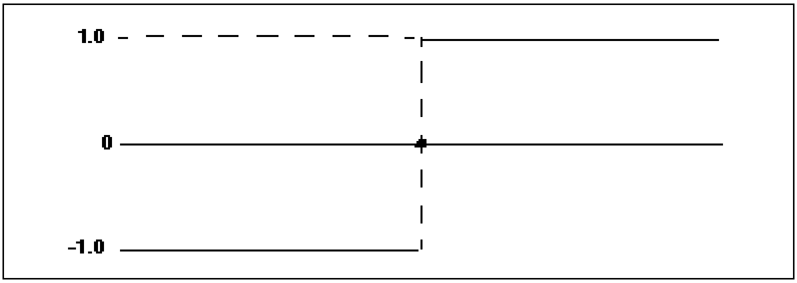

Figure 3-2. Graphical Evaluation of sign( )

sin(a), cos(a), and tan(a)

`sin( )` computes the sine of its input whereas `cos( )` computes the cosine of its input. Both functions require a floating point input and return a floating point result in the range of –1 to 1. `tan(a)` computes the tangent of *a*. If *a* is a multiple of π, `tan( )` overflows; SystemBuild does not trap IEEE floating point NaN (not a number) or Inf (infinity). The output of `tan( )` is floating point.

sinh(a), cosh(a) and tanh(a)

These functions compute the respective hyperbolic functions. `sinh( )` returns a floating point value. `cosh( )` returns a value that is greater than or equal to unity. `tanh( )` returns a value greater than –1 and less than +1.

sqrt(a)

This function returns the square root of its input argument. A run-time error occurs if the input argument is negative. Both the argument and the returned value are floating point.

swap(a,b)

This function swaps the values referenced by *a* and *b*. *a* and *b* must be simple variable name references and can both be either floating point or integer.

trg(a, x, y, z)

`trg` accepts floating point arguments and returns a floating point result. The function is evaluated as follows:

If *a* is	Then the function evaluates to
== *y*	1.0
Less than or equal to *x* OR greater than or equal to *z*	0.0
In the interval (*x*,*y*)	An interpolated value between 0.0 and 1.0
In the interval (*y*,*z*)	An interpolated value between 1.0 and 0.0

The values *x*, *y*, and *z*, must be increasing. Notice that *x* may be equal to *y* and/or *y* may be equal to *z*. Figure 3-3 shows a graphical representation.

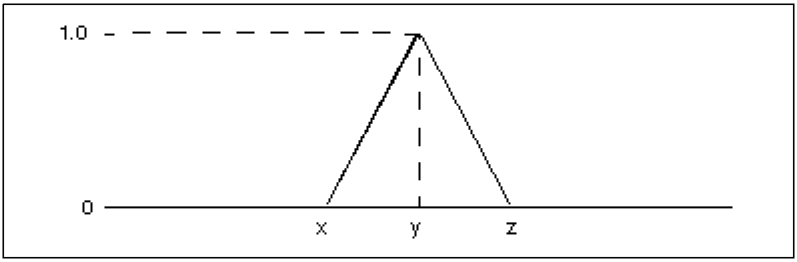

Figure 3-3. Graphical Evaluation of trg( )

uRand(s,v), nRand(s,v), ouRand(s, ouLast, timeInterval, timeConst,v)

These functions generate random numbers. The first argument, *s*, is an integer seed. The seed must be declared as a parameter (not a literal), so it can be changed by the function.

- `uRand( )` is a uniform random number generator that returns a floating point value in the range of 0.0 to 1.0 in the *v* argument.
- `nRand( )` is a normal random number generator that returns a floating point value in the *v* argument. *v* is a Gaussian value that has zero mean and unit variance.
- `ouRand( )` implements the Ornstein-Uhlenbeck process for generating band-limited white noise. It is correlated with past history given the floating point values *ouLast*, *timeInterval*, and *timeConst*. The *timeInterval* should be the delta time between the current and previous function call. *ouLast* is the last value returned from the previous function call. The random value is returned in the *v* argument.

4 BlockScript Examples

This chapter includes examples using BlockScript in both SystemBuild and BetterState. The *SystemBuild Model Usage* section provides models for SystemBuild alone. The *Generating a Series of Pulses* section provides a model that includes BlockScript usage in both SystemBuild and BetterState.

SystemBuild Model Usage

This section provides examples and usability tips. The major topics are:

- *SystemBuild Examples*
- *Debugging Tip*
- *Converting BlockScript Blocks to UCBs for Faster Simulations*

SystemBuild Examples

The following sections contain examples that demonstrate BlockScript capabilities. The first two examples show how an equation can be expressed as BlockScript and included in a model. The remaining examples are scripts that demonstrate BlockScript solutions for a variety of problems.

Bessel Equation BlockScript Block

This example uses a BlockScript block to model and solve a nonlinear differential equation, also known as a Bessel equation of order zero:

$$y'' + \frac{1}{u} y' + y = 0$$

To use the equation in BlockScript, it must be transformed to state-space representation:

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -x_1 - \frac{1}{u}x_2 \\ y &= x_1 \end{aligned} \tag{4-1}$$

To simulate and plot the Bessel equation BlockScript block example, complete the following steps:

1. Load the catalog file from the Xmath command area:

```
load file="$SYSBLD/examples/blockscript_example/blkscript_ex1.cat"
```

2. From the Catalog Browser, double-click **BlockScript_Example1**.

   The BlockScript block **Bessel_eq_BScript** displays the script used to implement the Bessel equation.

3. In Xmath, enter the time and input vectors:

   ```
   t = [0:.1:10]'; u = ones(t);
   ```

4. Open the **Bessel_eq_BScript** BlockScript Block dialog box. Click the **Code** tab, and examine the source code for the block. Notice the parameters, `x1_init` and `x2_init`.

5. Click the **Parameters** tab, and scroll through the parameters. Notice the parameters, `x1_init` and `x2_init`, on this tab. Also notice the **%variable** names assigned to them.

   You can assign different values to these parameters using the **%variable** names in Xmath.

6. From the Xmath command area, simulate the model and plot the results:

```
[ ,y] = sim( "BlockScript_Example1", t, u, {vars} );
plot( t, y, {title = "Solution of Bessel eq. order = 0",
   x_lab = "time [s]"} )?
```

Discrete PID Controller BlockScript Block

This example illustrates the BlockScript implementation of a discrete PID controller. This controller is available as a standard block; however, it also can be modeled successfully using the BlockScript block. In some instances, you may want a PID controller with dynamically scheduled gains that can be adjusted during actual simulation. The BlockScript implementation is a good solution in this case.

To keep the example simple, you do not modify the gains; however, the BlockScript implementation of the PID controller presented is ready to be used with dynamically adjusted gains.

This example uses the following equations for the proportional, integral, and derivative components. The equations are represented in the z domain:

$y_p = K_p \; u$ proportional component

$y_i = \left(\frac{K_i T_s}{z-1}\right) u$ integral component (Forward Euler Integrator)

$y_d = \left(\frac{K_d (z-1)}{T_s z}\right) u$ derivative component

The output is:

$$y = y_p + y_i + y_d$$

The state-space representation of the above dynamic system is:

$x_1[k+1] = x_1[k] + T_s u$ the integral state

$x_2[k+1] = u[k]$ the derivative state

$$y[k] = K_p u[k] + K_i x_1[k] + \frac{K_d (u[k] - x_2[k])}{T_s}$$

where:

T_S is the sample period for the discrete PID controller

K_p is the proportional component gain

K_i is the integral component gain

K_d is the derivative component gain

To simulate and plot the discrete PID controller BlockScript block example, complete the following steps:

1. Load the Catalog file from the Xmath command area:

```
load file = "$SYSBLD/examples/blockscript_example/blkscript_ex1.cat"
```

2. From the Catalog Browser, double-click **BlockScript_Example2**.

   The BlockScript block PID_Ctrl_BScript contains the script for the PID controller.

3. In Xmath, enter the time and input vectors:

```
t = [0:.001:.04]'; u = ones(t);
```

4. In the diagram, notice that the outputs of the PID_params AlgebraicExpression block are the inputs for the PID_Ctrl_BScript block. On the **Parameters** tab of the PID_params AlgebraicExpression block, notice that these parameters have the **%variable** name `pid_gains`.

   You can input the values that are in the dialog box through Xmath by typing the following statements:

```
kp   = 2;       #-- Proportional component gain
ki   = 100;     #-- Integral component gain
kd   = 0.002;   #-- Derivative component gain
pid_gains = [kp, ki, kd];
```

   You can change these values the same way.

5. Open the PID_Ctrl_BScript BlockScript block. On the **Parameters** tab, notice the **%variables** by which you can enter initial values.

   You can input the values that in the dialog box through Xmath by typing the following statements:

```
ts    = 0.001  #-- Sample period for the discrete PID controller[s]
x0_1  = 0;     #-- Initial value for integral state #1
x0_2  = 0;      #-- Initial value for derivative state #2
```

   You can change these values the same way.

6. From the Xmath command area, simulate the model and plot the results:

```
[, y] = sim( "BlockScript_Example2", t, u, {vars} );
plot( t, y, {marker, x_lab = "time [s]",
title = "Cl. Loop step resp. (PID controller -> Motor)" } )?
```

Three-Cycle Delay BlockScript Block

Example 4-1 implements a three-cycle delay block. The standard delay block implementation in SystemBuild uses states and next states/derivatives. Although the SystemBuild implementation is a complete solution, it may be expensive for simple needs. Here you use a BlockScript block to develop a custom algorithm that is highly efficient.

Example 4-1 Three-Cycle Delay

```
inputs: u;
outputs: y;
parameters: (DelayBuffer, Index);

float   u(y.size), y(:);
y.type  DelayBuffer(y.size, 3);
integer Index;

for i = 1:y.size do
   y(i) = DelayBuffer(i, Index);
   DelayBuffer(i, Index) = u(i);
endfor;

Index = 1 + Mod(Index, 3);
```

The parameter `DelayBuffer` is used for holding the input value and is copied into the output variable when it is appropriate to do so. This buffer is two-dimensional with the number of rows equal to the number of outputs and number of columns equal to the number of delay stages, three in this example. Actual delay is accomplished by treating this buffer as a circular buffer and moving the read/write index in a circular fashion.

The parameter `Index` is used to record the circular indexing details. This example also illustrates the use of parameters for remembering values from one cycle to another. Using states for such a simple application would be a burden because states are double-buffered.

`DelayBuffer` is initialized to an initial value specified on the BlockScript block parameters tab. Similarly, the parameter `Index` also can be initialized to an appropriate value.

Linear Interpolation Algorithm BlockScript Block

Example 4-2 implements a simple linear interpolation algorithm. In the SystemBuild implementation of linear interpolation, the interpolation tables are parameters to the block. Circumstances can represent a need to interpolate among input values—that is, the interpolation tables can be dynamic. This can be efficiently implemented in BlockScript using the input variable to represent both the actual input and the interpolation table.

Example 4-2 Interpolating Among Input Values

```
inputs: u;
outputs: y;
parameters: (Gain);
float u(:), ulocal(u.size-1), y;
integer index, length;
float slope;

for i = 1:u.size-1 do
   ulocal(i) = u(1+i);
endfor;
length = (u.size - 1) / 2;

found = false;
index = 0;
while (!found) do
  if (u(1) < ulocal(index+1)) then
    found = true;
  else
    index = index + 1;
  endif;

  if (index == length) then
    found = true;
  endif;
endwhile;

if (index == 0) then
  yout = ulocal(length+1);
elseif (index == length) then
  yout = ulocal(length*2);
else
  slope = (ulocal(index+length+1) -
ulocal(index+length)) /
          (ulocal(index+1) - ulocal(index));
```

```
  y = ulocal(index+length) + slope * (u(1) -
ulocal(index));
endif;
```

The first occurrence of the input vector u represents the actual input, whereas the remaining occurrences represent the interpolation input and output tables. The input and output tables are the same size. The input variable u is copied into a local variable ulocal because only local variables can be indexed with a while loop. Based on these tables, slope is calculated; slope is used along with the actual input value to determine the output value.

Hysteresis BlockScript Block

Consider the following BlockScript script for the Hysteresis (Backlash) block for continuous SuperBlocks in SystemBuild. To write this file to your current working directory, enter the following in the Xmath command area:

```
copyfile "$SYSBLD/examples/blockscript_example/hysteresis.txt"
```

Notice that all vector sizes are inherited from the **Outputs** field in the dialog box. This means that dimension changes in the **Inputs** and **States** fields are ignored. Likewise, the sizes for the parameters are fixed to match the Outputs dimension. In the script, omega is the cutoff frequency from the block dialog box. Notice that this program uses estate and halfw, two local variables that are defined when they are first used.

Example 4-3 Hysteresis Script

```
inputs: u;
outputs: y;
states: x;
Derivatives: xdot;
parameters: (omega, width, slope);

float y(:), u(y.size), x(y.size),
xdot(y.size);

float omega(y.size),
width(y.size), slope(y.size);

for i = 1:y.size do
   y(i)=slope(i)*x(i);
   halfw=width(i)/2.0;
   estate = u(i)-x(i);
   if estate>halfw then
      xdot(i) = omega(i)*(estate
         - halfw);
```

```
      elseif estate < -halfw then
         xdot(i) = omega(i)*(estate
            + halfw);

      else
         xdot(i) = 0.0;
      endif;
   endfor;
```

Generating a Series of Pulses

`series_of_pulses` is an example that uses a State Transition Diagram that interfaces with a SystemBuild block diagram. The block diagram has two BlockScript blocks whereas the statechart uses BlockScript for the conditions and actions.

You can find the actual model in the SystemBuild `examples` directory. To load and run the model, type the following command in the Xmath command area:

```
exec file = "$SYSBLD/examples/BlockScriptPulses/series_of_pulses.ms"
```

The purpose of the model is to output a series of pulses when a single start command is specified. The frequency of the pulses is 200 Hz with a 50% duty cycle. To obtain this frequency, you define a discrete periodic SuperBlock with a sample period of 0.0025 seconds, the amount of time that you want the pulses to be high in value. The **Series of Pulses** SuperBlock is shown in Figure 4-1.

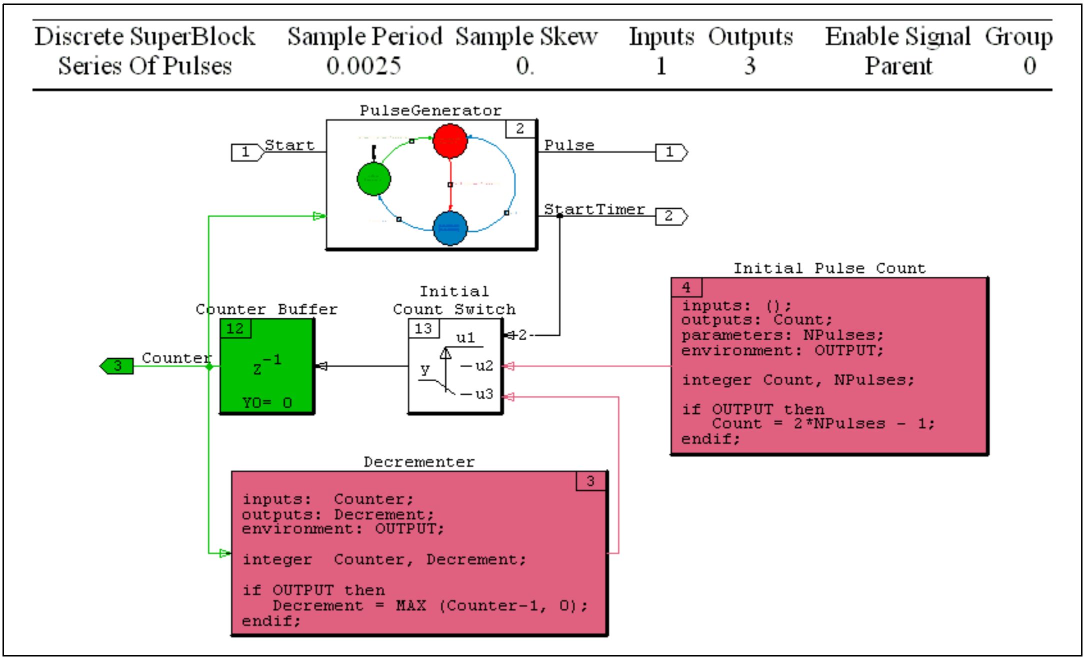

Figure 4-1. Series of Pulses SuperBlock

The pulses values, high and low, are defined by alternation between two states in a State Transition diagram, **pulseHigh** and **pulseLow**. You start the series of pulses by a user command, which comes from an input to the `sim( )` command in Xmath. When the series of pulses is started, a countdown timer is loaded with an initial value. When the timer decrements to zero, the pulses stop. This occurs when the State Transition diagram transitions from the **pulseLow** state to the **idle** state. The **PulseGenerator** State Transition diagram is shown in Figure 4-2.

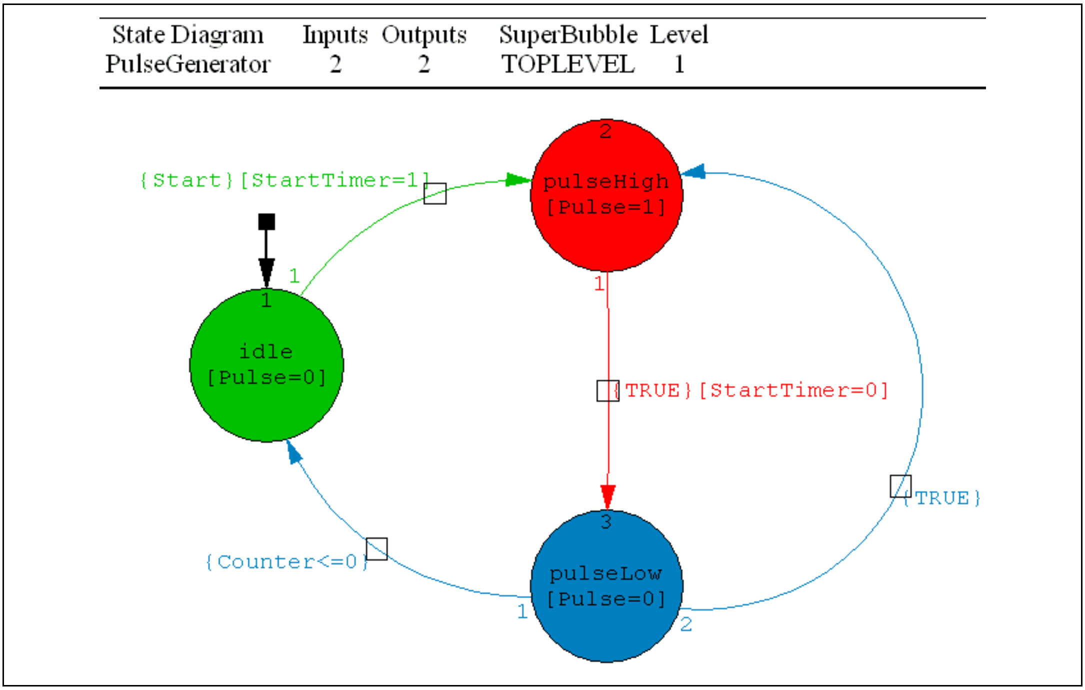

Figure 4-2. PulseGenerator State Transition Diagram

The results of the simulation are shown in Figure 4-3.

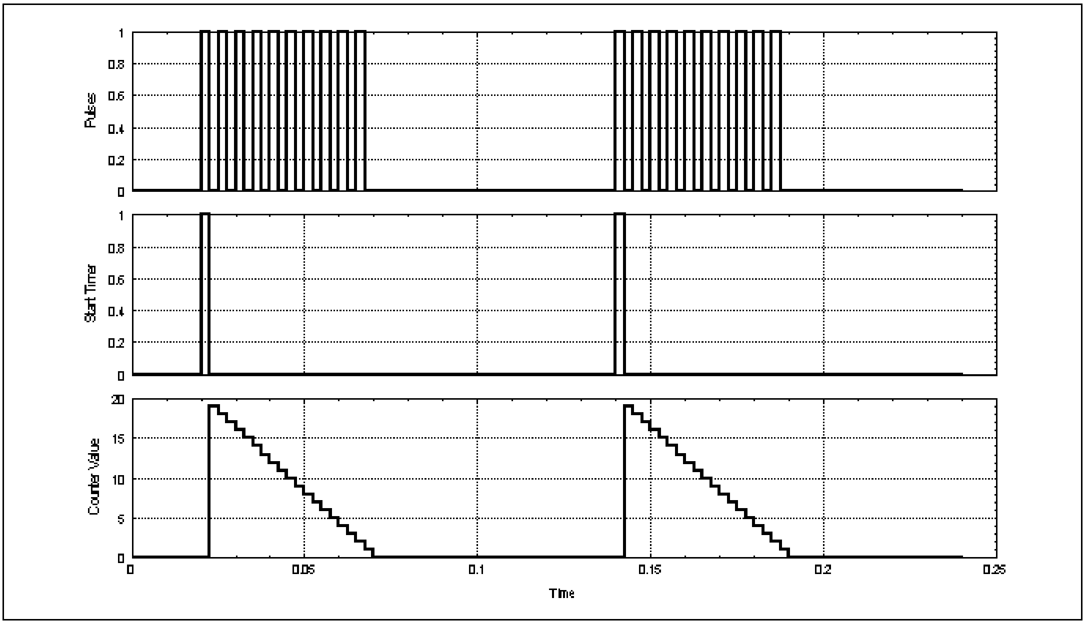

Figure 4-3. Plot Output from Series of Pulse Model

Implementing a Pulse Width, Pulse Frequency Modulator

In this example, you implement a Pulse Width, Pulse Frequency modulator or PWPF. You can find the model in the SystemBuild `examples` directory. To load and run the model, type the following command in the Xmath command area:

```
exec file = "$SYSBLD/examples/BlockScriptPWPF/pwpf.ms"
```

A typical application is in thruster control of spacecraft. In this application, it is difficult to open and close the thruster valves in a continuous fashion. It is more convenient to open the valves all the way for a short moment, followed by closing them completely. The translational position of a spacecraft is controlled by having an array of thrusters, some in direct opposition to the others. The control device sends a digital signal to open the valve to a thruster on one side of the craft causing the spacecraft to move in a direction opposite to that of the thrust. To compensate for overshoot, a signal can be sent to open the valve to a thruster on the other side.

In order to match the desired continuous control signal, you want the energy supplied to the opposing thrusters to match the energy of the desired input, which is defined by some control law. In this example, you choose an arbitrary input signal constructed from sine waves:

```
theInput = sin(t) + 0.25 * sin(3*t)
```

The energy can be calculated as the area under the pulses or the integral of the pulse train. In this application, it is best to vary both the pulse width and the pulse frequency.

The example system is constructed by feeding back the thruster control in a servomechanism loop. The on-off control signal is constructed from a relay with hysteresis and dead zone. The effect of the feedback is to drive the error signal to zero, which also drives the state in the relay into the dead zone. This state is the integral of the difference between input and output:

$$theState = \int Kpf(theInput - Kmf \cdot theOutput)\ dt$$

where:

Kpf is the pulse frequency gain

Kmf is the modulation factor gain

$$MF = modulation\ factor = \frac{TimeOn}{TimeOn + TimeOff}$$

$$PF = pulse\ frequency = \frac{1}{TimeOn + TimeOff}$$

The logic of the relay is shown in Figure 4-4.

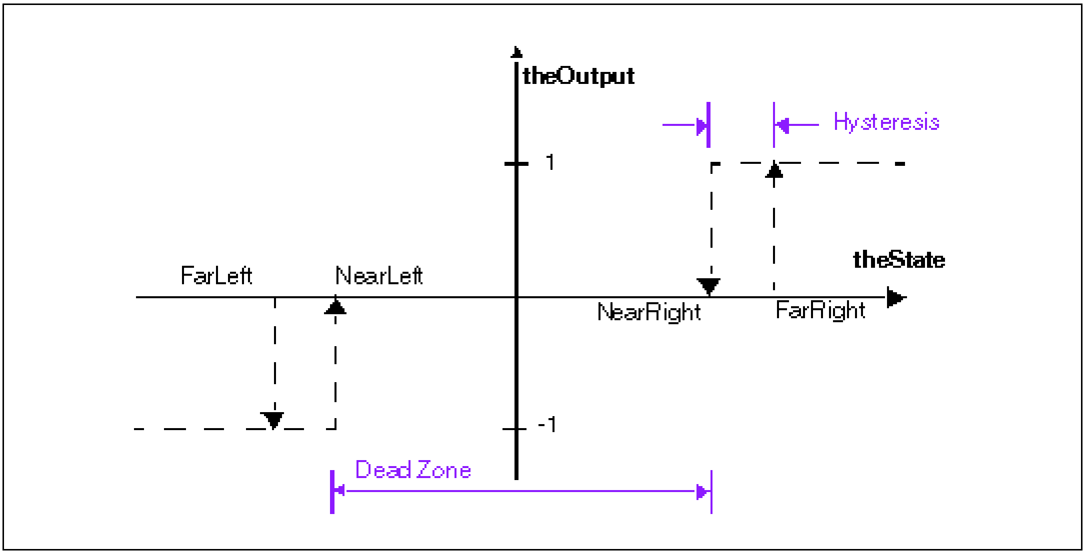

Figure 4-4. Relay Logic

You want the hysteresis in the trigger to make it switch and lock in on the new value so that small variations in the input about the switch point prevent the trigger from chattering back and forth between two values.

The purpose for the dead zone is to conserve fuel. You only want to fire the thrusters when there is enough difference between desired input and delivered output.

Typically, the function is symmetric:

$$\text{Dead Zone} = \text{Far Right} - \text{Near Left} = \text{Near Right} - \text{Far Left}$$

$$\text{Hysteresis} = \text{Far Right} - \text{Near Right} = \text{Near Left} - \text{Far Left}$$

The modulation factor, *MF*, is inversely proportional to *Kmf*. In this example you choose *Kmf* to be unity (1) such that the area under the output follows the area under the input. However, a slight decrease in *Kmf* (for example, to 0.95) makes the pulses wider.

The pulse frequency, *PF*, is proportional to *Kpf*/*Hysteresis*. Either increasing *Kpf* or decreasing *Hysteresis* results in more pulse switching by the relay.

A top-level continuous SuperBlock, **Comparison of PWPF Outputs**, tests the system. The **Pulse FrequencyWidth Modulator** is exercised and its output, as well as the input to the system, are integrated for comparison. Figure 4-5 shows this SuperBlock.

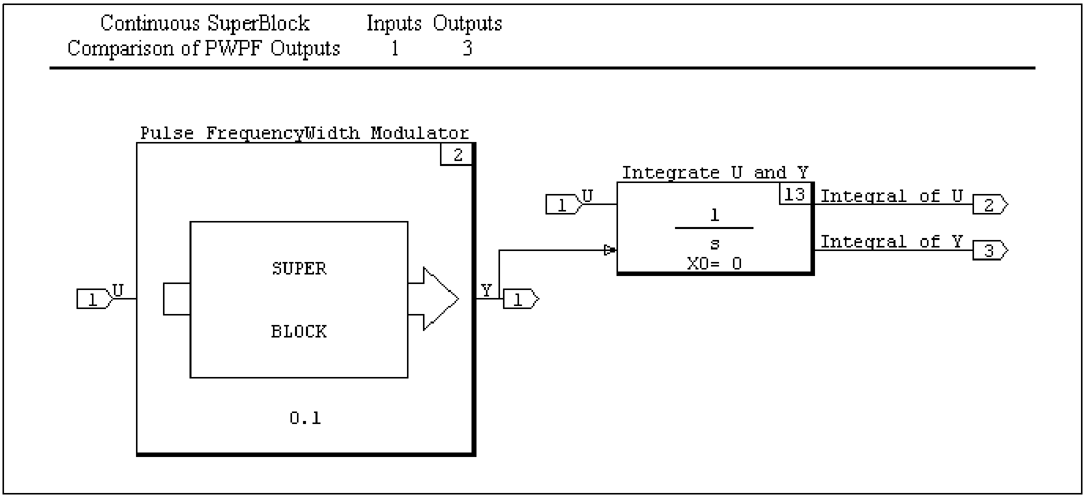

Figure 4-5. Comparison of PWPF Outputs SuperBlock

The **Pulse FrequencyWidth Modulator**, shown in Figure 4-6, is a discrete SuperBlock that contains a BlockScript block with a custom icon named **Relay BlockScript**.

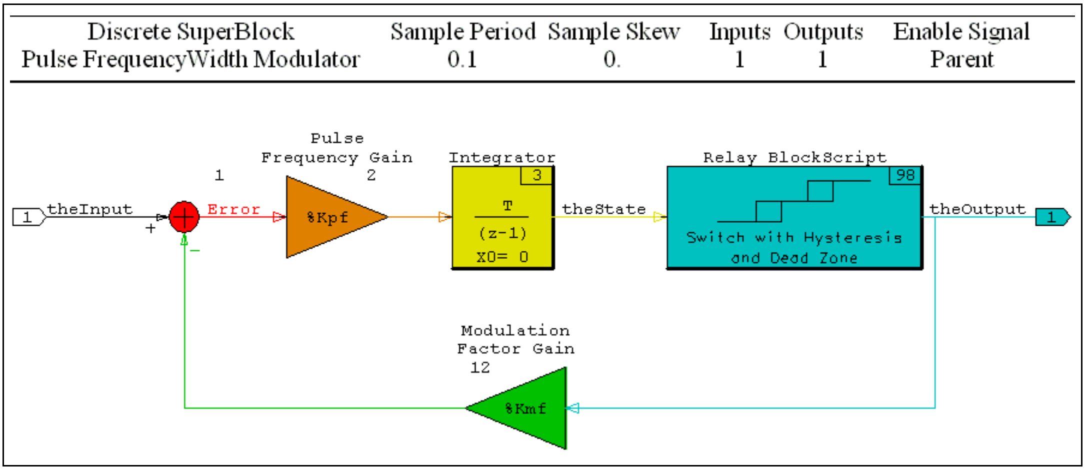

Figure 4-6. Pulse FrequencyWidth Modulator SuperBlock

The first plot that you receive from this model is the input superimposed on the pulse output from the model, as shown in Figure 4-7.

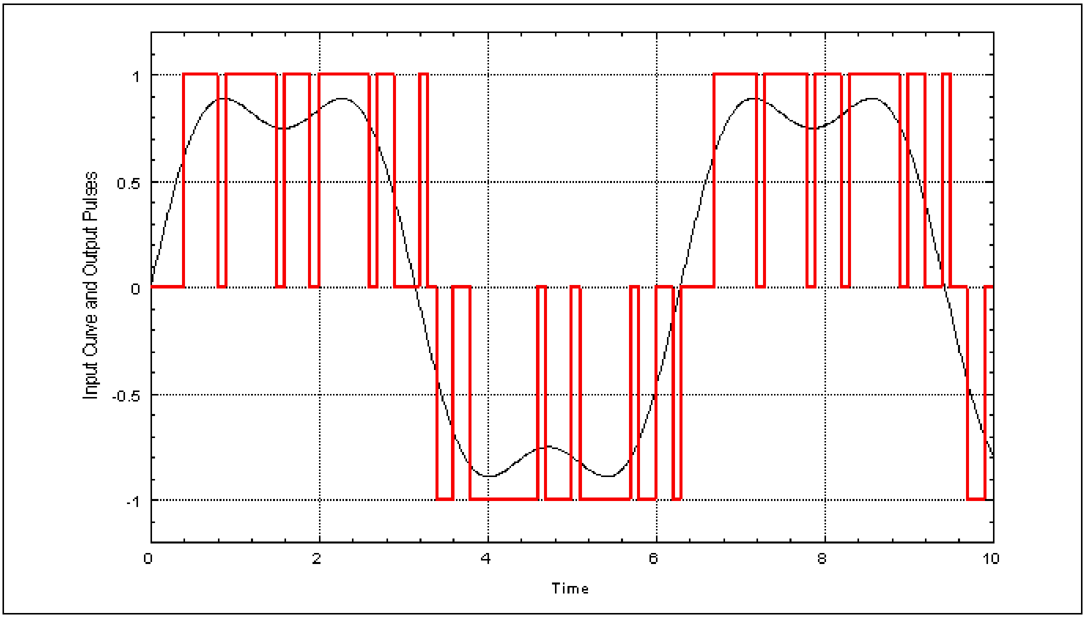

Figure 4-7. Pulse Output Compared with Input to Model

The second plot is the output from the top-level SuperBlock, which compares the integral of the input to the integral of the pulses, as shown in Figure 4-8.

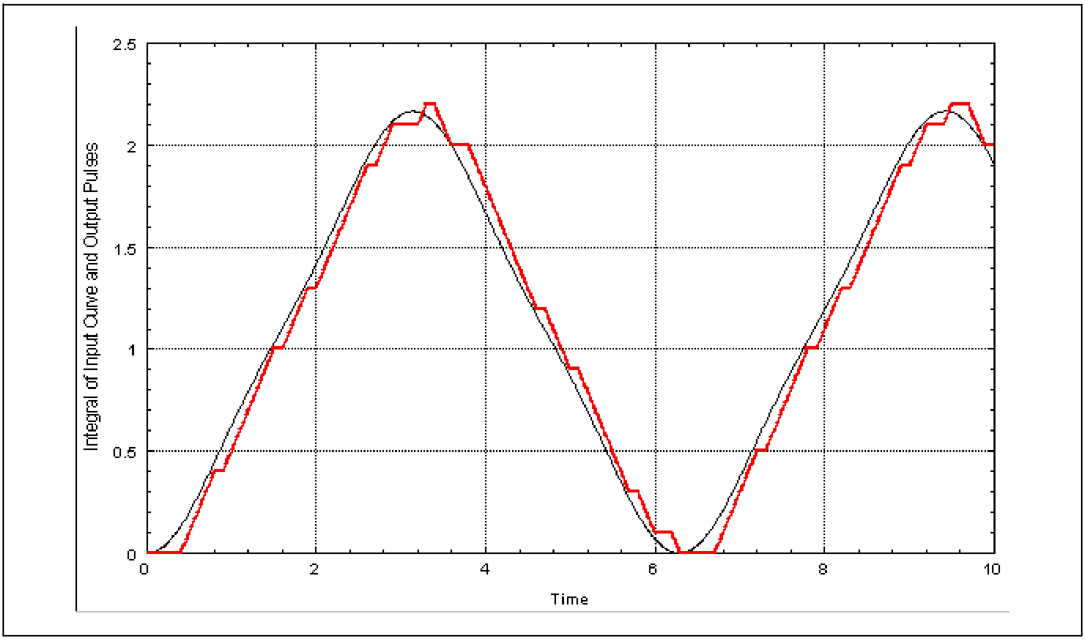

Figure 4-8. Integral of Pulse Output and the Integral of Input to Model

Both *Kpf* and *Kmf* are **%variables** in the Gain blocks. Therefore, you can change these in the Xmath commands area and run the simulation again.

Debugging Tip

With minor modifications, you can include all or part of the body of your BlockScript program in a MathScript function. You then can execute it from Xmath and run it with the MathScript debugger.

Converting BlockScript Blocks to UCBs for Faster Simulations

During simulation, BlockScript statements are interpreted for execution. Other types of blocks are not interpreted and are evaluated by built-in functions. As a result, simulation speed is reduced when you use BlockScript blocks.

A solution is available for AutoCode customers. This method involves placing the BlockScript block or blocks inside a procedure SuperBlock, generating stand-alone procedure code from the SuperBlock, compiling and linking the generated code, and finally invoking the stand-alone procedure as a UserCode block (UCB).

Note This procedure gives enhanced performance at the expense of flexibility. You cannot use **%variables** inside a stand-alone procedure.

To convert a BlockScript block to a UCB for simulation, complete the following steps:

1. In the SuperBlock Editor, create a new SuperBlock. Name it `MYPROC` and set the **Type** field to **Procedure**. Click **OK**.
2. Open the User Programmed palette of the Palette Browser and drag a BlockScript block onto your new SuperBlock. Name the BlockScript block `MYBLOCK`.
3. Write and debug your BlockScript block, or use a block from the examples in this chapter. Upload the SuperBlock to the Catalog Browser.
4. From the Catalog Browser, select **File»New»SuperBlock** to create a new SuperBlock. Name it `MYSUPER`, make its type discrete, and specify at least one output. Click **OK**.
5. Position the Catalog Browser and the SuperBlock Editor that contains `MYSUPER` so that you can see both. In the Catalog Browser, click the SuperBlock hierarchy heading (in the left pane) so that all SuperBlocks are displayed in the Contents view (in the right pane). Locate the SuperBlock `MYPROC` in the Contents view. Drag `MYPROC` from the Catalog Browser into the Editor. Select **File»Update** to make sure the new information appears in the Catalog Browser.
6. Select `MYSUPER` in the Catalog Browser SuperBlock hierarchy. Select **Tools»AutoCode**. In the Generate Real-Time Code dialog box, select **Procedures** in **Code Style** field. Click **OK**.

   The file that is generated, `MYSUPER.C`, is the source code for your stand-alone procedure.
7. Open `MYPROC` in an editor. To replace the BlockScript block, raise the Palette Browser and drag a UCB icon from the User Programmed palette so that it covers the BlockScript block.

   When you release the mouse, the UCB will have replaced the BlockScript block.
8. Open the UCB for editing. In the **Name** field, type `MYPROC`. In the **Function Name** field, type `MYSUPER.C`. Make sure that the **UCB**

Inputs, **Outputs**, and **States** are consistent with the original BlockScript block settings. Click **OK**.

The first time the new SuperBlock is simulated, the procedure code is compiled and linked into your simulator, creating a local version of the `simucb` shared library file. Every subsequent time you run the simulator, the local version is used.

Note 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.

A

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at `ni.com` for technical support and professional services:

- **Support**—Online technical support resources at `ni.com/support` include the following:
  - **Self-Help Resources**—For immediate answers and solutions, visit the award-winning National Instruments Web site for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on.
  - **Free Technical Support**—All registered users receive free Basic Service, which includes access to hundreds of Application Engineers worldwide in the NI Developer Exchange at `ni.com/exchange`. National Instruments Application Engineers make sure every question receives an answer.
- **Training and Certification**—Visit `ni.com/training` for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit `ni.com/alliance`.

If you searched `ni.com` and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of `ni.com/niglobal` to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Index

Symbols

: wildcard, 2-9
|, 3-4

A

abort( ), 3-8
abs( ), 3-8
absolute value, computing, 3-8
ABSTOL environment variable, 2-11
acos( ), 3-8
algorithm, linear interpolation, 4-6
arc cosine function, 3-8
arc sine function, 3-8
arc tangent functions, 3-9
arithmetic expressions, 3-3
assignment statements, 3-3
assignment statements and expressions, 3-3
atan( ), 3-9
atan2( ), 3-9
AutoCode
 function to stop during execution, 3-8
 program phases, 2-2

B

bClear( ), 3-9
Bessel equation example, 4-1
bit
 logical operation functions, 3-9
 shifting functions, 3-9
bitAnd( ), 3-9
bitLshift( ), 3-9
bitNot( ), 3-9
bitOr( ), 3-9
bitRshift( ), 3-9
BlockScript
 assignment statements, 3-3
 block, 2-1
  Bessel equation, 4-1
  converting to UCB, 4-17
  hysteresis, 4-7
  linear interpolation algorithm, 4-6
  PID controller, 4-2
  three-cycle delay, 4-5
 data types, 2-2
 debugging
  in MathScript function, 4-17
  tip, 4-17
 decision-making constructs, 3-5
 examples, 4-1
 expressions, 3-3
 functions provided, 3-8
 language description, 3-1
 looping constructs, 3-5
 operators and precedence, 3-1
 program structure, 2-1
 usage, 1-1
  in SystemBuild, 2-1
 variables, categories, 2-5
bound( ), 3-9
bSet( ), 3-9
bTest( ), 3-9
bToggle( ), 3-9

C

case sensitivity rules, 2-5
Case statement, 3-6
casting function, 2-9
categories, BlockScript variables, 2-5
ceiling( ), 3-11
code generation, data type, 2-11

colon wildcard, 2-9
compiling blocks, 4-17
constructs
  decision-making, 3-5
  looping, 3-5
conventions used in the manual, *iv*
cos( ), 3-12
cosh( ), 3-12

D

data type(s), 2-2, 2-7
  code generation, 2-11
  implicit assignment, 2-7, 2-10
  Logical, 2-11
  rules, 2-7
debugging tip, BlockScript, 4-17
decision-making constructs, 3-5
Derivatives variable category, 2-6
diagnostic tools (NI resources), A-1
dimensioning with wildcard, 2-9
documentation
  conventions used in the manual, *iv*
  NI resources, A-1
drivers (NI resources), A-1

E

Environment variable category, 2-6
environment variable(s), 2-11
  case sensitivity, 2-5
  INIT, 2-4
  OUTPUT, 2-3
  STATE, 2-3
EPSILON environment variable, 2-11
example(s)
  Bessel equation, 4-1
  generating series of pulses, 4-8
  hysteresis BlockScript block, 4-7
  linear interpolation BlockScript block, 4-6
  nonlinear breakpoints block, 2-8
  PID controller BlockScript block, 4-2
  pulse width, pulse frequency modulator, 4-11
  simple BlockScript program, 2-2
  SystemBuild, 4-1
  three-cycle delay BlockScript block, 4-5
examples (NI resources), A-1
Exit statement, 3-7
exp( ), 3-9
expressions
  arithmetic, 3-3
  logical, 3-4
  range, 3-4
  relational, 3-3
  set, 3-4

F

float( ), 3-8
floor( ), 3-11
For loop, 3-5
functions, size of variable, 3-8

H

help, technical support, A-1
hyperbolic trig functions, 3-12
hysteresis example, 4-7

I

If clause, 3-5
INIT environment variable, 2-4, 2-11
Inputs variable category, 2-5
instrument drivers (NI resources), A-1
integer( ), 3-8
intrinsic functions, 3-8
Iterate statement, 3-7

K

KnowledgeBase, A-1

L

language, 3-1
linear interpolation algorithm, 4-6
log( ), 3-10
log10( ), 3-10
Logical data type, 2-11
logical expressions, 3-4
looping constructs, 3-5

M

max( ), 3-10
maximum, function to compute, 3-10
min( ), 3-10
minimum, function to compute, 3-10
mod( ), 3-10
model(s), SystemBuild, 4-1
modulo function, 3-10

N

National Instruments support and services, A-1
Next_States variable category, 2-6
NI support and services, A-1
nRand( ), 3-14

O

operators, 3-1
    precedence table, 3-2
ouRand( ), 3-14
OUTPUT environment variable, 2-3, 2-12
output phase, 2-2
Outputs variable category, 2-5

P

Parameters variable category, 2-6
phases, program
    output, 2-2
    simulation and AutoCode, 2-2
    state, 2-2
PI environment variable, 2-12
PID controller example, 4-2
precedence, BlockScript operators, 3-1
program
    See also example(s)
    phases
        determining, 2-11
        environment variables, 2-3
        for simulation and AutoCode, 2-2
    structure, 2-1
programming examples (NI resources), A-1
pulse width, pulse frequency modulator, 4-11
pulses, example generating a series, 4-8

Q

quad( ), 3-10
    graphical diagram of results, 3-11
quantizing floating point numbers, functions, 3-11

R

random number generator functions, 3-14
range expressions, 3-4
relational expressions, 3-3
RELTOL environment variable, 2-12
round( ), 3-11
rules
    data typing, 2-7
    dimensioning with wildcard, 2-9

S

Select clause, 3-6
set expressions, 3-4
sign( ), 3-11
    graphical representation of function, 3-12
simulation
    function to stop, 3-8
    program phases, 2-2
sin( ), 3-12
sinh( ), 3-12
size of variable functions, 3-8
software (NI resources), A-1
sqrt( ), 3-12
square root function, 3-12
STATE environment variable, 2-3, 2-12
state, phase, 2-2
States variable category, 2-6
support, technical, A-1
swap( ), 3-12
SystemBuild
    BlockScript block paradigm, 1-2
    models, 4-1

T

tan( ), 3-12
tanh( ), 3-12
technical support, A-1
three-cycle delay BlockScript block, 4-5
TIME environment variable, 2-12
training and certification (NI resources), A-1
trg( ), graphical representation, 3-13
trig functions, 3-8, 3-9, 3-12
troubleshooting (NI resources), A-1
truncate( ), 3-11
TSAMP environment variable, 2-12
TSTART environment variable, 2-12
Typecheck with BlockScript block, 2-11

U

union operator |, 3-4
uRand( ), 3-14

V

var.columns, 2-7, 3-8
var.rows, 2-7, 3-8
var.size, 2-7, 3-8
var.type( ), 2-9, 3-8
variable(s)
    BlockScript categories, 2-5
    case sensitivity, 2-5
    default names, 2-5
    definition in BlockScript program, 2-2
    determining dimension, 2-7
    dimensioning with wildcard, 2-9
    environment
        case sensitivity, 2-5
        delineating phases, 2-3
    functions for size of, 3-8
    list order, 2-5
    Logical, 2-11
    parameterized, limit, 2-7
    passing data between phases, 2-5
    restricted usage in While loops, 3-5
    types for SystemBuild programs, 2-4
vertical bar usage, 3-4

W

Web resources, A-1
While loop, variable usage in, 3-5
wildcard, 2-9