

MATRIX[®]_x

7.0

AUTOCODE[®] 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, PersonalJWorks, pSOS+, pSOSim, pSOSystem, SingleStep, SNiFF+, VxDCOM, 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	Manual Organization	1
1.2	Rapid Prototyping Concept	2
1.3	Automatic Code Generation Process	5
1.4	Using AutoCode with BetterState	7
1.5	Profile of the Generated Program	9
1.6	AutoCode-Generated Reusable Procedures	12
1.7	Using MATRIX _x Help	12
1.8	Related Publications	13
2	Using AutoCode	15
2.1	How to Generate Real-Time Code	15
2.1.1	Generating Code from Within SystemBuild	16
2.1.2	Generating Code from Xmath	16
2.1.3	Generating Code from the Operating System	17
2.1.4	Limitations/Restrictions	18

2.2	Generating Non-Customized Code	19
2.3	Generating Customized Code	21
2.4	Using Templates	24
2.5	Applications of AutoCode-Generated Code	25
2.5.1	Standalone Simulation	26
	Standalone Simulation	28
2.5.2	Simulation Options	29
2.5.3	Rapid Prototyping	30
2.5.4	Real-Time Simulation	31
2.5.5	Implement Embedded Real-Time Control	31
2.6	How to Integrate Generated Code into Your Target	31
2.6.1	Loading the Fixed-Point Demo	32
2.6.2	Determining System Scaling	34
2.6.3	Modular Programming	37
2.6.4	Comparing the Output	40
2.6.5	Implementation	41
2.6.6	Optimizations	43
2.6.7	Integration and Test	44
2.7	How to Write Production Quality Code (Graphically)	45
2.7.1	Graphical Solutions	46
	Design Abstractions	46
	Code Abstractions	47
2.7.2	Labels and Names	48
2.7.3	Modular Programming Through Procedures	49
3	Using AutoCode with BetterState	53
3.1	Procedural and Event-Driven BetterState Charts	53

3.2	Generating Code for a BetterStateChart Block	54
3.2.1	Handling BetterState Charts That Call Procedures	57
3.2.2	Handling BetterState Charts That Read or Write Variable Blocks	57
3.3	Using BlockScript User Code	58
4	Managing and Scheduling Applications	59
4.1	Real-Time Application Scheduler	59
4.1.1	Subsystems	60
4.1.2	Flow of Control in the Generated Program	61
4.2	Sequence of Scheduler Operations	63
4.3	Properties of Scheduled Subsystems	69
4.3.1	Free-Running Periodic Subsystems	70
4.3.2	Enabled Periodic Subsystems	71
4.3.3	Triggered Subsystems	74
4.4	Properties of Asynchronous Subsystems	78
4.4.1	Start-up Procedure	79
4.4.2	Asynchronous Trigger Subsystems	81
4.4.3	Interrupt Procedure	82
4.4.4	Background Procedure	82
4.5	Reentrancy and Preemption: The Dispatcher	83
4.6	Scheduler Examples	84
4.6.1	Dispatching and Pre-emption Example	84
4.6.2	Pseudo-Rate Scheduler	91
4.6.3	Operating with Skew	93
4.7	Scheduler Errors	95
4.7.1	Scheduler or Subsystem Overflow	95

4.7.2	Examples Where Overflow is Irrelevant or Cannot Happen	96
5	Code Generation for Discrete Systems	99
5.1	Introduction	99
5.2	How to Generate Code for Discrete Systems	100
5.3	Introduction to Vectorized Code	100
5.4	Introduction to Optimized Code	101
5.5	Introduction to Procedural Code	101
5.6	Sample Generated Code	102
5.6.1	Sample C Code	102
5.6.2	Sample Ada Code	106
6	Code Generation for Continuous Systems	111
6.1	Introduction	111
6.2	Integrators	112
6.3	Limitations	113
6.4	How to Generate Code for Continuous or Hybrid Systems	113
6.4.1	Generating Code for Continuous Systems from SystemBuild ...	114
6.4.2	Xmath Command Options for Continuous Code Generation	114
6.4.3	OS Command Options for Continuous Code Generation	115
6.5	Sample Generated C Code	117
6.6	Sample Generated Ada Code	123
6.7	Hints	127
7	Using VxWorks with AutoCode	129

7.1	Template Features	129
7.2	Generating Code	130
7.3	Code Testing Method	131
7.4	Increasing SIMNT Memory Size	134
7.5	Usage Notes	136
8	Customizing AutoCode and Generated Code	139
8.1	Introduction	139
8.2	AutoCode Configuration Options	140
8.3	Templates	140
8.4	BlockScript Block	140
8.5	Data Parameterization	142
8.6	UserCode Block	143
8.7	Macro Procedure Block	143
8.8	ZeroCrossing Blocks and Resettable Integrators	144
8.9	User-Defined Code Comments	144
	8.9.1 Using a User-Defined Code Comment	145
	8.9.2 Limitations	146
9	Introduction to Software Constructs with AutoCode	147
9.1	Introduction	147
9.2	Standard Procedure SuperBlocks	148
9.3	Variable Blocks	148
	9.3.1 Global	148

9.3.2	Local	149
9.4	Graphical Software Constructs	149
9.5	IfThenElse Block	150
9.5.1	IfThenElse Block Example	150
9.5.2	Looping	152
9.5.3	Ordering or Sequencing the Flow of Data and Calculations	153
9.5.4	Using Local or Global Variables	153
9.5.5	Other Coding Considerations	153
9.6	Iterator Block	154
9.7	Explicit Block Sequencing	154
9.8	Example Model	155
A	AutoCode Options	161
A.1	Options When Invoking AutoCode	161
A.2	Using the autostar.opt File	169
A.3	Mapping Options	171
A.3.1	Setting Subsystem Priorities	171
A.3.2	Setting Subsystem Skews	173
A.3.3	Setting Processor Subsystem Map	173
A.3.4	Processor Map Specification from the OS Prompt	174
B	Software Development Kit	175
B.1	Scope	175
B.2	Supported Versions and Languages	176
B.3	Overview	177
B.3.1	Procedures-Only SystemBuild Model	177

B.3.2	Limitations	177
B.3.3	Application Programming Interface	177
	Interface Structure	178
	Initialization() Function	178
	Execute() Function	178
B.3.4	Driver Program	178
B.4	C API	178
B.4.1	Logical Design	179
B.4.2	Interface Structure	179
B.4.3	Initialize() Function	179
B.4.4	Execute() Function	180
B.4.5	Physical Design (c_sdk.tpl)	180
	Header File	181
	Source File	181
B.4.6	Physical Design (c_sdk_m.tpl)	181
	Header Files	181
	Source File	181
B.4.7	Compilation and Link Details	182
	Standalone Library Header Files	182
	Platform Indicator	182
	Standalone Library Source Files	182
B.5	Sample Code (C-SDK) Example	183
B.6	Wheel Program (C-SDK) Example	185
B.7	C++ API	188
B.7.1	Logical Design	188
	Class	188
	Private Data Member	188
	Constructor Method	189
	Execute Method	189
B.7.2	Physical Design (cpp_sdk.tpl)	189

	Header File	190
	Source File	190
B.7.3	Physical Design (cpp_sdk_m.tpl)	190
	Header Files	190
	Source File	191
B.7.4	Compilation and Link Details	191
B.7.5	Example 3: Sample Code (CPP-SDK)	191
	Header File (interfaces.h)	191
	Driver Program (main.cpp)	193
B.7.6	Example 4: Wheel Program (CPP-SDK)	193
	Step 0: Set up	193
	Step 1: Generate API	194
	Step 2: Compile and Link API Functions and Driver Program ..	194
	Driver Program (wheeldriver.cpp)	195
B.8	Ada API	196
B.8.1	Logical Design	196
	Package	196
	Interface Record	196
	Initialize Procedure	197
	Execute() Function	197
B.8.2	Physical Design (package feather_ext_pkg)	198
B.8.3	Compilation and Link Details	198
B.8.4	Example 5: Sample Code (Ada-SDK)	198
	Package Specification	198
	Driver Program	199
C	Sample AutoCode Output	201
Index		217

1

Introduction

This manual provides an overview of the automatic code generation process using AutoCode[®]. With AutoCode you will soon be automatically generating robust, high-quality, real-time C or Ada source code from SystemBuild[™] block diagrams.

1.1 Manual Organization

This guide provides the information you need to get started using AutoCode[®]. This guide is organized as follows.

- Chapter 1 (this chapter) provides an overview of the rapid prototyping concept, the automatic code generation process, and the nature of the real-time generated code.
- [Chapter 2, Using AutoCode](#), explains how to generate real-time code by invoking AutoCode from SystemBuild[™], Xmath[®], or the operating system prompt. Generated code applications are also discussed.
- [Chapter 3, Using AutoCode with BetterState](#), details the management of the application control flow via the real-time scheduler.
- [Chapter 4, Managing and Scheduling Applications](#), details the management of the application control flow via the real-time scheduler. Topics of discussion include scheduler operation sequence, subsystem properties, subsystem interruption, and examples of scheduler operation.

- [Chapter 5, Code Generation for Discrete Systems](#), describes the scheduler architecture as it relates to discrete code generation. Topics include IPAR and LPAR.
- [Chapter 6, Code Generation for Continuous Systems](#), describes the scheduler architecture as it relates to continuous code generation. Topics include fixed-step integrators, user-defined integrators, and how to generate code for continuous and hybrid systems.
- [Chapter 7, Using VxWorks with AutoCode](#), describes the VxWorks AutoCode C template package with MATRIX_X® 7.X and Tornado 2.
- [Chapter 8, Customizing AutoCode and Generated Code](#), provides advanced methods for customizing AutoCode and its output real-time code using AutoCode configuration options, templates (see the *Template Programming Language User's Guide*), BlockScript, and %variables.
- [Chapter 9, Introduction to Software Constructs with AutoCode](#), describes UserCode Blocks, Macro Procedure Blocks, and Procedure SuperBlocks.
- [Appendix A, AutoCode Options](#), describes options that can be used when invoking AutoCode from within the Xmath Commands window or from the OS prompt. This appendix also describes how to use an **autostart.opt** file.
- [Appendix B, Software Development Kit](#), describes the AutoCode Procedure Software Development Kit (ACP SDK). The SDK provides users of AutoCode generated code an Application Programming Interface (API) to generated Standard and Startup Procedure SuperBlock code.
- [Appendix C, Sample AutoCode Output](#), shows the generated AutoCode output from the IfThenElse example shown in [Chapter 9](#).

This guide also has an [Index](#).

For more advanced details and information necessary to customize both AutoCode and the generated real-time output code, see the *Template Programming Language User's Guide* or the *AutoCode Reference*.

1.2 Rapid Prototyping Concept

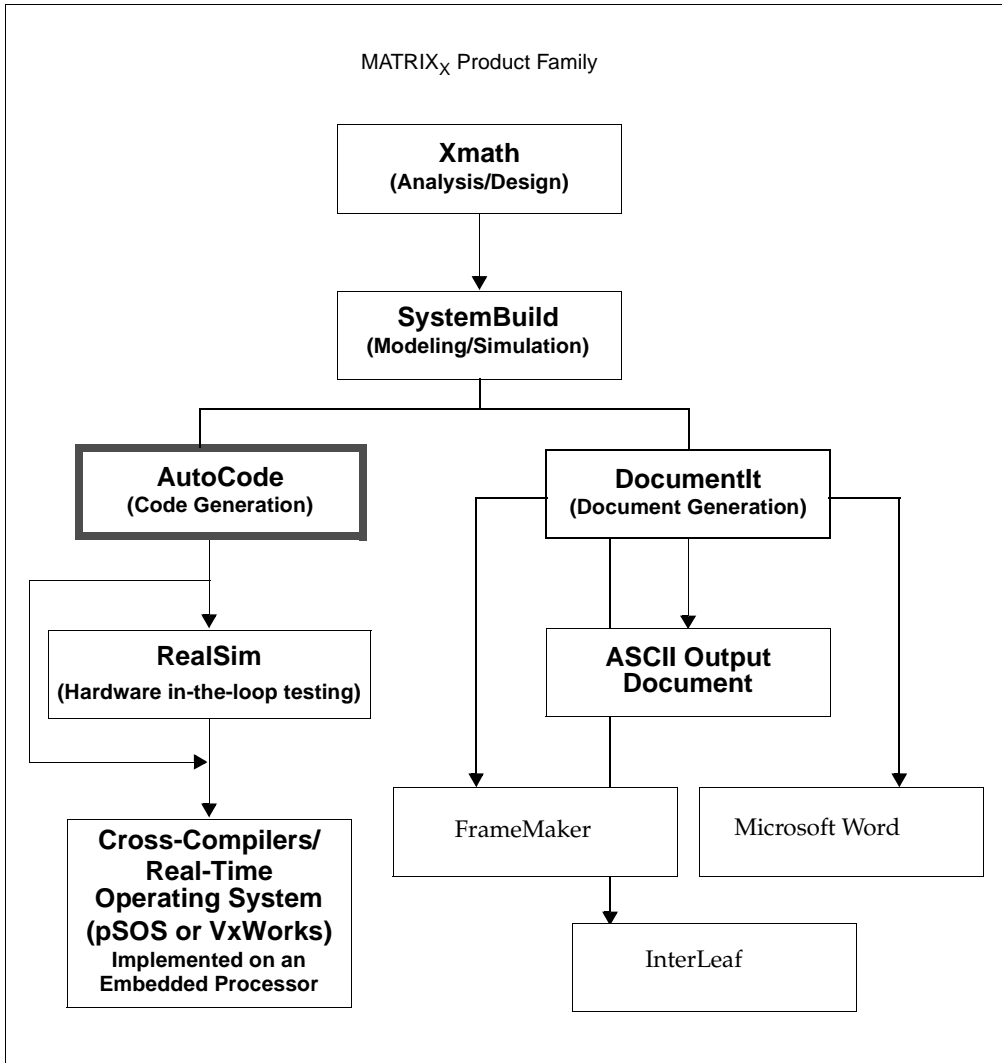
Conventional real-time system development usually takes place in stages, with separate tools for control design, software engineering, data acquisition, and

testing. The MATRIX_X Product Family integrates tools for each stage of system development into a single environment. This allows a design to move easily from one stage to the next, making it possible to create a working prototype early in the design process.

Within the MATRIX_X SystemBuild and Xmath products, you can build, simulate, analyze, test, and debug a model. You can then use AutoCode to generate real-time code in a high-level language (C or Ada) for the model. The generated application code can be evaluated on the host with SystemBuild simulation or run on the RealSim™ controller for hardware in-the-loop testing. The generated application code can be cross-compiled and linked for implementation on an embedded processor. You can also use DocumentIt™ to generate documentation.

Figure 1-1 shows AutoCode in the MATRIX_X product line.

Figure 1-1 AutoCode in the MATRIX_x Product Line



1.3 Automatic Code Generation Process

As an integral part of the Wind River Rapid Prototyping concept, AutoCode lets you generate high-level language code from a SystemBuild block diagram model quickly and automatically. A typical sequence for using AutoCode is as follows (this sequence corresponds to the sequence shown in [Figure 1-2](#), p.6):

1. Build the Model and Validate Through Simulation

You can quickly develop the continuous-time plant model and corresponding discrete-time controller SuperBlocks using SystemBuild block diagrams. The SystemBuild model is built up from a large palette of blocks that combine to describe the way that the model works and how it should be controlled. You can then analyze and simulate the plant and controller in Xmath; if you find any errors, you can easily amend the model and simulate it again until you are satisfied with its performance. You can perform parametric studies in simulation and pass the values from the Xmath workspace into the generated code.

2. Customize Code Generation

You can tailor your generated code using the template programming language (TPL), provided in AutoCode. This programming language lets you customize the code for a wide variety of specialized purposes. Run-time parameterization can be programmed into the generated program. Also, configuration information can be entered when the real-time code is generated.

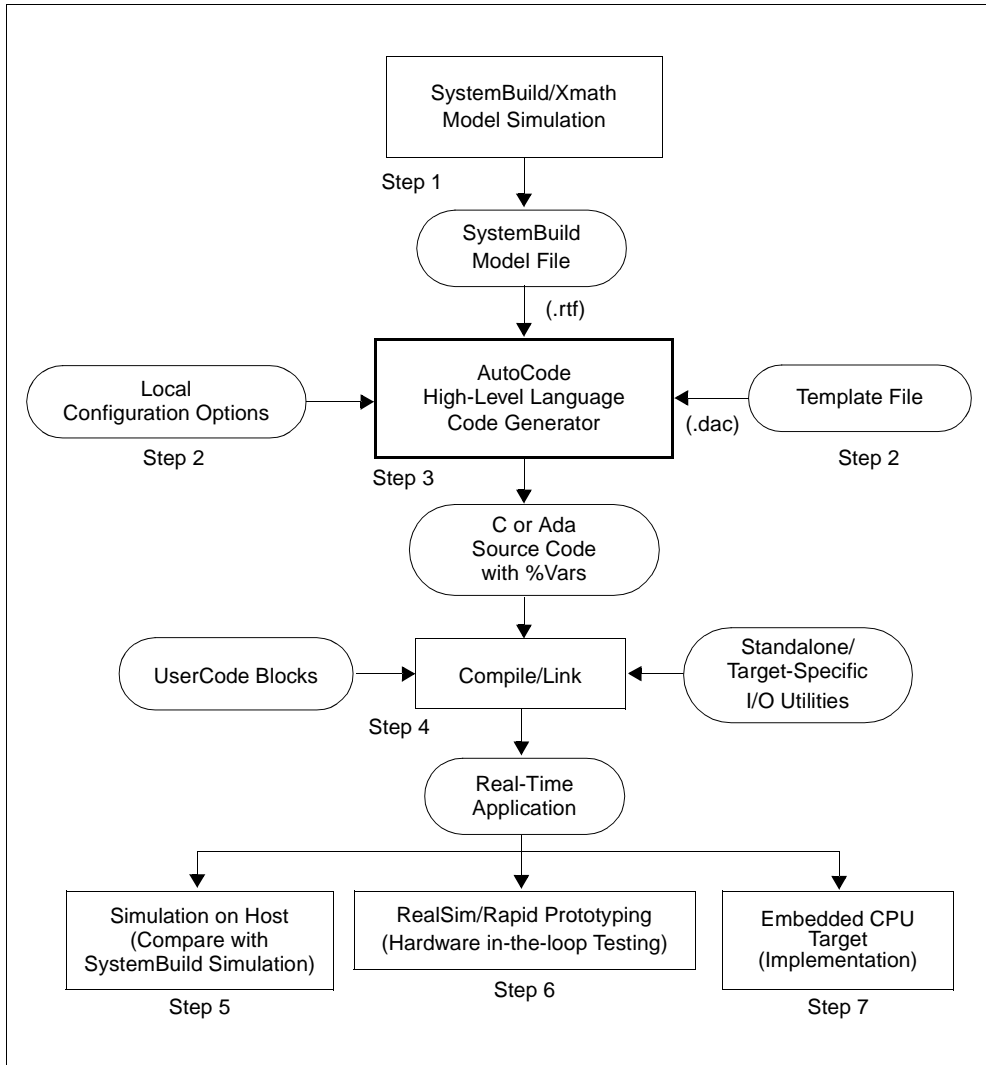
3. Generate the Real-Time Code

You can invoke AutoCode from inside SystemBuild, from the Xmath Commands window, or from the operating system command line. AutoCode processes discrete-time and continuous-time SuperBlocks to generate high-level language code in C or Ada.

4. Compile and Link

You can customize the environment in which the generated code runs by editing standalone Input/Output utilities files included with AutoCode. Additionally, you can further enhance the functionality of your model by adding UserCode Blocks. Compile and link the generated code with these standalone files and any UserCode Blocks that you will be using to produce a standalone real-time application program for simulation on the host. Refer to [2.5](#), p.25.

Figure 1-2 AutoCode Automatic Code Generation Process



5. Validate the Generated Code Through Simulation Comparison

Now you can test and simulate the generated code on the host and feed the results back to Xmath for comparison with the SystemBuild simulation data. These steps are described in 2.5, p.25.

6. Test with Real or Prototyped Hardware

You can use a rapid prototyping tool such as RealSim to implement a real-time controller, or to perform real-time hardware-in-the-loop testing with actual or emulated hardware. RealSim customers are provided with special templates and target-specific utilities to generate and link code specific to RealSim.

7. Implement the Finished Code on the Target

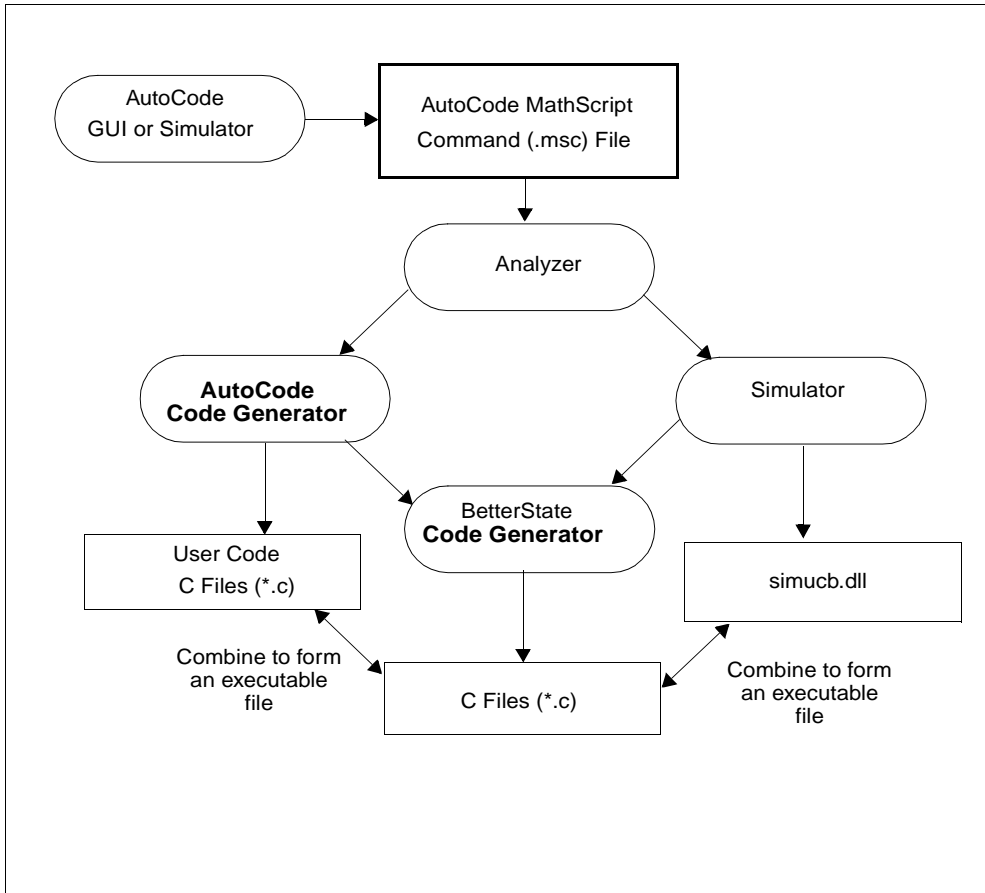
After you have completed all needed testing and simulation to optimize the functionality and performance of your application, the perfected code can be implemented on the target processor.

1.4 Using AutoCode with BetterState

The AutoCode code generation command includes a call to request BetterState to generate code for its statecharts. AutoCode does not generate code for BetterState charts directly. The BetterState code generator interfaces with BlockScript to produce the correct output language from the model. The generated code handles interfacing to AutoCode and simulation. This interface supports calling SystemBuild procedures and access to SystemBuild global variables.

The process for generating code for AutoCode and BetterState, from the AutoCode GUI or the simulator, is shown in [Figure 1-3](#).

Figure 1-3 **Generating Code for AutoCode and BetterState**



The AutoCode-BetterState interface uses the AutoCode Software Development Kit (SDK) interface which is described in [Appendix B](#).

The AutoCode-BetterState interface provides the following features:

- Fixed-point libraries (standalone utilities and more source code files)
- Variable step-size solver in AutoCode
- Code Generation support for VxWorks
- makefile generation with the **acmake** command

- Name mangling feature (new flag for Code Generator; AutoCode changes names in the user's model)

For more information about this interface, see [Chapter 3, Using AutoCode with BetterState](#). For information about using VxWorks with AutoCode, see [Chapter 7](#).

1.5 Profile of the Generated Program

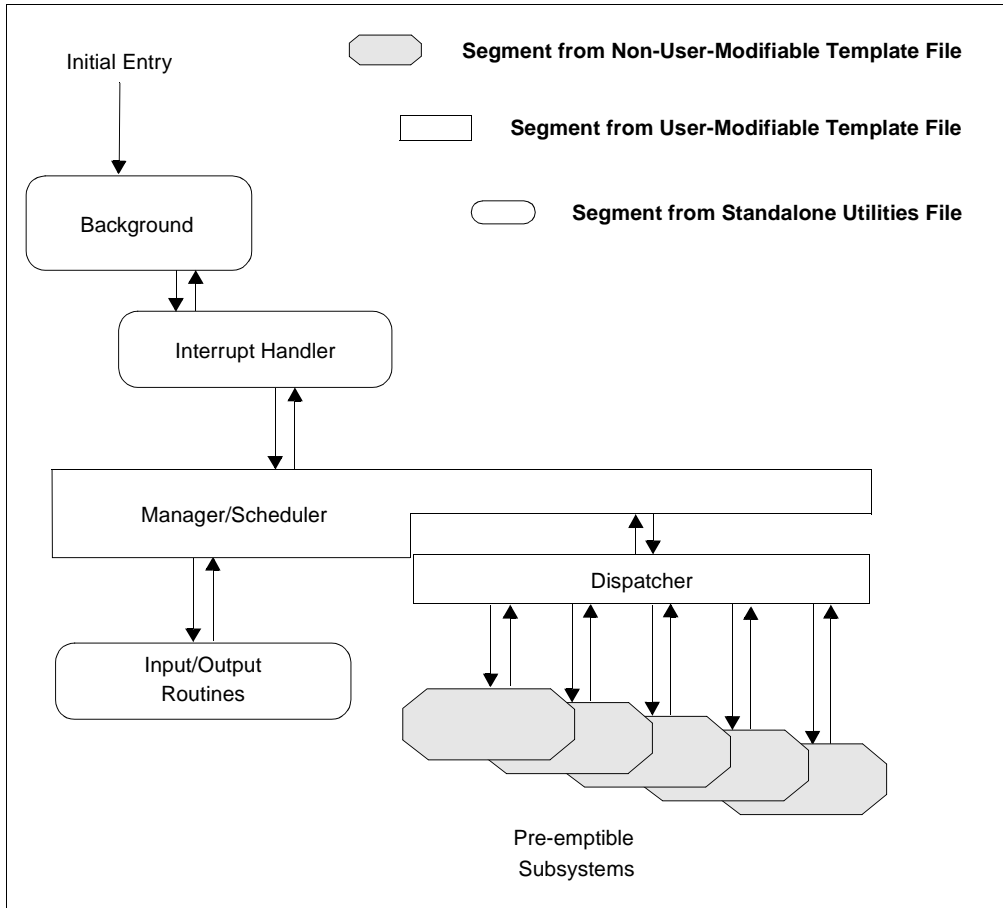
If no user-originated changes are made to the template program, the generated application program consists of calls to a time-critical application manager/scheduler, a re-entrant dispatcher, one or more pre-emptible subsystems, input/output functions, a timer interrupt handler, and a background function (see [Figure 1-4](#), p.10). The calls to the modules of the generated program are in the template program, which lets you modify the way the generated program is structured and expand it as needed. Under control of the default template file, the application program is assembled from components taken from a variety of files, with different provisions for user modification, as explained in the following:

- Manager/Scheduler

The manager/scheduler is a time-critical routine that performs external input/output functions for the application program, takes care of various housekeeping tasks, and generates a dispatch list of the subsystems that are ready to be executed.

Tailoring: This routine can be customized by modifying or rewriting the supplied real-time scheduler. Under control of the template file, any number or variety of scheduler programs can be used.

Figure 1-4 Components of the Generated Application Program



- **Dispatcher**

The dispatcher dispatches the subsystems that are ready to be executed from the dispatch list in a prioritized order. Highest priority subsystems get dispatched first.

Tailoring: Dispatcher logic can be customized by modifying the template file.

- **Subsystems**

The subsystems contain the code implementing block algorithms generated from the SystemBuild model. They implement real-time activities by

accepting inputs and posting outputs at times derived from the sampling rates of the SuperBlocks in the SystemBuild model, under control of the manager/scheduler.

Tailoring: Customization of block code is supported only via the BlockScript and UserCode blocks. The block code for other block types is proprietary. Refer to *8.4 BlockScript Block*, p.140 for details regarding BlockScript.

- I/O Routines

The main function of I/O routines is to provide input data to the AutoCode real-time application on every scheduler cycle and to obtain the computed outputs from it. The supplied I/O routines read inputs from a MATRIX_X formatted ASCII file and write outputs to a file in the same format.

Tailoring: The input/output routines are part of the standalone utilities file and are intended to be user modified. Any variety of I/O routines can be user-written and invoked as needed under control of the template program.

- Timer Interrupt Handler

The timer interrupt handler calls (or invokes) the manager/scheduler at a specified time interval. This is not needed for standalone simulation on a host.

Tailoring: This routine is intended to be user-defined, such that it invokes the scheduler on every minor cycle.

- Background Function

The background function performs idle time non-time-critical tasks such as self-diagnosis or updating a display; its essential qualification is to be interruptible. For standalone simulation, it merely calls the scheduler for user-specified simulation cycles as specified by time vector.

Tailoring: This is part of the standalone utilities file that can be user-modified or completely rewritten.

For a detailed explanation of the flow of control in the generated program, see [Chapter 4, Managing and Scheduling Applications](#).

1.6 AutoCode-Generated Reusable Procedures

In the earlier sections of this chapter, we looked at the process of generating a real-time scheduled application program and its nature. However, in some cases, you might not want to generate scheduler and related data structures. Rather, you might want to generate merely algorithmic procedures (subroutines). AutoCode provides an option to let you generate only algorithmic procedures from Procedure SuperBlocks, without the scheduler and its data structures.

Following are some uses of generating reusable algorithmic procedures:

- Linking them to your own real-time scheduler (executive) or simulator
- Linking them to the SystemBuild simulator for algorithmic verification and speeding simulation by using integer math in the procedure

The AutoCode generated procedures link to the SystemBuild simulator via a UserCode Block in the SystemBuild model. The procedure describing how to link them is provided in Chapters 2 and 3 of the *AutoCode Reference*.

1.7 Using MATRIX_X Help

MATRIX_X 7.X provides a hypertext markup language (HTML) help system. The MATRIX_X Help system is a self-contained system with multiple hypertext links from one component to another. This help system, augmented by online manuals, covers most MATRIX_X topics except for installation. For installation information, see your online or printed manuals.

The MATRIX_X Help system requires Netscape Communicator 4.03 (included on the MATRIX_X CD) or later. On UNIX systems, an OEM version of Navigator is automatically included in the MATRIX_X installation. On PC systems, Netscape Communicator must be installed independently using the Netscape installation procedure included on the MATRIX_X CD.

Additional Netscape Information

For more information on Netscape products, see Netscape's home page at www.netscape.com.

1.8 Related Publications

Wind River provides a complete library of publications to support its products. In addition to this guide, publications that you may find particularly useful when using AutoCode include the following:

- *Xmath User's Guide*
- *SystemBuild User's Guide*
- *BetterState User's Guide*
- *BlockScript User's Guide*
- *RealSim User's Guide*
- *AutoCode Reference*
- *Template Programming Language User's Guide*
- *DocumentIt User's Guide*

For additional documentation, see the MATRIX_X Help or the Wind River home page at www.windriver.com.

2

Using AutoCode

This chapter explains how to generate real-time code by invoking AutoCode from SystemBuild, the Xmath Commands window, or the operating system prompt. This chapter also discusses generated code applications.

2.1 How to Generate Real-Time Code

Using AutoCode, you can generate C or Ada high-level language code from:

SystemBuild — generates a real-time file (.rtf) and then source code from a model, after selecting it from the Catalog Browser. For ease of use, this is the recommended method of code generation.

Xmath — generates an .rtf file and then source code from a model, using an Xmath command.

Operating system prompt — generates source code from an already-existing .rtf file, using the **autostar** command from the operating system prompt.

For Xmath Commands window or operating system command options, see [Appendix A, AutoCode Options](#).

2.1.1 Generating Code from Within SystemBuild

To use AutoCode while inside SystemBuild, select a SuperBlock in the Catalog Browser, and then select Tools→AutoCode to open the dialog. Instructions for using this dialog are in the MATRIX_x Help.

Depending on the template file and command options used, the code generated can be either C code or Ada code.

2.1.2 Generating Code from Xmath

The **autocode** command lets you process a model to generate C or Ada code. Two syntax formats are supported:

```
autocode, {model = name1, file = name2, ...  
language = name3, tpldac = name4, ...  
rtf = name5, vars, typecheck}
```

```
autocode model, {options}
```

where *name1* identifies the model to be processed for code generation. The model will be either:

- A string (in “quotes”). Must be the name of a SuperBlock that exists in the current SystemBuild catalog. This SuperBlock is analyzed and processed to generate code.
- A variable (not in quotes). Variables should be assigned to a string, the string must be the name of a SuperBlock in the current catalog; it is analyzed and processed to generate code.

Whenever a file name or other string is included in a command string, it must be enclosed in quotes, but a variable name must *not* be in quotes.

Keywords for the second type of syntax are the same as for the first syntax, except **model**. See [Appendix A](#) for the Xmath command options.

Examples:

```
autocode "topSB"
```

The system generates a real-time file named *topSB.rtf*. It loads this file and processes it to produce C code. The output file name is *topSB.c*.

```
autocode "topSB", {tpldac="mytemplet",!vars}
```

```
autocode, {model = "topSB", tpldac = "mytemplet", !vars}
```

Either syntax processes the SuperBlock *topSB* in the current catalog to produce C code, using the direct access template file *MyTemplate* and no Xmath %variables. The output file name is *topSB.c*.

2.1.3 Generating Code from the Operating System

If a model file already exists, it is also possible to execute AutoCode from the operating system prompt. The file intended for processing must be a real-time file (.rtf). At the operating system prompt, execute the command:

```
% autostar {options} model_file.rtf
```

Many of the options are the same as the fields in the Generate Real-Time Code dialog. See [Appendix A](#) for the operating system command options.

AutoCode runs, creating a high-level language file. When the operating system prompt returns, the process is complete.

Examples:

```
% autostar -h
```

shows a help display.

```
% autostar -l c SysBld_file.rtf
```

processes the model file *SysBld_file.rtf* to produce a C code file named *SysBld_file.c*. All default settings are accepted. It assumes a direct access template file named *c_sim.dac* exists in your working directory.

```
% autostar -l a -t ada_rt.tpl -sd 6 -o CodeFile.a MyModel.rtf
```

processes the ASCII template file *ada_rt.tpl* and produces a direct access template file *ada_rt.dac*. The model file *MyModel.rtf* is then processed, producing Ada code in file *CodeFile.a*, which contains numeric literals encoded using a maximum of 6 significant digits of precision.

```
% autostar -l c -t c_sim.tpl
```

compiles the template file *c_sim.tpl* to produce a .dac file named *c_sim.dac*.

2.1.4 Limitations/Restrictions

- SystemBuild models processed by AutoCode cannot contain algebraic loops.
- AutoCode models cannot accept data that includes complex numbers.
- The input time vector must start at 0.0.
- AutoCode does not support the following blocks:
 - MathScript Block
 - HDL CoSim Block
 - Implicit UserCode Block
- Macro and Inline Procedure SuperBlocks are not supported within Conditional SuperBlocks.
- When AutoCode encounters a user input Interactive Animation icon, generated code assigns a constant value equal to the initial value of the icon.
- Only block comments are inserted in the generated-for-display Interactive Animation icons.
- AutoCode does not generate parameterized code for the following blocks even though parameters within these blocks can be parameterized using the %variable notation:
 - State Space Block
 - Num, Den Block
 - Gain, Zeros, Poles Block
 - Gain, Damps, Freqs Block



NOTE: Dynamic systems are transformed to optimize performance and the mapping between the Xmath value and the block variable is lost. Variable space is allocated and initialized but the values are hardcoded.

The AutoCode software lets you generate ANSI C or Ada code automatically from SystemBuild models.

You can generate code from the Catalog Browser in SystemBuild or use the **autocode** Xmath command. The generated code represents a complete implementation of the model. The generated code can be targeted for and run on other computers or an actual controller. The default target is a standalone simulation that you can execute on your computer; you can then load the results of the simulation back into Xmath for analysis.

2.2 Generating Non-Customized Code

With Xmath running on your host, generate code for the sample Discrete Cruise System model by taking these steps:

1. Make sure you are in a directory where you want to save your code (and that you have write permission in that directory). If not, enter the following command from the Xmath Commands window:

```
set directory ="your_working_directory"
```

2. From the Xmath Commands window, type the following command to load the model:

```
load "$SYSBLD\demo\cruise_demo\cruise_d.cat";
```



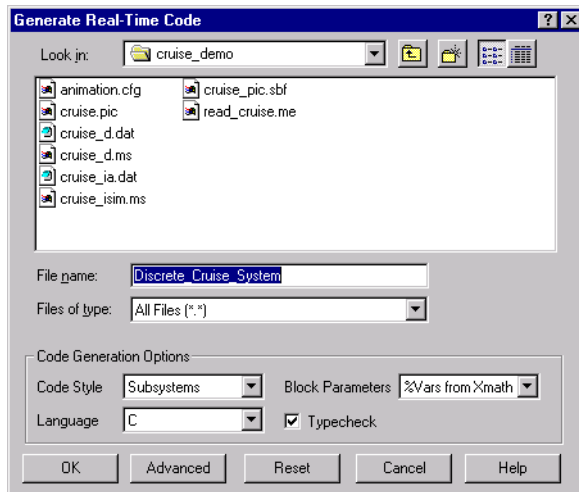
NOTE: The Xmath Commands window is the only place that can recognize environment variables. For loading with other methods, you must know that full pathname of the SystemBuild directory. Xmath commands use *\$env_var*; whereas, commands that go directly to the operating system, such as **oscmd**, use the operating system convention (for example, *%env_var%* for Windows).

3. From the SystemBuild - Catalog Browser, select the Discrete Cruise System SuperBlock.



NOTE: You must generate code from a top level SuperBlock.

4. From the Catalog Browser, select Tools→AutoCode to bring up the Generate Real-Time Code dialog.



or select a top level SuperBlock and enter Ctrl-G to display the dialog.

5. Enter a name in the File name field or accept the default, **Discrete_Cruise_System**.
6. In the Code Generation Options section, select:
 - a. **Subsystems** or **Procedures** in the Code Style field.
 - b. **C**, **Ada**, or **RTF only** as the Language.
 - c. **%Vars from Xmath** or **Block Defaults** in the Block Parameters field.
 - d. Or deselect Typecheck.
7. Click OK to start the code generation process.
8. Display the Xmath Commands window to monitor the progress of the code generation.
9. Once the code generation is complete, look for a statement similar to the following in the Xmath log area:

```
Output generated in d:\user\test\Discrete_Cruise_System.c.
```

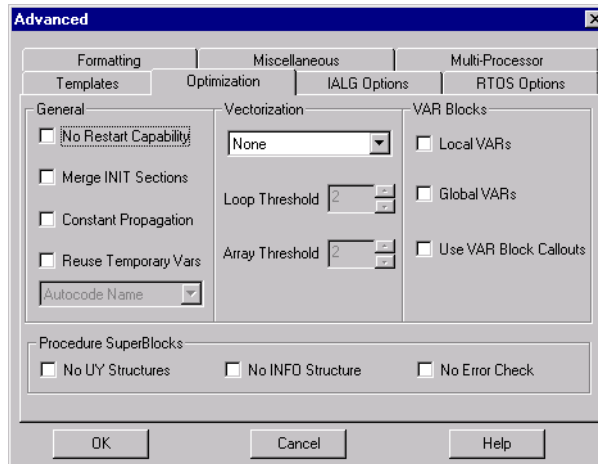
```
Code generation complete.
```

10. (Optional) Display the output file in the Xmath output area by entering a **type** command similar to the following in the Xmath Commands window:

```
oscmd ("type d:\user\test\Discrete_Cruise_System.c")
```

2.3 Generating Customized Code

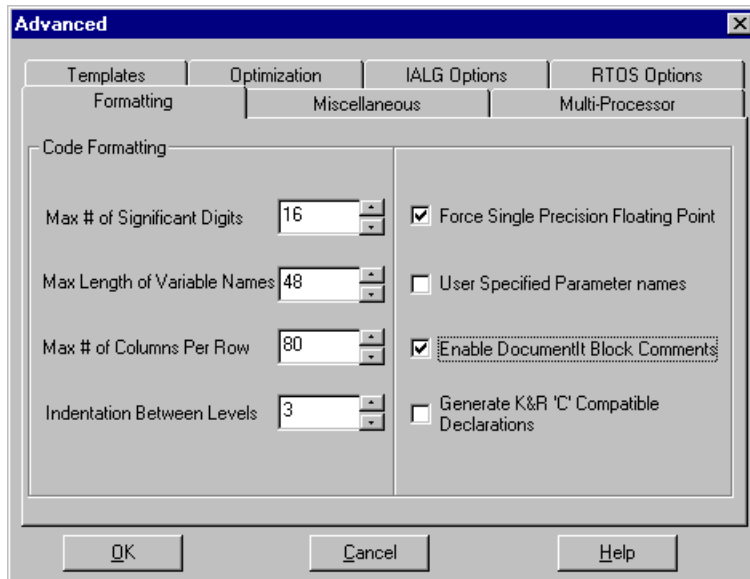
To customize your AutoCode output, click Advanced on the Generate Real-Time Code dialog; this brings up the Advanced dialog.



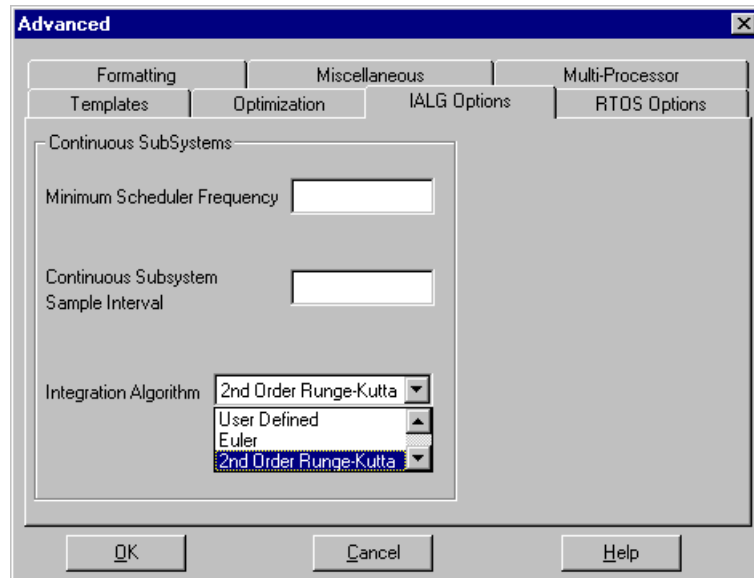
You can use the Advanced dialog or use keywords with the **autocode** Xmath command to customize the generated code as follows:

Templates tab — lets you control the formatting of the output of AutoCode to meet a variety of software needs; you can modify the overall architecture of generated code, customize the scheduler, modify data structures and external I/O calls, add user code, and so forth. Using the Template Programming Language (TPL), you can tailor any part of the code except the hierarchy logic and the elementary blocks. Numerous templates are available, including one to customize the generated code for the pSOSystem real-time operating system. For more information on templates, see the *Template Programming Language User's Guide*.

Formatting tab — lets you set the form of the generated code. You can specify settings for the maximum number of significant digits (default is 16), maximum length of variable names (default is 48), maximum number of columns per row (default is 80), and other formatting settings (as shown).



IALG Options (Integration Algorithms) tab — lets you select an integration algorithm such as Euler, Runge Kutta, Kutta-Merson, or a user-defined algorithm. This tab also lets you set a minimum scheduler frequency and a continuous subsystem sample interval.



Multi-Processor tab — lets you specify a processor map, startup map, background map, interrupt map, skew map, priority map, or map file. You can also specify up to 10 processors and select shared memory callouts.

Optimization tab (shown in 2.3, p.21) — lets you make general, vectorization, and VAR block settings that affect code size and efficiency (see the *AutoCode Reference* for details).

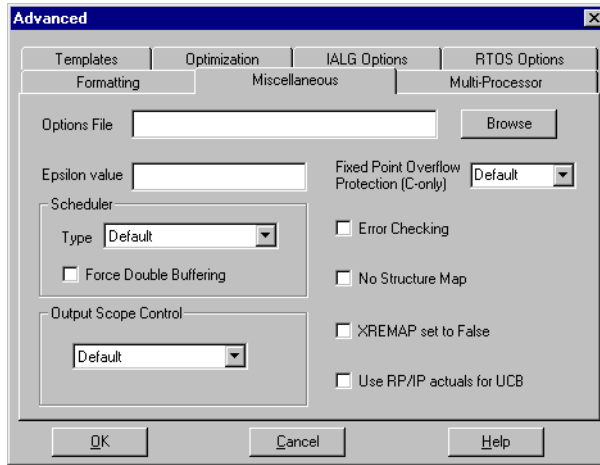
Miscellaneous tab (shown in Figure 2-1) — lets you select an options file, the type of scheduler, output scope control, and various other settings. For information about how to set **epsilon**, for example, see [Appendix A, AutoCode Options](#).

RTOS (real-time operating system) tab — lets you specify a configuration file and whether to generate extended procedure INFO data structures.

Once you have customized your settings, you click OK in the Advanced dialog; then you generate code by clicking OK in the Generate Real-Time Code dialog.

For information about **autocode** keywords, see [Appendix A, AutoCode Options](#) and the MATRIX_X Help.

Figure 2-1 **Advanced Dialog, Miscellaneous Tab**



2.4 Using Templates

The AutoCode software template feature enables you to create accurate software for your application. Unique to AutoCode, the Wind River templates give you control over comment density (adding or deleting comments), application of specific data structures or the initialization of variables. Templates are designed to produce language-specific code for real-time execution in a host environment.

The provided templates can be used for starting points for tailoring generated code to suit specific embedded targets. You can specify the AutoCode template file from the Templates tab of the Advanced dialog (as shown in [Figure 2-2](#)).

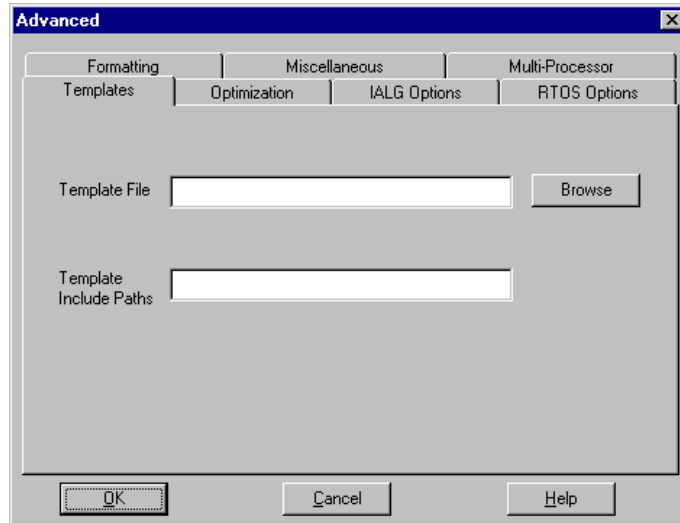


Figure 2-2 Advanced Dialog, Templates Tab

The template can be either a template file (.tpl) or a direct-access file (.dac).

2.5 Applications of AutoCode-Generated Code

In Section 2.1 you learned how to generate real-time code using AutoCode from SystemBuild, the Xmath Commands window, and the operating system prompt. These are applications of the AutoCode-generated code:

- Standalone simulation on the host machine
- Rapid Prototyping
- Real-time simulation
- Embedded real-time control

2.5.1 Standalone Simulation

AutoCode lets you execute code from a standalone UNIX or Windows system; use the same input vectors for SystemBuild simulation; test the generated application with these vectors; and load the output vectors back to Xmath for comparison and analysis.



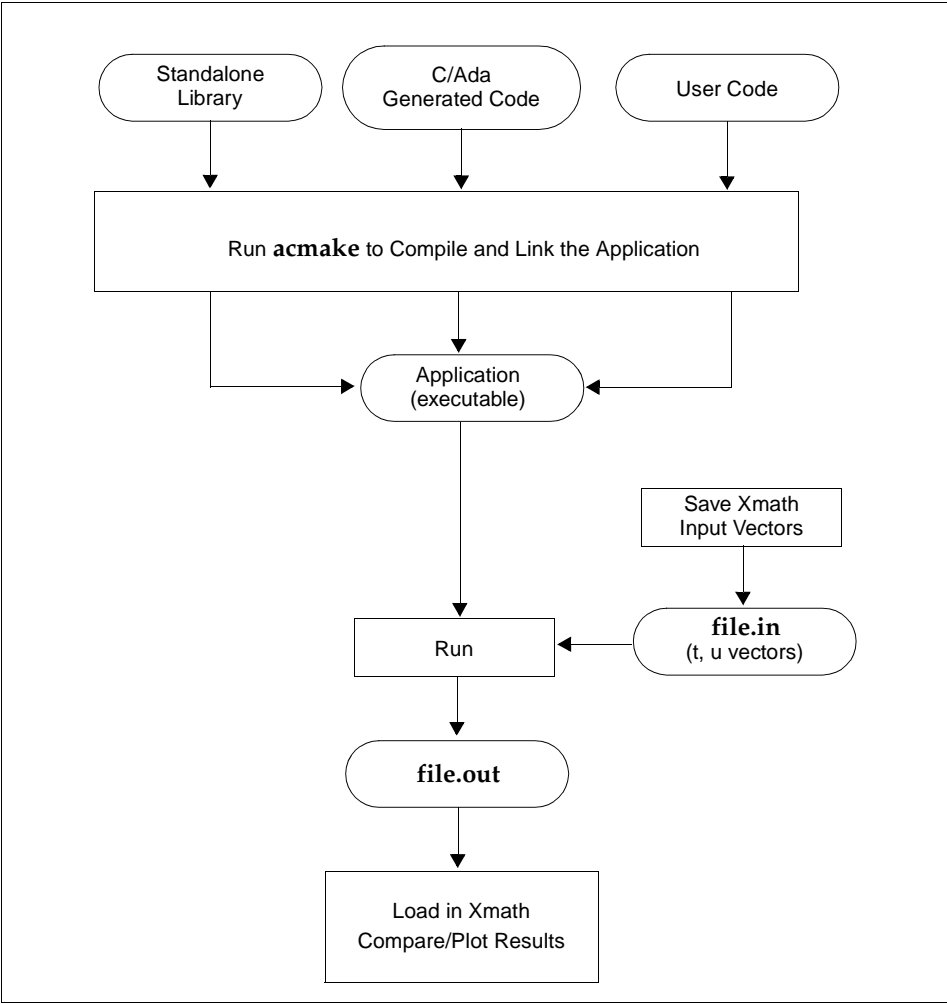
NOTE: Use the `csi` option so generated code will match sim results for continuous systems.

When a system has been modeled in SystemBuild and its source code has been generated using AutoCode, you can compare the outputs from the generated code against those obtained from simulating the block diagram. The Standalone Library, provided in your `src` distribution directory, supports this function.

The Standalone Library is a collection of subroutines that performs the operations of the target-specific utilities to allow testing of the generated code in a traditional non-real-time host or PC environment. The Standalone Library provides services such as reading in an input data file and producing an output file.

The role of the Standalone Library in testing the generated application code against the simulations is illustrated in [Figure 2-3](#), p.27.

Figure 2-3 **Compiling, Linking, and Running the Generated Program**



2

Standalone Simulation

The steps required to test the generated code from an OS prompt are:

1. Depending upon your platform, do either of the following:
 - a. On a Windows system, make sure that the environment variables are set up according to the *Microsoft Visual C++ Getting Started* manual. On a Windows 98 or Windows 95 system, use the **Set envvar =** command in the **autoexec.bat** file. On a Windows NT or Windows 2000 system, select the Environment tab of System Properties in Start→Settings→Control Panel.

The **path** should have: **C:\MSDEV\BIN**.

The **include** should have:

C:\MSDEV\INCLUDE;C:\MSDEV\MFC\INCLUDE.

The **lib** should have: **C:\MSDEV\LIB;C:\MSDEV\MFC\LIB**.

- b. On a UNIX system, make sure the path to your compiler includes your path environment variable. If not, enter the **which cc** command to display your compiler path. If necessary, add the compiler path to you path environment variable by entering:

```
setenv PATH ${path}:compiler_path
```

2. Generate code from the SystemBuild - Catalog Browser.
3. Run the **acmake** command to generate a makefile that automatically rebuilds all out-of-date objects and then relinks the executable. You can use the **acmake** command to complete an incremental build (Ada only) or a full build. For example:

```
acmake update -f model.mak    Incremental build for Ada only (the C makefile  
                               automatically determines whether to do an  
                               incremental or full build)
```

```
acmake -f model.mak          Full build for Windows
```

```
acmake -f model.mk           Full build for UNIX
```

You can use the **clean** option to remove all build objects from a directory:

```
acmake clean -f model.mak    Deletes all build generated files from the  
                               directory
```

4. Create an input file, **file.in**, containing (column) vectors **t** (time) and **u** (inputs), in MATRIX_X ASCII format, using Xmath as follows:

```
t = [0: ...]';  
u = [ ... ];  
save t u file = "file.in" {matrixx, ascii};
```

Time vector **t** should always start with the first element value as zero. **t** and **u** vectors should have the same number of rows.

5. Run your executable from the OS prompt and specify the MATRIX_X ASCII file name as **file.in** and the output file name as **file.out**, as shown below.

```
% executable  
Enter xmath {matrixx,ascii} formatted input filename: file.in  
Enter output filename: file.out
```

6. Load the output from the generated code **file.out** back into Xmath and compare it to the simulation. (See 2.5.2, p.29 for the simulation options. Use the **[te,ye]** extended-time simulation feature to obtain all the discrete time points.) The file contains two items, the output matrix **yrt** and the time vector **ytime**. The generated code and simulation results should match very closely. For more details on each subroutine in the Standalone Library, refer to the comments in the source file.

After you have completed all needed testing and simulation to optimize the functionality and performance of your application, the perfected code can be implemented onto the target processor.

2.5.2 Simulation Options

For best results when using the Standalone Library to compare the generated code with simulation results, you first need to set up the simulator to imitate the generated code's real-time behavior. This is done through simulator options.

The appropriate Xmath command is:

```
setsbdefault, {actiming, extend, typecheck}
```

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.

Three simulation keyword values are forced so that the initialization and posting of outputs match AutoCode.

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. 0 = Outputs of continuous subsystems only are computed based on initial conditions and inputs. Outputs of discrete subsystems are set to $\sqrt{\epsilon}$.
dtout = 0	No extra output time points are specified. This keyword forces the outputs of the simulation to be posted only at the minor cycle of the simulation scheduler, which is defined by the least common multiple of the sampling intervals and timing requirements of the subsystems.

This command can be included in your set-up file. It is a good idea to use these options whenever your discrete controller model is intended for eventual implementation on a digital computer. Analyze your top-level SuperBlock from the SystemBuild menu. For more information, see the MATRIX_X Help or the *SystemBuild User's Guide*.

Assuming that you have executed the **setsbdefaults** statement shown above, you can execute the simulation with the command:

```
[te, ye] = sim(model, t, u, {sim keywords});
```

where:

model is a text string enclosed in double quotes, which is the name of the top-level SuperBlock in the SuperBlock Editor.

t is the required time vector.

u is an input data matrix.

2.5.3 Rapid Prototyping

AutoCode provides the means for fast implementation of SystemBuild block diagrams without lengthy manual coding. You can speed up design iterations by simply editing the block diagrams and generating new code. For more information on rapid prototyping, see the *RealSim User's Guide*.

2.5.4 Real-Time Simulation

AutoCode lets you create and execute code and perform hardware-in-the-loop simulations for an entire system using RealSim. For details regarding real-time simulation, see the *RealSim User's Guide*.

2.5.5 Implement Embedded Real-Time Control

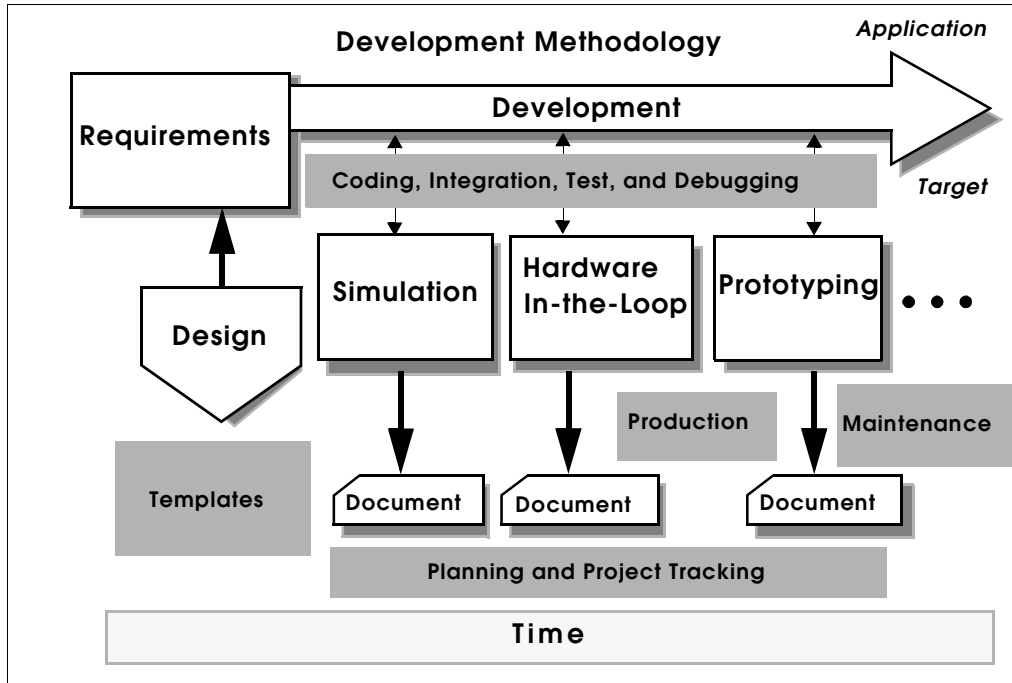
AutoCode lets you generate code for real-time controllers. You can cross-compile, link, and download onto a wide variety of target processors.

2.6 How to Integrate Generated Code into Your Target

This section describes how to integrate automatically generated code into your target-specific application. As shown in [Figure 2-4](#) the development process is

complex and iterative. AutoCode can be used to shorten the coding time and to advance more rapidly into the integration, test, and final debugging stages.

Figure 2-4 Overview of Development Process Using Generated Code

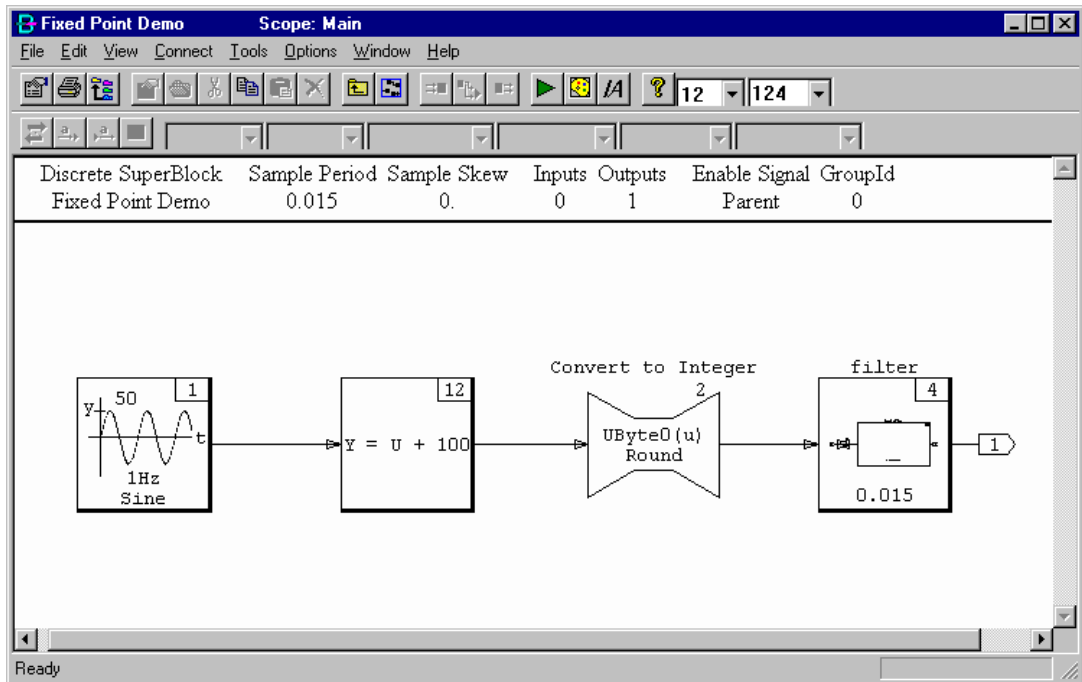


➔ **NOTE:** Development templates can be user-defined for each application or target.

2.6.1 Loading the Fixed-Point Demo

The fixed-point demo (`fixed-point.dat`) found in the `sysbld\demo` directory addresses the following system requirements:

- Design a 5th order filter using fixed-point arithmetic.
- Target the filter for a Motorola 68K board (fixed-point) or simulator
- Minimize code size (limited ROM/RAM)



The fixed-point demo model illustrates how users can analyze and compare fixed versus floating point behavior when all or part of the system is modeled with fixed-point arithmetic. Fixed-point arithmetic is commonly used in high-volume applications to approximate floating-point arithmetic on an integer only microprocessor. SystemBuild provides an extensive set of intrinsic fixed-point data types and analysis tools for fixed-point simulation as described in the *SystemBuild User's Guide*. This implementation allows fixed-point simulation to be easily toggled on and off to facilitate comparison with floating-point behavior. The analysis tools help you identify quantization effects and select fixed-point scale factors.

This model includes a filter implemented using fixed-point arithmetic: all numerical calculations are performed with real numbers represented by an unsigned short (16 bits) data type. The radix is the number of bits used to represent the fractional part of a fixed-point number. By increasing the radix, you increase the resolution, but you also decrease the range of the numbers that can be represented. Adjusting the radix of a fixed-point number to allow the appropriate range and resolution is called scaling. For example, an unsigned byte (8 bits) with

a radix 2 can take on a minimum value of 9, a maximum value of only 31.875, but has a resolution of 0.125.

2.6.2 Determining System Scaling

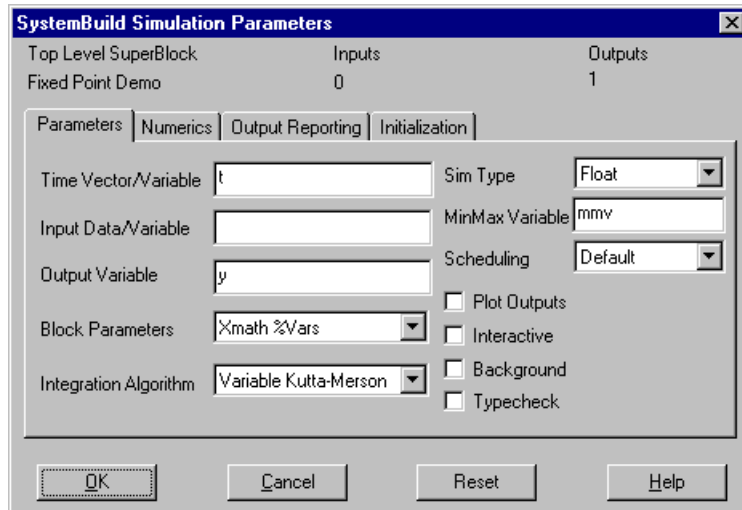
You can use the Minmax Display tool to help you track down fixed-point problems. This tool allows you to examine all output minimum values, maximum values, and the times they were achieved. The tool also tracks any fixed-point overflow that occurs during a simulation. To see the results of a simulation, enter **minmax_display** from the Xmath Commands window and click Load from the Load Dataset dialog. Once you have your minmax display from any simulation, you can select any SuperBlock and any block within a SuperBlock and view DataType, Max Value, Max Time, Min Value, and Min Time.

You can also use the Minmax Display tool to monitor dynamic ranges of signals. For fixed-point signals, overflow and underflow of fixed-point simulations are detected and tracked, along with the time at which the events occurred. You can navigate directly to the block where an overflow occurred. You can then use this information to determine the proper scaling for the system. For example:

1. From the Xmath Commands window, define time variable t as follows:

```
t=[0:0.1:100]';
```

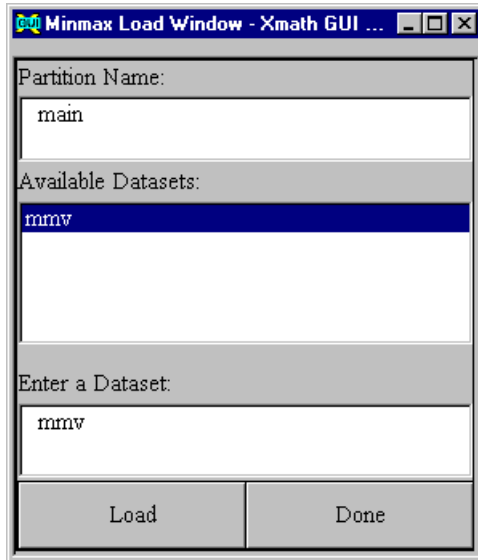
- From the SuperBlock Editor window, select Tools→Simulate and enter a Time Vector/Variable, an Output Variable, and a MinMax Variable in the SystemBuild Simulation Parameters dialog as shown and click OK:



- From the Xmath Commands window, launch the Minmax Display tool with the following command:

```
minmax_display
```

4. To view a MinMax Variable dataset, select Special→Load and the Minmax Load Window appears.



5. Select a dataset (for example, **mmv** for the MinMax Variable) and Click Load. The Minmax Display window appears with analysis data for the fixed-point demo.

The screenshot shows a window titled "Minmax Display - Xmath GUI Tool" with a menu bar (Special, Select, Help). The main area displays the following information:

Dataset: main.mmv

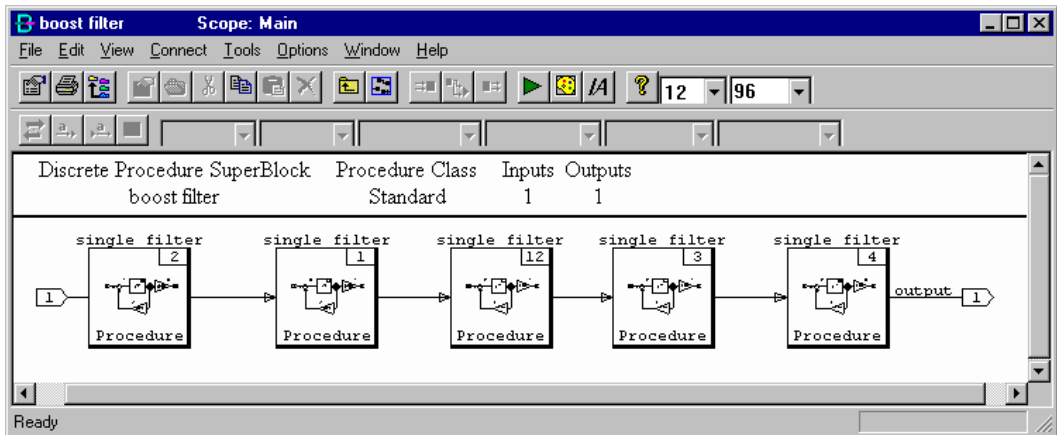
Output Name	Data Type	Max Value	Max Time	Min Value	Min Time
output	UShort7	147.623	33.405	0	0

SuperBlocks	Blocks	Overflow	Underflow	Protection
1 Fixed Point Demo	filter_4			
2 filter_4	block.1			
3 boost filter_4	block.12			
4 single filter_2	Convert to Integer.2			
5 single filter_1				
6 single filter_12				
7 single filter_3				
8 single filter_4				

2.6.3 Modular Programming

The fixed-point demo consists of a sine wave signal generator with a magnitude between -50 and 50, an algebraic expression that offsets the sine wave by 100 so that its magnitude is between 50 and 150, a signal converter to change from type float to type unsigned byte, and a filter. The filter is implemented with fixed-point arithmetic. Notice that the "single filter" SuperBlock is implemented as a Procedure Class of Standard.

Demonstrating the re-use of blocks, the "single filter" SuperBlock is built once, but reused 5 times in the "boost filter" SuperBlock.



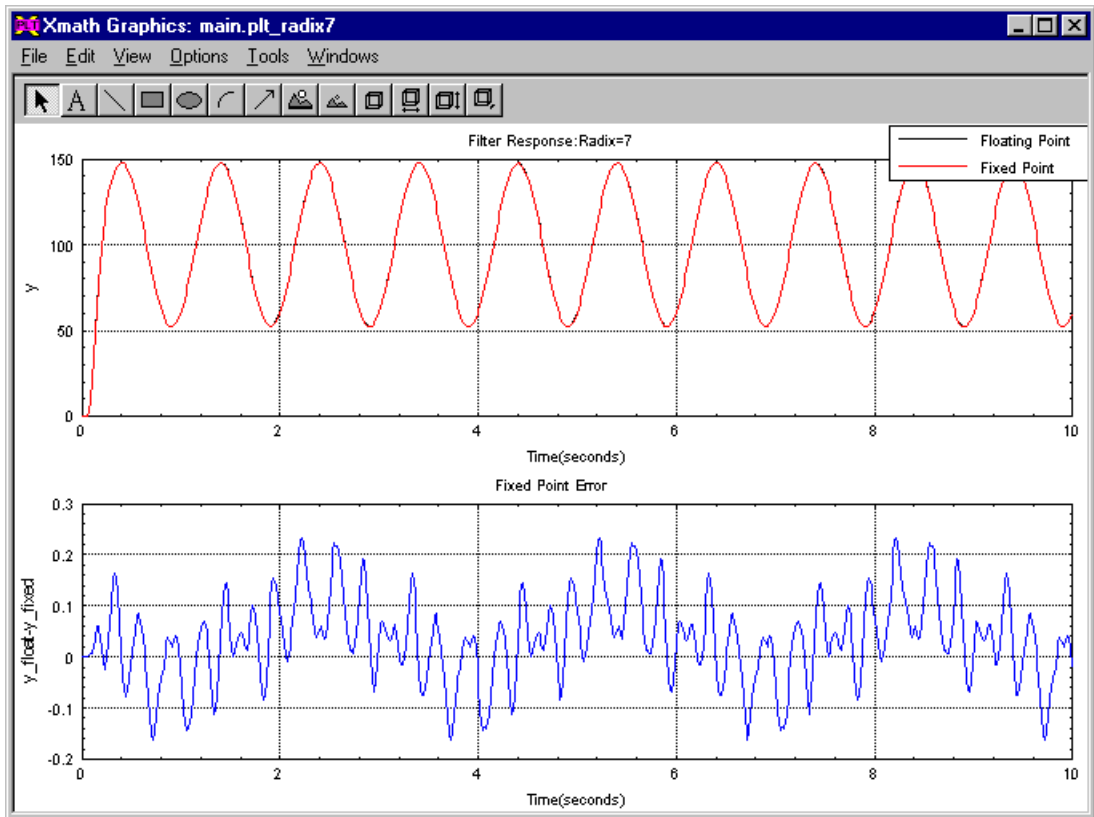
When the same SuperBlock is implemented over-and-over within a model, one should consider using a Procedure SuperBlock. In generated code, a procedure is comparable to a software function; that is, it is called by another program and data is passed via arguments. Thus, the application's overall code size can be reduced. The use of procedures assists in meeting any requirements for code size minimization.

In [Example 2-1](#), all elements of the "single filter" SuperBlock use a user-defined type called `filrad`. `filrad` is a fixed-point data type: an unsigned short with radix 7 (US7). Simulating the model twice, first using floating-point arithmetic and then using fixed-point arithmetic, the results shown in [Figure 2-5](#) are observed.

Example 2-1 Floating-Point Versus Fixed-Point Using Radix 7

```
t=[0:0:.015:10]';
modifyUserType "filrad",{radix=7}
yflt=sim("FixedPointDemo",t);
yfix=sim("FixedPointDemo",t,{fixpt=1});
plot(t,[yflt,yfix],{xlab="Time(seconds)",ylab="y",title="Filter
Response:Radix=7",rows=2,
row=1,legend=["Floating Point","Fixed Point"]});
plt_radix7=plot(t,[yflt-yfix],
{xlab="Time(seconds)",ylab="y_float-y_fixed",title="Fixed Point
Error",line_color="blue",rows=2,row=2,keep})?
```

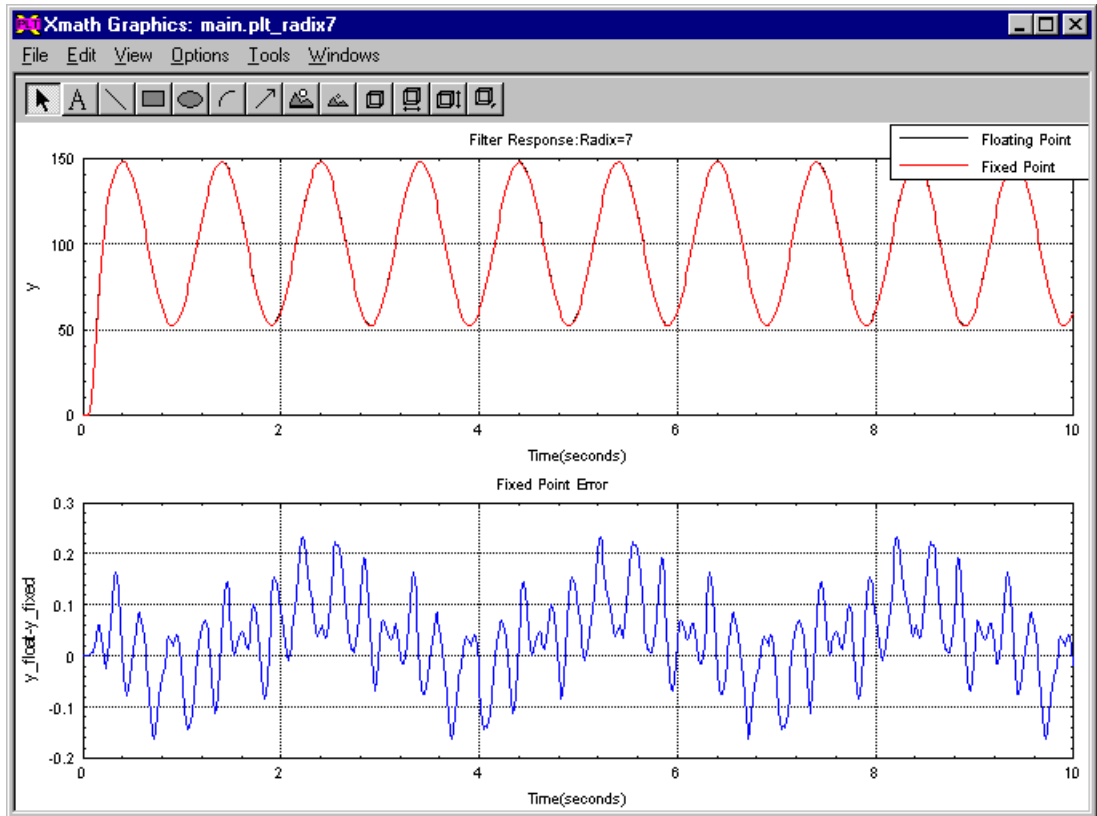
Figure 2-5 Floating-Point Versus Fixed-Point Using Radix 7



Example 2-2 Floating-Point Versus Fixed-Point Using Radix 8

```
t=[0:0:.015:10]';
modifyUserType "filrad",{radix=8}
yflt=sim("FixedPointDemo",t);
yfix=sim("FixedPointDemo",t,{fixpt=1});
plot(t,[yflt,yfix],{xlab="Time(seconds)",ylab="y",title="Filter
Response:Radix=8",rows=2,
row=1,legend=["Floating Point","Fixed Point"]})?
plt_radix8=plot(t,[yflt-yfix],
{xlab="Time(seconds)",ylab="y_float-y_fixed",title="Fixed Point
Error",line_color="blue",rows=2,row=2,keep})?
```

Figure 2-6 Floating-Point Versus Fixed-Point Using Radix 8



2.6.4 Comparing the Output

At first glance, the floating-point and fixed-point responses look very similar. The floating-point response is obscured by the fixed-point response because the responses are so close. However, the plot of error shows that there is a small difference. When we change the radix of `filrad` to 8 (US8), this improves the resolution (0.00390625) of the fixed-point arithmetic but reduces the range (0 to 255.9960938) of numbers that can be represented. Again, the model is simulated twice, once with floating-point arithmetic and once with fixed-point arithmetic.

Notice that there is now a very obvious saturation effect in the filter. There is a significant error between the floating-point and fixed-point responses caused by adjusting the radix of the fixed-point representation.

2.6.5 Implementation

Typically the phase of development where code is generated (or written) is called the Implementation or Software Development phase. Before you generate code, you must set up target-specific utilities and templates for a particular application. The templates provide a model-independent way to tailor the generated code to a specific target. Any changes to a model translate directly to changes in the generated code but do not affect the template.

For example, we can use the `c_sim.tpl` template (excerpt shown in [Example 2-3](#)) from the `case\Acc\templates` folder to generate C code targeted for the Motorola 68K board. The utilities file provides a means to accommodate input and output, which are specific to the target. The `sa_utils.c` file (excerpt shown in [Example 2-4](#)) from the `case\Acc\src` folder provides a means to accommodate input and output, which are specific to the target.

Example 2-3 `c_sim.tpl` Template File (Excerpt)

```
--
-- File      : c_sim.tpl
-- Project   : (C); AutoCode/C
-- Edit level :
--
-- Abstract:

This template is designed to produce both C executive code for
real-time execution in a host environment or modular procedures-only
code. This template can be used as a starting point for tailoring the
generated code to suit specific embedded targets. In particular,
changes would be required in the application scheduler code to
transform the scheduler into an interrupt task driven by a real-time
clock. Also supported is the ability to generate the code into either
a single file or multiple files.

*@

@INT i,j@  @/ Global variables, mainly used for loop counters.

@INC "c_core.tpl"@@
@INC "c_intgr.tpl"@@
@INC "c_sched.tpl"@@
@INC "c_async.tpl"@@
@INC "c_echart.tpl"@@
@INC "c_api.tpl"@@

@*

This main segment defines the control flow of code generation. It invokes
library tpl functions and other tpl functions defined below. User can
customize any of the segments defined in this template or implement new
segments (tpl functions) thereby controlling the output of the code
```

generator.

```
*@  
  
@SEGMENT MAIN()@@  
@ASSERT STRCMP("C",language_s)@@  
@ASSERT not multiprocessor_b@@  
@IFF procs_only_b and nprocedures_i eq 0@@  
@ FILEOPEN("stdout", "append")@@  
  ** MODEL ERROR  
  **  
  ** Cannot use 'procedures-only' option with a model  
  ** that does not have any STANDARD PROCEDURES  
  **  
@ FILECLOSE@
```

Example 2-4 sa_utils.c File (Excerpt)

```
** File      : sa_utils.c  
** Project   : Autocode/C  
**  
** Abstract:  
**  
**      Definitions for AutoCode files.  
**  
** The following routines provide a set of utility procedures  
** that can be used with the code generated by AutoCode/C.  
** These routines allow testing of the generated code in a host  
** environment and provide a link with Xmath via MATRIXx ASCII formatted  
** output files.  
*/  
  
#if (__STDC__ || defined(__cplusplus) || defined(c_plusplus))  
#include <stdlib.h>  
#include <string.h>  
#else  
extern int strcmp();  
extern void exit();  
  
#endif  
#include <stdio.h>  
#include <math.h>  
#include "sa_sys.h"  
#include "sa_defn.h"  
#include "sa_types.h"  
#include "sa_math.h"  
  
#if (ANSI_PROTOTYPES)  
static void fatalerr(RT_INTEGER errorCode);  
#else
```

```
static void fatalerr();
#endif

/* Storage allocation parameters.

 * To change storage size limits, modify the following parameters.
 * -----
 * MAXU      = max number of storage elements for input values in U
 * MAXUTIM   = max number of input time points in UTIME
 * MAXY      = max number of storage elements for output values in Y
 * MAXYTIM   = max number of output time points in YTIME          */

#define MAXU      100000L
#define MAXUTIM   10000L

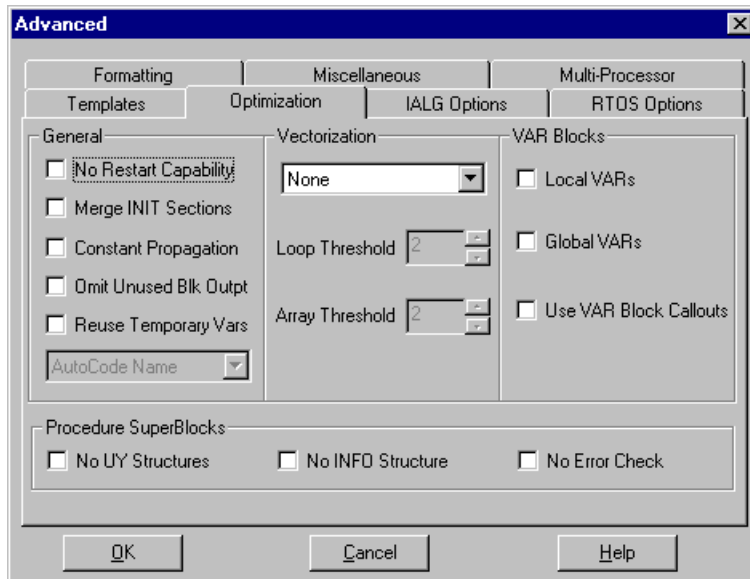
#if defined(OSF1)
#define MAXY      120000L
#define MAXYTIM   30000L
#else
#define MAXY      80000L
#define MAXYTIM   20000L
#endif

#define REQVER    700
#define REQVAR    2
#define DOUBLE_PRECISION TRUE
```

2.6.6 Optimizations

AutoCode offers many code optimization options as indicated on the Optimization tab from the Advanced dialog. These options are available in the form of “flags” that can be turned on and off. These flags can be set to achieve compiler-like optimizations, reduction of stack space, and/or object size, and improvement of runtime performance.

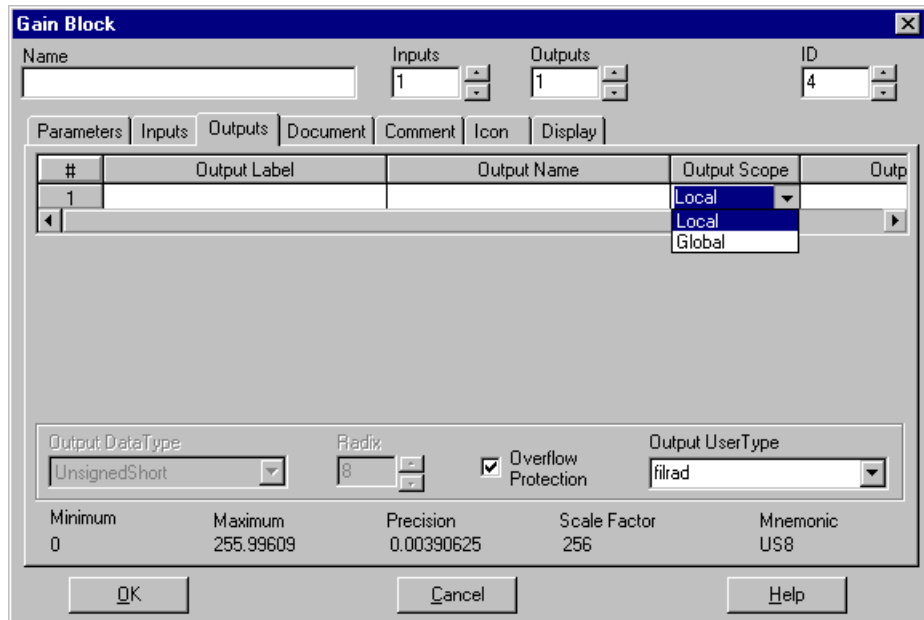
For many users, accepting the default settings is the best approach. For experienced MATRIX_X users, a recommended approach is to experiment with existing models and regenerate code with different options selected. Once a model has been tested and meets performance expectations, optimization flags can be turned on and efficient production quality code can be generated. You can evaluate your optimization results against your particular application.



Beyond the target-specific utilities and templates lies consideration for interfacing with an existing scheduler. AutoCode provides its own scheduler that can be included in with the automatic generation of code, but it may become necessary to interface with an existing scheduler. With modifications to the template, the crystal frequency of the processor can be used as the timing base. The interrupt service routine (ISR) provides entry to the scheduler and incremental time ticks to the timer.

2.6.7 Integration and Test

An important part of the development process is to test iteratively. Once you have built part of your model, you can test the partially built model to observe performance or whether the model meets a particular requirement. One way to do this is to set up test points. You can establish a test point for any given block by setting a signal's scope to global. This is done from the Output tab of the block dialog.



Whether you select local or global scoping has no effect on simulation, but AutoCode sets up a software scope for the signal. During testing, you can easily access a global signal. Planning ahead for this test points can improve your integration and test procedures considerably.

2.7 How to Write Production Quality Code (Graphically)

This section describes the mechanics and the benefits of writing software graphically and then automatically generating code. The MATRIX_X toolset has been designed for graphical software production. As illustrated in [Figure 2-4](#), p. 32, the software development process consists of requirements design, coding, and integration and test. The traditional method of hand-coding and hand-correcting the code is not considered to be the most effective method in the days of “fast time to market.”

2.7.1 Graphical Solutions

If requirements can be displayed graphically, then they can be used by system design and software design teams. Software engineers can then focus more on architecture on verification. In order to program graphically, certain design abstractions need to be understood and used. Basic or primitive blocks can be used to build more complex designs. Interconnections can be represented as connections between blocks. Data can be assigned to variables.

Design Abstractions

The SystemBuild and AutoCode tools make use of the following design abstractions:

SuperBlock — This block is user-defined and can be periodic or a procedure. A Periodic SuperBlock lets you define the exact rate at which the block must run (either continuous or at a discrete rate). A Procedure SuperBlock takes on the timing attributes of its calling block (the block's parent in the hierarchy).

Blocks — Or “building blocks” are the basis for defining algorithms. The SystemBuild environment has several types of blocks:

Predefined Blocks — Available from the Palette Browser provide a wide range of functionality: algebraic, dynamic, logical, interpolation tables, trigonometric, piece-wise linear, transformation, and signal generators

BlockScript Blocks — Provide general programming functionality via a scripting language. The BlockScript block must be interpreted, so usage does add some overhead to the model design.

UserCode Blocks — Link either C or FORTRAN subroutines to the SystemBuild environment.

BetterStateChart Blocks — Provide a link to BetterState statecharts. With the BetterState module, statecharts can be used within SystemBuild models. You can simulate your model with Sim, ISIM, and RealSim and generate code with AutoCode capabilities for BetterState statecharts. Statechart models can be used to generate C or Ada code, and interactive simulation enables the animation of BetterState diagrams.

Custom Blocks — Provide users with the ability to customize any block and put it on the Palette Browser. Customization can be as simple as changing an icon or changing the associated Help, or it can be as complex as the design of a new block with Help, and icon and callbacks (from the predefined block, a UCB, or BlockScript).

Signals — Typically signify data flow and can be represented visually through interblock connections, or through variable reads and writes.

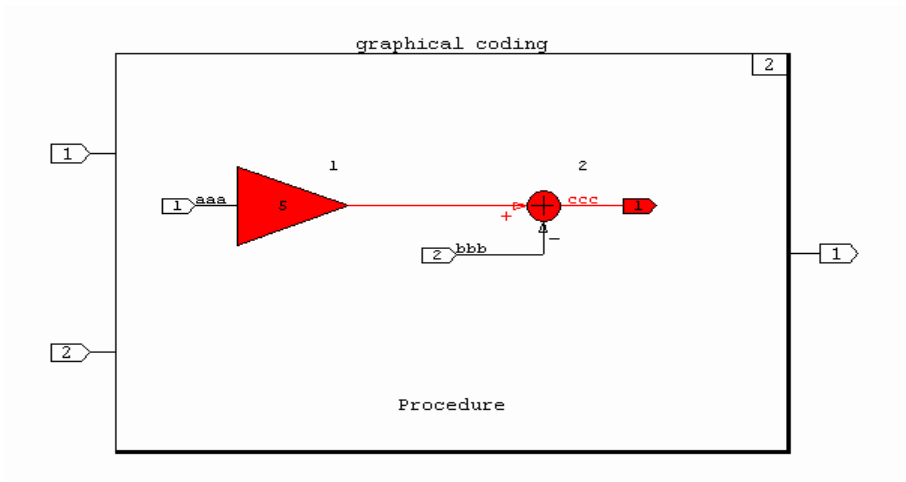
Code Abstractions

SystemBuild maps its design abstractions into code abstractions supported by AutoCode and BetterState. This includes the following code abstractions:

Functions and Packages — C functions and Ada packages are mapped into SuperBlocks. Software modules typically make use of functions and packages to represent algorithms or routines that are used multiple times and usually become part of a library. A SuperBlock can be used in the same way. It has lower level functions (blocks), has parameters, variables, and inputs and outputs.

Code Segments — Blocks and local or global scoped variables (signals). Software code segments are graphically represented as SystemBuild blocks. [Figure 2-7](#) and [Example 2-5](#) show a comparison of a simple procedure (graphical coding) and the corresponding generated code. This example implements the block representation of a gain and a summer block.

Figure 2-7 Floating-Point Versus Fixed-Point Using Radix 8



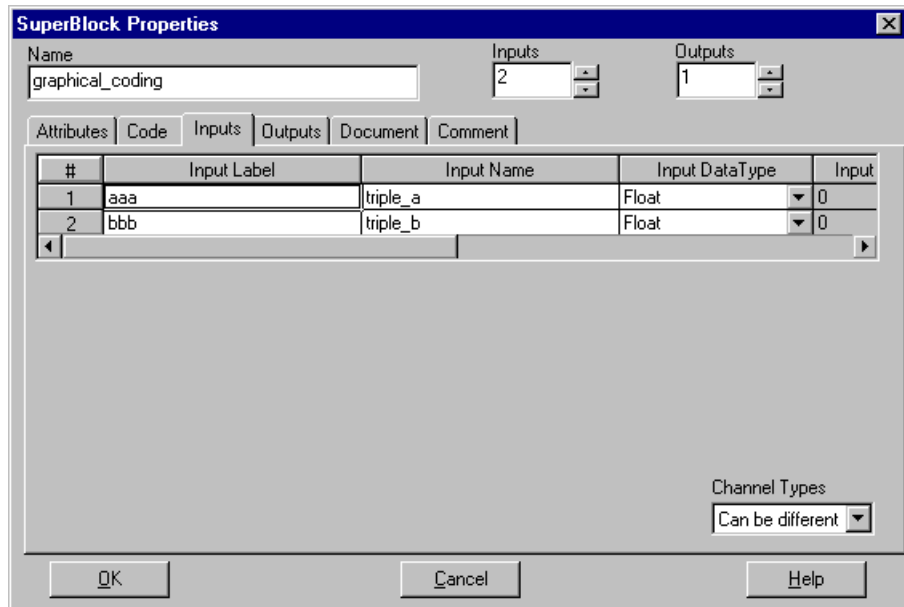
2.7.2 Labels and Names

Label names should be given to all but the most trivial of signals. Labels significantly improve code readability and maintainability.

SuperBlock inputs have two modes of assigning input names: specifying labels locally or inheriting higher level labels. Entering names for local inputs is always a good idea, even if you are doing top down design. Entering local names helps when the SuperBlock is reused in other applications or parts of the model, as these names appear in the connections editor, making it easier to make the correct connections.

In AutoCode, labels are the mechanism for controlling vectorization in code, so good labeling is very important.

A label appears within the diagram, while a name does not. As shown below the Input Labels of **aaa** and **bbb** defined in the SuperBlock Properties dialog appear in the diagram shown in [Figure 2-7](#) and the input names of **triple_a** and **triple_b** appear in [Example 2-5](#). The Input Name field is used only by AutoCode. For basic blocks, you have a choice of specifying a label and a name. AutoCode uses the name of a signal, if it exists, before using its label to create the variable used in the generated code.



A common use for a name is when the variable in the generated code must be different than (or longer than) the label. Never use a name without a corresponding label (the name will not appear in the diagram). For most designs, there is no need to use a name.

2.7.3 Modular Programming Through Procedures

SuperBlocks provide a means for a variety of modular programming styles. The Procedure SuperBlock represents a generated software procedure, which is called as a standalone function. As in hand coding, the contents of a procedure represent a function. That function may contain calls to other functions, resulting in a hierarchy of procedures, each calling procedures below it. Similarly, the SystemBuild environment uses a hierarchy of SuperBlocks. Six types of Procedure SuperBlocks can be used to obtain the desired code results: Standard, Inline, Macro, Background, Interrupt, and Startup. For simulation and code generation, SuperBlocks are grouped into subsystems, by rate or by SuperBlock type (for example, Procedure or triggered). For example, all msec SuperBlocks comprise one subsystem, all Standard procedures comprise another system, and so forth.

Standard Procedures — Are reusable and re-entrant. They can be nested in any Discrete, Triggered, or Procedure SuperBlock.

Inline Procedures — Generate inlined code which provides modular design and optimization at code generation time. Unlike other classes of Procedure SuperBlocks, an Inline Procedure is treated as part of the parent subsystem, not an individual subsystem. The primitive blocks nested within an Inline Procedure are merged into the subsystem of the parent SuperBlock. As a result, use of Inline Procedures can result in a different block execution order and can help eliminate potential algebraic loops. This also reduces AutoCode generated code size and procedure call overhead for nested Procedure SuperBlocks.

Macro Procedures — Allow the programmer to substitute a user-supplied macro statement in place of a call to a generated procedure. This allows you to directly call a special I/O or utility function from the generated code, but execute equivalent behavior modeled with SystemBuild blocks at Simulation time.

Background Procedures — Contain tasks that are performed when the system is otherwise idle.

Interrupt Procedures — Contain computations performed within an asynchronous Interrupt Service Routine (ISR) in a real-time environment.

Startup Procedures — Contain initialization calculations and assignments performed prior to the start of simulation or just after system hardware initialization in a real-time environment.

Example 2-5 **Sample Generated Code from a Discrete SuperBlock Procedure**

```
/****** Procedure: graphical_coding *****/
void graphical_coding( struct _graphical_coding_u *U
,struct _graphical_coding_y *Y
,struct _graphical_coding_info *I
)
{
    RT_INTEGER *iinfo = &I->iinfo[0];
    /***** Local Block Outputs. *****/

    RT_FLOAT graphical_coding_1_1;

    /***** Output Update. *****/
    /* ----- Gain Block */
    /* {graphical_coding..1} */
    graphical_coding_1_1 = 5.0*U->triple_a;
    /* ----- Summer */
    /* {graphical_coding..2} */
    Y->ccc = graphical_coding_1_1 - U->triple_b;
```

```
        iinfo[1] = 0;  
EXEC_ERROR: return;  
}
```

Note that variables, as in software, can be scoped locally to the procedure, or globally. A local scope signal is data represented as a stack variable, while a global scope signal is data represented by a global variable, with assigned memory address, if needed. In this example, the inputs **aaa** and **bbb** are locally scoped, while the output **ccc** is scoped as a global. In the code, since **ccc** is global, it is not required as an argument of the procedure.

3

Using AutoCode with BetterState

This chapter explains how to use AutoCode with SystemBuild models that have BetterState charts. For more information about BetterState, see the “BetterStateChart Block” chapter in the *SystemBuild User’s Guide*, the *BetterState User’s Guide*, and the “BetterStateChart Block” topic in MATRIX_x Help.

3.1 Procedural and Event-Driven BetterState Charts

AutoCode provides support for procedural BetterState charts and event-driven BetterState charts, but certain limitations and distinctions have to be considered:

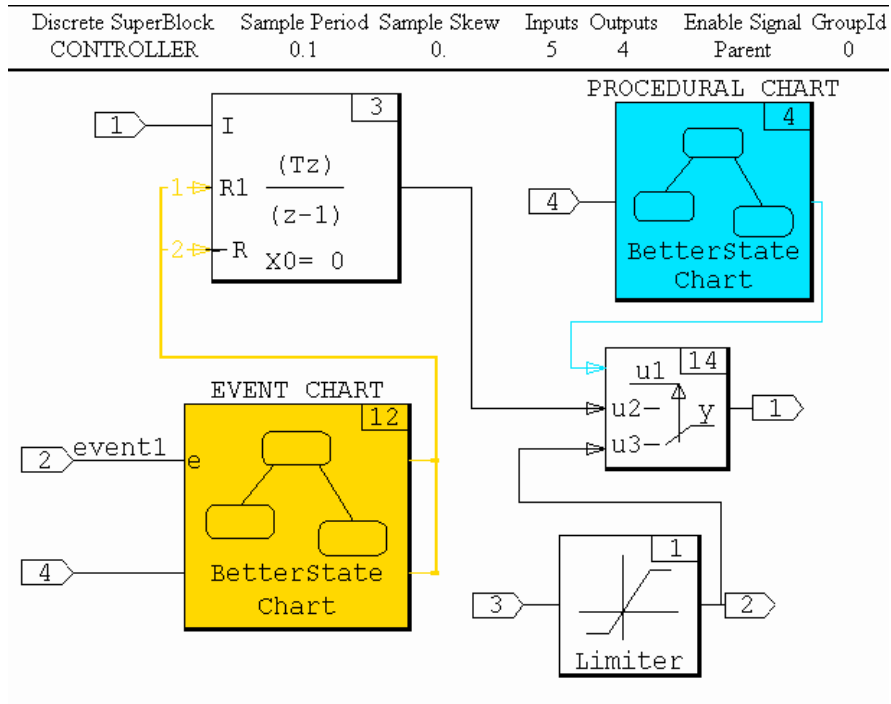
- A procedural BetterState chart is generated as a callout within the C or Ada procedure implementing the containing subsystem. The procedural BetterState chart can be in any subsystem type other than continuous.
- For event-driven BetterState charts, each chart instance is generated as a unique subsystem, implemented by a C or Ada procedure. An event source *must* be a system external input. Wrappers are generated on a per-pin basis for each external input pin driving an event-driven chart’s trigger input.



NOTE: Event-driven BetterState charts can be in any SuperBlock type.

Figure 3-1 illustrates an example system with both an event-driven chart and a procedural chart.

Figure 3-1 Model with BetterState Event-Driven and Procedural Charts



3.2 Generating Code for a BetterStateChart Block

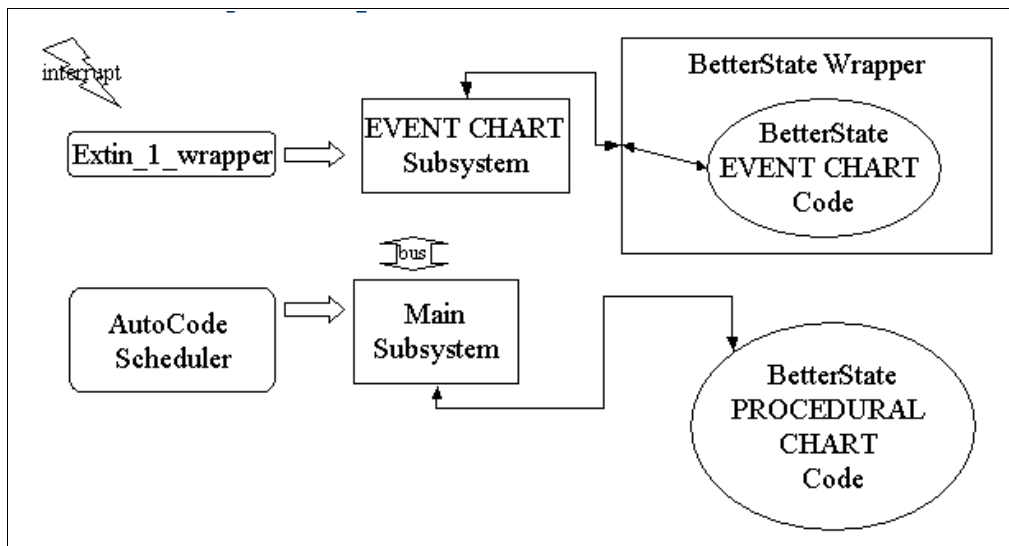
Generating code for the SystemBuild model CONTROLLER, as shown in Figure 3-1, gives us the following:

- The main subsystem containing block code for the integrator, signal path, limiter, and PROCEDURAL CHART (a callout).
- The event-driven chart (EVENT CHART) subsystem, which is primarily a callout to functions inside a BetterState wrapper file.

- A hook paired with the external input driving the event-driven chart. (Invoking it calls the dependent event-driven chart.)
- BetterState code conforming to AutoCode style implementing the charts (of both types).
- The BetterState wrapper which lies, figuratively, between AutoCode and the BetterState code for the event chart.

Figure 3-2 shows this process graphically.

Figure 3-2 Code Generation for BetterState Event-Driven and Procedural Charts



Example 3-1 shows the block code for PROCEDURAL CHART in the Main subsystem. Example 3-2 shows excerpts from the actual generated code for the EVENT CHART (event-driven chart) subsystem.

Example 3-1 Block Code for PROCEDURAL CHART in the Main Subsystem

```

/* BetterState Chart Block */
/* {PROCEDURAL_CHART.4} */
if( INIT ) {
    CHRT_PROCEDURAL_CHART_BSInit(&PROCEDURAL_CHART_4_bs, U->event1_1,
                                &PROCEDURAL_CHART_1);
}
CHRT_PROCEDURAL_CHART(&PROCEDURAL_CHART_4_bs, U->event1_1,
                      &PROCEDURAL_CHART_1);
    
```

Example 3-2 **Generated Code for EVENT CHART Subsystem**

```

/***** Subsystem 1 *****/
void subsys_1(struct _Subsys_1_in *U, struct _Subsys_1_out *Y,
             RT_INTEGER EVENT_ID) {
    if (SUBSYS_PREINIT[1]) {
        /* ----- BetterStateChart Block */
        /* {EVENT_CHART.12} */
        CHRT_EVENT_CHART_init_wrp(&EVENT_CHART_12_bs, TIME, U->event1_1,
                                &Y->chart_out1, &Y->chart_out2);

        return;
    }
    /***** Event Update *****/
    if( EVENT_ID ) {
        switch( EVENT_ID ) {
            case 2 :
                /* ----- BetterStateChart Block */
                /* {EVENT_CHART.12} */
                CHRT_EVENT_CHART_event_wrp(&EVENT_CHART_12_bs, EVENT_ID-1, TIME,
                                          U->event1_1, &Y->chart_out1,
                                          &Y->chart_out2);

                break;
        }
    }
}

```

The generated code in this example creates two wrappers: one for the external input monitor (shown in [Example 3-3](#)) and one for the EVENT CHART subsystem (shown in [Example 3-4](#)).

Example 3-3 **Wrapper Code for External Input Monitor for CONTROLLER**

```

/** Wrapper for External Input Monit 1 `CONTROLLER_1` */
void wrap_extin_monit_1() {
    static RT_INTEGER reentering=FALSE;
    if( reentering ) { Signal_An_Error(1); return; }
    reentering = TRUE;
    wrap_echart_subsys_1(2);
    reentering = FALSE;
}

```

Example 3-4 **Block Code for EVENT CHART Subsystem**

```

/** wrapper for EVENT CHART subsys_1 */
void wrap_echart_subsys_1( RT_INTEGER EVENT_ID ) {
    RT_INTEGER status;
    if( TASK_STATE[1] != BLOCKED ) { Signal_An_Error(1); return; }
    TASK_STATE[1] = RUNNING;
    Update_DS_With_Externals();
    subsys_1_in.event1 = sys_extin.event1;
    subsys_1_in.event1_1 = sys_extin.event1_2;
    SUBSYS_TIME[1] = ELAPSED_TIME;
    subsys_1(&subsys_1_in, ssl_outw, EVENT_ID);
    if( ERROR_FLAG[1] == OK ) {

```

```

        Update_Outputs(1); TASK_STATE[1]=BLOCKED;
    }
    System_Extout_Copy();
    if ( (status = External_Output ()) != OK ) { Signal_An_Error(1); return;}
}

```

3.2.1 Handling BetterState Charts That Call Procedures

If your BetterState chart (event-driven or procedural) calls one or more procedures, then the BetterState code must call AutoCode procedures. This is handled by generating a special wrapper known as the BSAPI or BESTAPI around each procedure called by a BetterState chart.

The API is scalar and of an agreed-upon form so that BetterState code can generate the proper calls. The BSAPI layer is generated automatically by AutoCode for each procedure marked by the Analyzer as needed by BetterState.

The BSAPI wrapper is similar but not identical to the SDK wrapper created in a previous release. For information about the SDK, see [Appendix B](#).

3.2.2 Handling BetterState Charts That Read or Write Variable Blocks

If your BetterState chart (event-driven or procedural) reads or writes variable blocks, AutoCode uses API calls to identify the given WriteVariable or ReadVariable block. These API calls account for potential name mangling issues, since BetterState can only use the unmangled name. For additional information about variable blocks, see the *SystemBuild User's Guide* or the MATRIX_x Help.

This process leads to the creation of several interrelated files. For example, consider a model with two procedural charts and two AutoCode procedures (called from the chart code). Generating code for Ada produces the following files:

<code>CHRT_best_chart1.a</code>	Chart code
<code>CHRT_best_chart2.a</code>	Chart code
<code>proc1.a</code>	Procedure specification
<code>proc1.a</code>	Procedure body
<code>proc1_bsapi.a</code>	BSAPI for procedure
<code>proc2.a</code>	Procedure specification
<code>proc2.a</code>	Procedure body

<code>proc2_bsapi_.a</code>	BSAPI for procedure
<code>model_.a</code>	Model for global specification
<code>model.a</code>	Model for global body
<code>model_ss1.a</code>	Subsystem 1 body

You can then use the **acmake** command to generate a makefile that automatically rebuilds all out-of-date objects and then relinks the executable. You can use the **acmake** command to complete an incremental build (Ada only) or a full build. For example:

<code>acmake update -f model.mak</code>	Incremental build for Ada only (the C makefile automatically determines whether to do an incremental or full build)
<code>acmake -f model.mak</code>	Full build for Windows
<code>acmake -f model.mk</code>	Full build for UNIX

You can use the **clean** option to remove all build objects from a directory:

<code>acmake clean -f model.mak</code>	Deletes all build generated files from the directory
--	--

3.3 Using BlockScript User Code

The BetterStateChart block is a reference to a BetterState chart that you create in BetterState. In a BetterState chart, you choose a language for the user code in a chart (for example, BlockScript). The inputs and outputs to the block are input and output chart arguments that you must define in the Data Dictionary. For additional information about the BetterStateChart block, see the MATRIX_x Help.

You provide code for state actions, transition conditions, and transition actions, or you can define an action as an invocation of a SystemBuild procedure SuperBlock. For this user code, you can use chart arguments or variables. You also must define all variables that you use in conditions and actions in BlockScript in the Data Dictionary. For additional information about the Data Dictionary, see the *BetterState User's Guide*.

4

Managing and Scheduling Applications

This chapter details the management of the application control flow via the real-time scheduler. Topics include scheduler operation sequence, subsystem properties, subsystem interruption, and examples of scheduler operation.

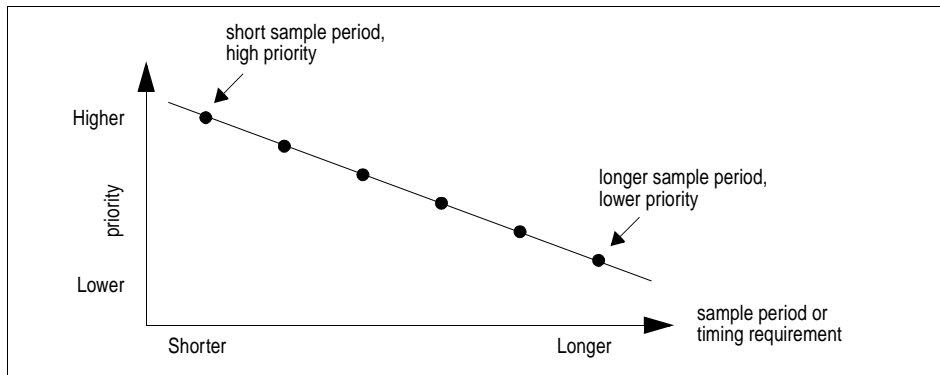
4.1 Real-Time Application Scheduler

AutoCode builds the scheduler as part of the real-time application program by means of the template file. The scheduler performs overall direction and control of inserting inputs, scheduling tasks, posting outputs, and dispatching the tasks that perform the work of the real-time system. Although you can tailor the scheduler as well as other parts of the code, the intention of this program is to provide a generic real-time scheduler, combining high performance with deterministic, prioritized, pre-emptive scheduling of application tasks that have different timing requirements.

The application scheduler operates on the principle of rate-monotonic scheduling, deriving priorities for the tasks from the repetition rate for periodic subsystems and the timing requirement for triggered subsystems. The algorithm assigns higher priority to the faster sample rate or timing requirement subsystems and lower priority to slower ones (see [Figure 4-1](#), p.60). The rate-monotonic algorithm maximizes the number of tasks that get to complete their operations in a given time. Using the rate-monotonic algorithm, all periodic tasks complete their operations if CPU task utilization does not exceed about 70%.

For consistent and deterministic operation in a real-time environment, the task subroutines are scheduled and dispatched as encapsulated objects, which accept inputs and post outputs strictly under control of the scheduler. For this reason, all external input and output operations are handled by the scheduler directly and inter-task data transfer is performed via input sample and hold. The scheduler is re-entrant except for the critical section, which must not be interrupted. The scheduler can be called externally by means of an interrupt handler (for real-time applications) or by a background task (for simulation).

Figure 4-1 **Rate-Monotonic Scheduling Algorithm**



4.1.1 Subsystems

The term *subsystem* refers to the entities that are scheduled and dispatched for execution by the generated scheduler. The terms subsystem and task can be used interchangeably. By definition, a subsystem is an independently-scheduled program object, consisting of a single computational thread, which accepts inputs and posts outputs under the control of the scheduler at scheduler-specified times and which can be pre-empted.

Subsystems are constructed by AutoCode from all SuperBlocks in a system that have the same computational timing requirements or attributes (sample rates, skew, timing requirements, enable signals, and triggers). AutoCode has four types of subsystem:

- Continuous subsystem** Dispatched every time the dispatcher is invoked.
- Free-running periodic subsystem** Executed repetitively at a fixed frequency.

Enabled periodic subsystem	Executed repetitively, but only while its enabling signal remains active.
Triggered subsystem	Executed as and when its trigger is detected.

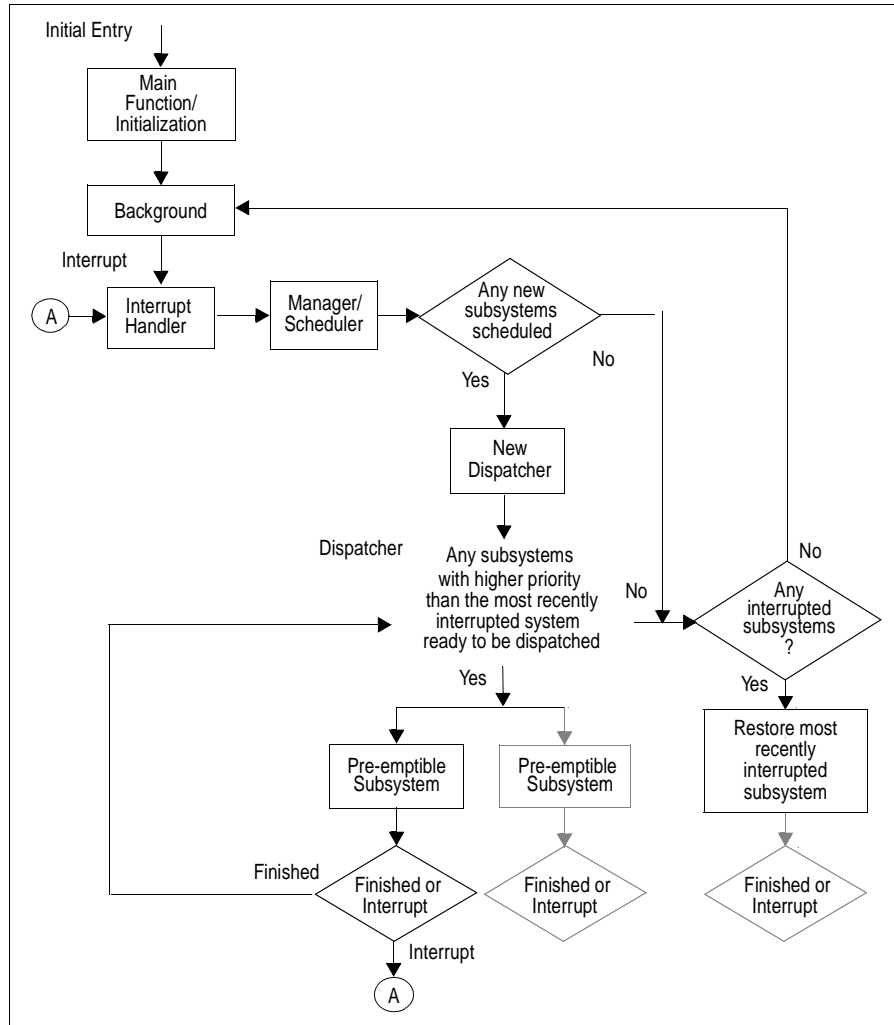
Throughout this discussion, the behavior of the scheduler and of the subsystems is explained in terms of interrupts or scheduler interruptions. These interruptions are implementation-dependent, involving a hardware timer interrupt, a wakeup call, or some other method of invoking the scheduler. The operation of the generated code is the same, regardless of which method of invoking the scheduler is used.

4.1.2 Flow of Control in the Generated Program

On start-up, initialization software (part of the standalone utilities) establishes a wakeup interrupt timing, time lines, priority queues, and initial conditions for the pre-emptible subsystems, and the manager/scheduler enters a ready state. As

illustrated in Figure 4-2, initial entry is to the background, which waits for the first interrupt or other wakeup action.

Figure 4-2 Flow of Control in the Generated Program



When the wakeup is received, the interrupt handler saves the interrupted context, if necessary, and passes control to the manager/scheduler. The scheduler checks external inputs and establishes a list of subsystems to be dispatched. It then posts

any external outputs and performs certain housekeeping before passing the dispatch list to the dispatcher.

The dispatcher is basically a big switch that passes control to the subsystem in the dispatch list that has highest priority. It always checks to see if there are any previously dispatched, but interrupted, subsystems with higher priority before dispatching a newly scheduled subsystem from its dispatch list. If there are, the most recently interrupted subsystem (which should be of highest priority among previously interrupted subsystems and newly scheduled subsystems) is restored by the interrupt handler and allowed to continue.

When the subsystem is finished, it passes control back to the dispatcher, which dispatches or restores the next-highest-priority subsystem, and so on. If all the currently dispatched subsystems and previously interrupted subsystems finish before a new timer interrupt is received, the interrupt handler returns control to the background. However, if a subsystem is still processing when the next timer interrupt is received, control passes to the interrupt handler, which again passes control to the manager/scheduler, which executes again.

In the case of Ada code with tasks representing subsystems, the dispatcher simultaneously dispatches all tasks that are ready. The Ada tasks have priorities associated with them which determine the CPU availability for each task.

If the manager/scheduler has nothing to schedule at the next timer interrupt, the scheduler passes control back to the interrupt handler. The interrupt handler then restores whatever was running at the time of the interrupt. If any subsystems or the dispatcher were interrupted part-way through their execution, the interrupt handler passes control back to whatever was running (subsystem or dispatcher) at the time of the interrupt. If no such dispatchers or subsystems remain, control returns to the background.

4.2 Sequence of Scheduler Operations

Figure 4-3, p. 65 illustrates the sequence of operations of the real-time application scheduler. The scheduler is represented as a bubble diagram (although it is not strictly a finite state machine), because during the “dispatch subsystems” phase (Bubble 9 in Figure 4-3), operations can be interrupted. During the critical section, however (Bubbles 1-8), it operates in the manner of a state machine. In the

discussion that follows, the term scheduler is sometimes used to refer to the critical section and dispatcher refers to the interruptible section.

Because the first eight steps in the scheduler's operation are non-interruptible, critical steps, it is a Wind River policy to optimize critical code for maximum performance and to execute in the same amount of time on every cycle. This minimizes output jitter and avoids performance problems.

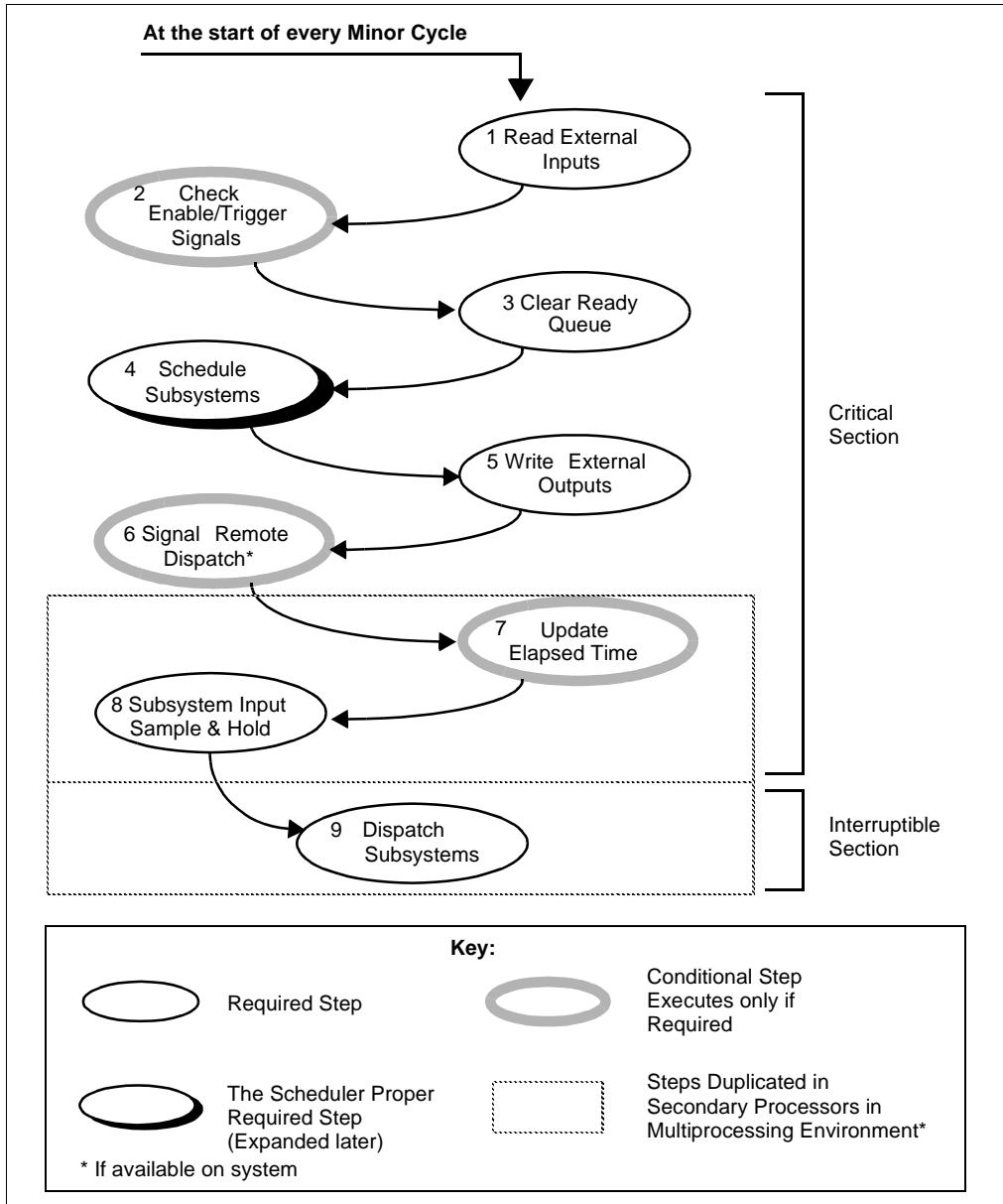
1. Read External Inputs

Bubble 1: On entry or re-entry, the scheduler collects the system external inputs so they can be used by the scheduled subsystems without a minor cycle delay. By definition, the minor cycle time of the application is the minimum scheduler cycle, a timing interval created from the sampling rates and timing requirements of all of the SuperBlocks in the system. The scheduler executes exactly once during each minor cycle.

2. Check Triggers and Enables

Bubble 2: The scheduler prepares for scheduling the subsystems by first determining which triggered and enabled subsystems are eligible to execute during this minor cycle. This step is not performed for systems that have no enabled or triggered subsystems. For enabled subsystems, the scheduler checks the subsystem state. If the state is blocked (that is, ready to run but waiting for an enable signal), the subsystem is ready to execute. If the enable signal is set, the scheduler queues it for execution.

Figure 4-3 Scheduler Operation



However, if the subsystem is in an idle state and the enable signal is true, the subsystem scheduler must determine whether the correct time has arrived for this subsystem to execute. If it is not yet time, the subsystem waits in the idle state until it is time to execute. See 4.3.2 *Enabled Periodic Subsystems*, p.71 for further details.

For triggered subsystems, if the trigger signal is true, the scheduler checks the subsystem state and proceeds appropriately for the subsystem type. At this stage, when a triggering signal is received and the triggered subsystem is in a blocked state waiting for a trigger, the scheduler queues it for execution. If the triggered subsystem is in an idle state, the subsystem is also queued for execution, but the outputs are posted or not posted, depending on the type of the subsystem. See 4.3.3 *Triggered Subsystems*, p.74 for further details.

3. Clear the Ready Queue

Bubble 3: The scheduler clears the ready queue for all the subsystems. The ready queue is established by the scheduler in Bubble 4 and used in Bubble 8 to determine which subsystems are to have their input sample-and-holds updated.

4. Schedule the Subsystems

Bubble 4: The scheduling algorithm is performed. For scheduling each of the subsystems, the scheduler checks for timing overflow (when a subsystem is still running, even though it is time to start running its next cycle; that is, the duration of the subsystem's execution has lasted longer than its cycle time). For each subsystem, the scheduler also checks all the criteria that determine whether the subsystem is to be dispatched.

These criteria are:

- Continuous

The continuous task is dispatched (through the integrator) every time the dispatcher is invoked.

An element of the scheduler is the Integrator (see [Figure 4-4](#), p.68). It performs continuous, fixed-step integration of states and implicitly dispatches the continuous subsystem to perform the state and output updates. The integrator/continuous task pair is by default treated as the fastest task to be dispatched by the scheduler. You can also specify the rate with the `-csi` option, but it must be at least as fast as the fastest periodic task in the model. For details regarding continuous code generation, see Chapter 5.

- Free-Running Periodic

The appropriate time has arrived, as determined by the time line table and the elapsed time counter.

- Enabled

The correct time has arrived, the enable signal is still true, and the subsystem was in the *Idle State* in the previous minor cycle (the state in which the enable signal is true, but the correct time has not yet arrived).

or:

The enable signal has just become true and the subsystem was in the *Blocked State* in the previous minor cycle (the state in which the enable signal is false).

- Triggered

The trigger signal has transitioned from false to true since the start of the last minor cycle. This condition is also checked for in Bubble 2. If the triggered block type is Asynchronous, and if the subsystem is to be dispatched by the Scheduler (that is, if the triggering signal is not an External Input), the subsystem triggers if the triggering signal has transitioned either from false to true or from true to false since the start of the last minor cycle. If the triggering signal for the Asynchronous subsystem is an External Input, the subsystem is dispatched separately from the Scheduler; see [Properties of Asynchronous Subsystems](#), p.78 for details.

Based on these criteria, the scheduler builds up the ready queue and adds the subsystems that are to execute to the dispatch list. The difference between the dispatch list and the ready queue is that subsystems that were on the dispatch list, but not dispatched on the previous cycle, are brought forward on the dispatch list for dispatching on this cycle or later. By contrast, the ready queue is cleared and built up again on every cycle.

As indicated in Bubble 4 and explained in more detail in the section that follows, the scheduler uses the computational attributes of the subsystems to establish the priority for dispatching the subsystems. The computational attributes include the sample rate and the type of the subsystem. The priority sequence established using the computational attributes runs from fastest sample rate or timing requirement to slowest. For a tie in sample rate or timing requirement, the priority for execution is based on the type of subsystem, from highest to lowest:

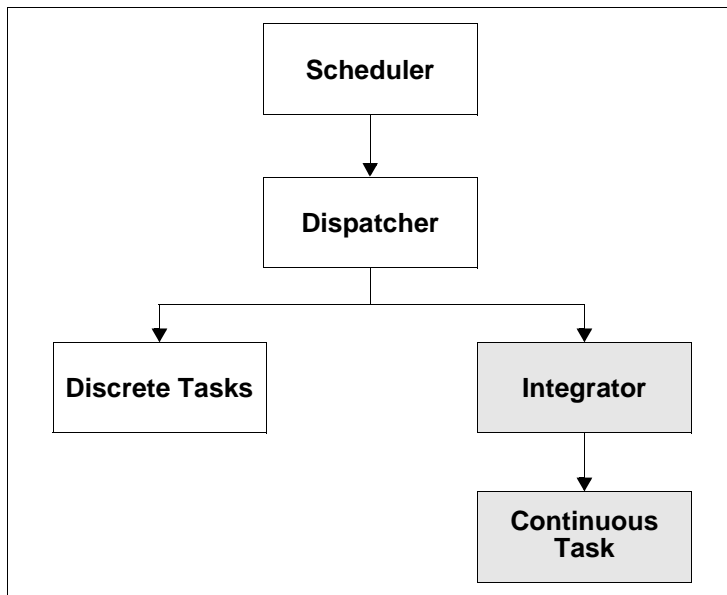
Continuous

Free-Running Periodic

- Enabled Periodic
- Triggered Asynchronous
- Triggered As-Soon-As-Finished
- Triggered At-Timing-Requirement
- Triggered At-Next-Trigger

Using the above priority rules, the subsystems are assigned their task IDs in the sequence 1, 2, 3, ... NTASKS, with the highest priority subsystem getting the lowest ID number (that is, 1) and the lowest priority subsystem getting an ID of NTASKS. NTASKS is the number of tasks in the system and is also the name of the variable in the code which represents that number. The task IDs are assigned at code generation time.

Figure 4-4 Scheduler Architecture



5. Write External Outputs and Signal Remote Dispatch

Bubbles 5 & 6: The scheduler calls the external output routine to post all subsystem outputs at every minor cycle. In multiprocessor implementations only, the dispatch list and a remote dispatch signal are posted to the

secondary processors (Bubble 6) to signal the availability of the dispatch list and mark the start of subsystem execution.

6. Update Elapsed Time and Sample & Hold

Bubbles 7 & 8: In multiprocessor implementations, every secondary processor's scheduler subsystem may or may not need to perform an elapsed time update (Bubble 7), but will be required to perform a "sample and hold" subsystem input (where the scheduler reads the inputs and latches them for use by the subsystem, Bubble 8) and a subsystem dispatch (Bubble 9). The scheduler updates the elapsed time counter (Bubble 7), if required. Bubble 8: the scheduler in each processor consults the ready queue to perform a sample-and-hold on the subsystem inputs for the subsystems that have been added to the dispatch list. For determinacy reasons, subsystems remaining in the dispatch list from a prior cycle will not have their inputs sampled again. This step is the end of the critical section.

7. Dispatch Subsystems

Bubble 9: The dispatcher is re-entrant and it can be interrupted at any point in its operations. This re-entrant step accepts the dispatch list and queues the subroutines for execution. This step also includes the execution of subroutines, which can be interrupted/pre-empted at any time.

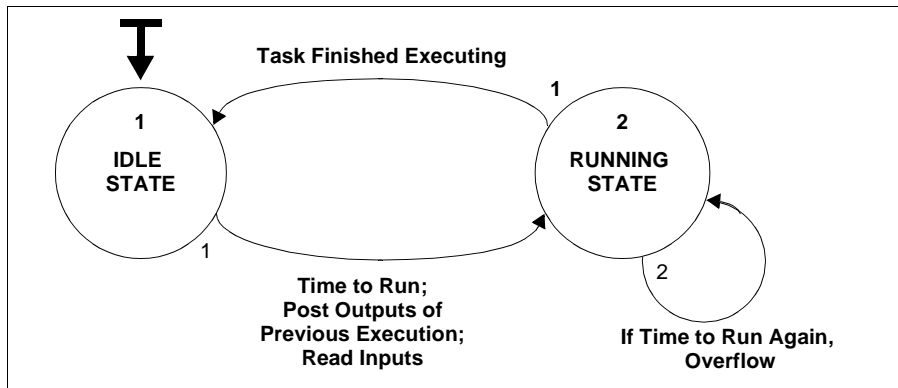
4.3 Properties of Scheduled Subsystems

A scheduled subsystem is viewed as a finite state machine that is represented as a State Transition Diagram (STD). A finite state machine always exists in exactly one of its defined states, where it remains until some change forces it to transition to another state. No more than one transition can take place on each cycle of the subsystem in which the STD resides. These STDs have the same timing conventions as other state machines in SystemBuild. The STDs are illustrated in [Figure 4-5](#), p.70, [Figure 4-6](#), p.71, and [Figure 4-8](#), p.76; associated timing diagrams are shown in [Figure 4-7](#), p.73 and [Figure 4-10](#), p.78.

4.3.1 Free-Running Periodic Subsystems

Figure 4-5 shows the operation of a free-running periodic subsystem. A free-running subsystem is always enabled and it exists in its idle state until the scheduler decides it is time for the subsystem to run (that is, its sample time has arrived). At that time, the subsystem posts its outputs and enters the running state. This also applies on start-up, when the subsystem theoretically has not yet executed and has no outputs to post; the simulation user can control the way that this start-up output is generated by using the `sim...{initmode}` command (see the *SystemBuild User's Guide*).

Figure 4-5 Free-Running Periodic Subsystem as a State Machine



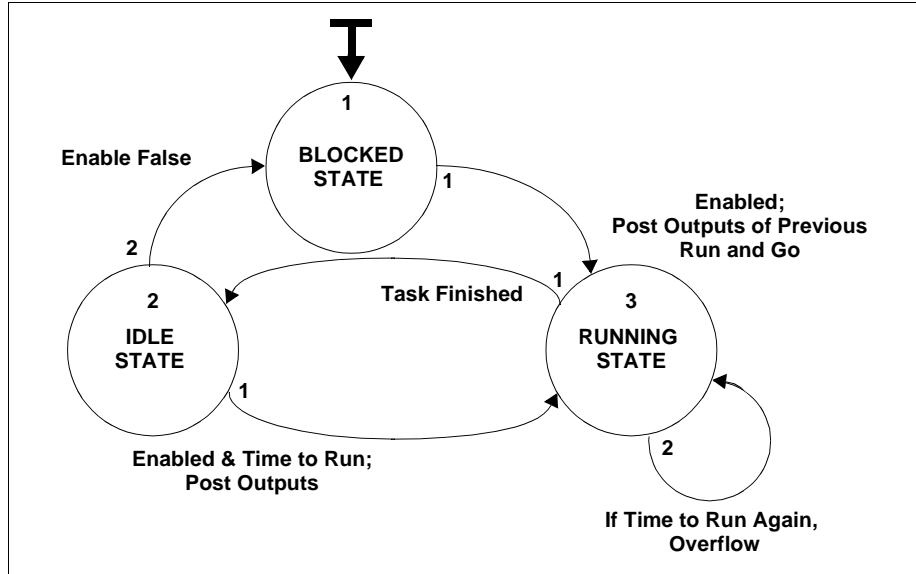
A subsystem accepts its inputs just before starting; this event is associated with the transition before the running state is entered. With specified exceptions, the subsystem posts its previous outputs at that same time (that is, just before running). The exceptions are associated with enabled and triggered subsystems and are shown in the State Transition Diagrams.

When a subsystem is running (that is, manipulating its outputs), the outputs of the previous cycle are latched (double-buffered) and can be read. Note the overflow condition that occurs when a subsystem is in the running state and it becomes time for it to start running again. This means that it did not finish its execution in time. This is a fatal error in systems that must operate in a real-time environment. When the subsystem finishes, it performs the appropriate housekeeping subsystems and then returns to the idle state; but for reasons of determinacy, it does not post its new outputs until it is started up again.

4.3.2 Enabled Periodic Subsystems

Figure 4-6 shows the state diagram for an enabled periodic subsystem, illustrating the blocked state during which it is disabled.

Figure 4-6 Enabled Periodic Subsystem as a State Machine



In simulation, these subsystems can be scheduled to run only on a predefined time line established by the subsystem's sample rate and when the subsystem is enabled at the same time. If a disabled subsystem is enabled after its synchronous start time in the time line, it has to wait until its next major cycle (the repetition time of the enabled periodic subsystem) to run. The blocked state is used in the generated code to eliminate latencies that would occur if a disabled subsystem were to receive its enable signal several minor cycles before its synchronous start time in the time line (its major cycle).

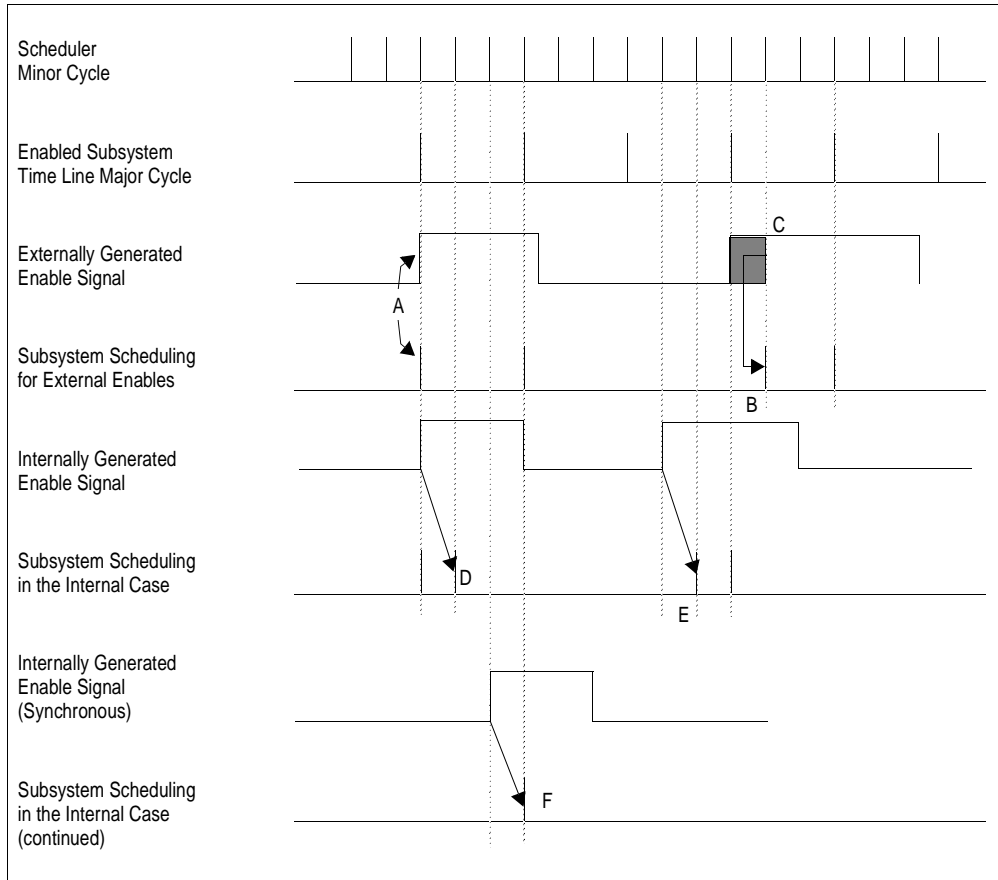
When the enable goes true while the subsystem is in the blocked state, the subsystem is scheduled to run at the next minor cycle; if it has priority over other contending subsystems, it is executed immediately, at the same minor cycle of the real-time scheduler. The behavior of the scheduler can differ, however, depending on whether the enable signal is generated internal to the system (as the output of another subsystem) or presented as an external input to the system. See the timing diagram (Figure 4-7, p.73) for these distinctions.

In the example shown in [Figure 4-7](#), the major cycle is three times as long as the minor cycle of the system. When an external enable signal is detected at the beginning of a major cycle, (as shown at point A); the enabled subsystem is dispatched for execution immediately.

In [Figure 4-7](#), item B shows the case in the generated code concerning enables that are presented as external inputs, asynchronous to the original time line. The scheduler attempts to process these inputs as quickly as possible. This is also shown in Bubble 2 of [Figure 4-3](#), p.65, where external inputs are gathered immediately upon initialization of the scheduler, so they can be processed without delay. In [Figure 4-7](#) at B, in the generated code the subsystem is queued for execution at the minor cycle following the instant when the enable signal was detected true. Note the timing window indicated at C. During that interval, it does not matter whether the external enable is presented synchronously with the

minor cycle or before it; both the simulation and the generated code operate identically.

Figure 4-7 Enabled Subsystem Timing



At D, the signal appears at the start of a major cycle and is applied immediately. At E, the signal appears one minor cycle after the start of a major cycle and incurs a two-minor-cycle delay.

At F, the enable occurs one minor cycle before the beginning of the major cycle and consequently, the delay is one minor cycle. By contrast, at points D, E, and F, illustrating the operation of generated code, the internally generated enable signal is also synchronized through the application scheduler. It is always delayed exactly one minor cycle before being applied. The reason for this delay is implied

in Bubble 8 of [Figure 4-3](#), p.65. Here, input sample-and-holds are performed after the determination of which subsystems are to be dispatched for execution on this (the first) minor cycle, so the application of the synchronous enable in generated code is always deferred for one minor cycle.

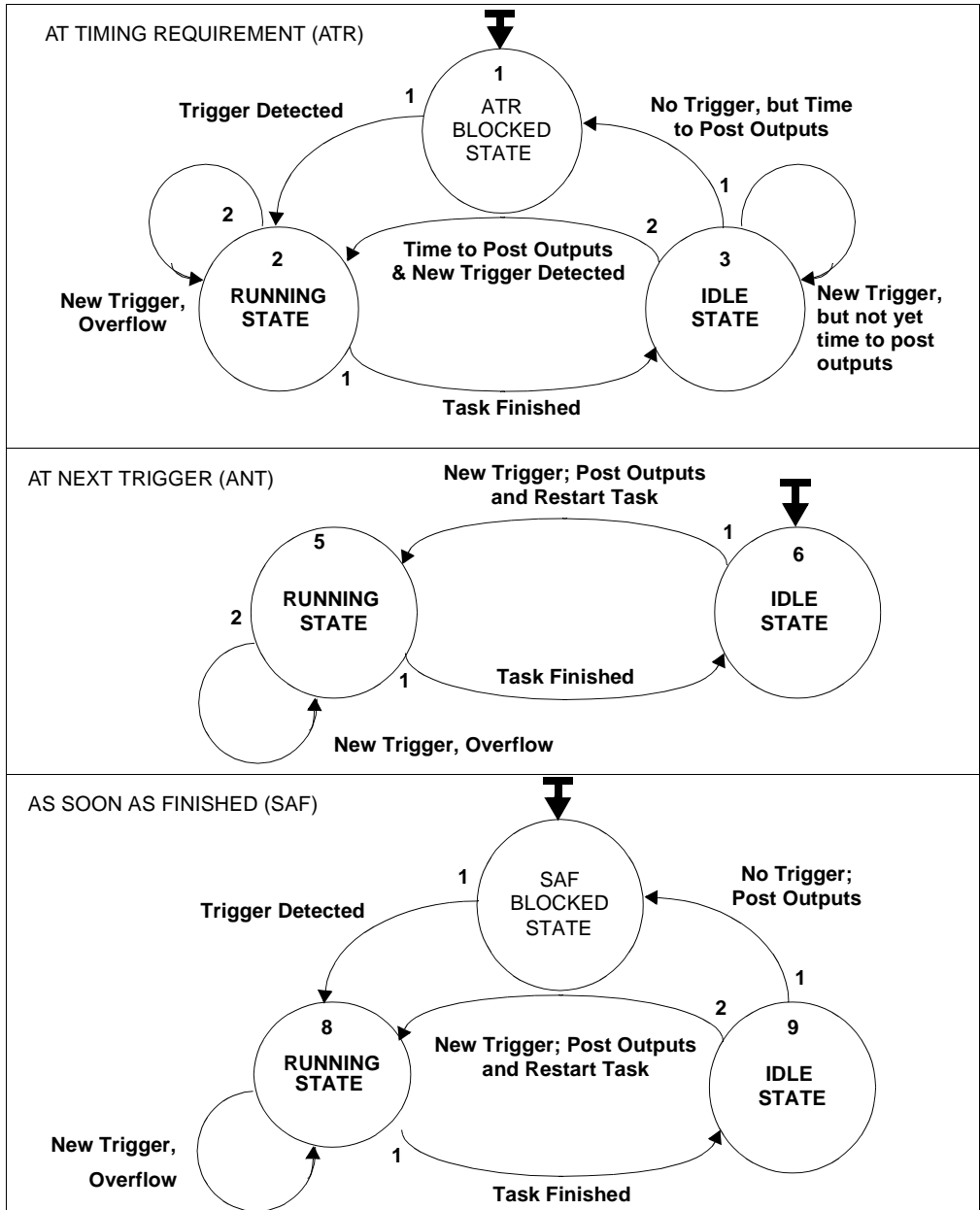
4.3.3 Triggered Subsystems

[Figure 4-8](#), p.76 shows the State Transition Diagrams for triggered, asynchronous subsystems. The types of triggered subsystems differ mainly in the manner in which they post their outputs. Just like enabled subsystems, certain types of triggered subsystems have a blocked state, from which they can be invoked immediately when the trigger is detected. (By default, the trigger is defined as the positive-going edge of the triggering signal, but provision is made in the template files for changing it to detect a negative-going edge.) The properties of the types of triggered subsystems are illustrated as State Transition Diagrams in [Figure 4-8](#), p.76, and [Figure 4-9](#), p.77 (for asynchronous), and in a consolidated timing diagram in [Figure 4-10](#), p.78. These properties are summarized as follows:

At Timing Requirement (ATR)	Specify a timing requirement (number of system minor cycles) in the SuperBlock block form. The outputs are always posted exactly that number of cycles after the subsystem is triggered for execution. ATR is used especially in systems where determinacy must be guaranteed.
At Next Trigger (ANT)	The subsystem only posts its outputs when it is next triggered for execution, however long that may be. ANT is used for modeling certain kinds of variable rate, but repetitive activities, such as a shaft that rotates at a variable speed. This type of subsystem has no blocked state.

As Soon As Finished (SAF)	The outputs are posted at the beginning of the minor cycle after the subsystem finishes running. SAF is used for maximum performance, but might compromise determinacy.
Asynchronous (ASYNC)	If dispatched by the scheduler, the outputs are posted at the beginning of the next minor cycle after the subsystem finishes running. This will occur if the triggering signal is an output of another subsystem. If dispatched as a result of an asynchronous interrupt, the outputs are posted as part of the interrupt handler, and are available to scheduled systems immediately.

Figure 4-8 Triggered Subsystems as State Machines



In Figure 4-10, p.78, a timing requirement equal to four scheduler minor cycles has been established. When the trigger signal is received, the subsystem is started at the next scheduler minor cycle (point A1). In this example, although the timing requirement is four (point A2), the subsystem actually completes execution in slightly less than three cycles. This illustrates the kinds of output posting:

- ASYNC - The output is seen at point B, three cycles after the subsystem was started, assuming this subsystem was dispatched by the scheduler.
- SAF - The output is seen at point B, three cycles after the subsystem was started.
- ATR - The output becomes available at point C, exactly four cycles after start-up.
- ANT - The output is not available until point D, when the subsystem is next triggered for execution.

Figure 4-9 **ASYNCHRONOUS Triggered Subsystems as State Machines**

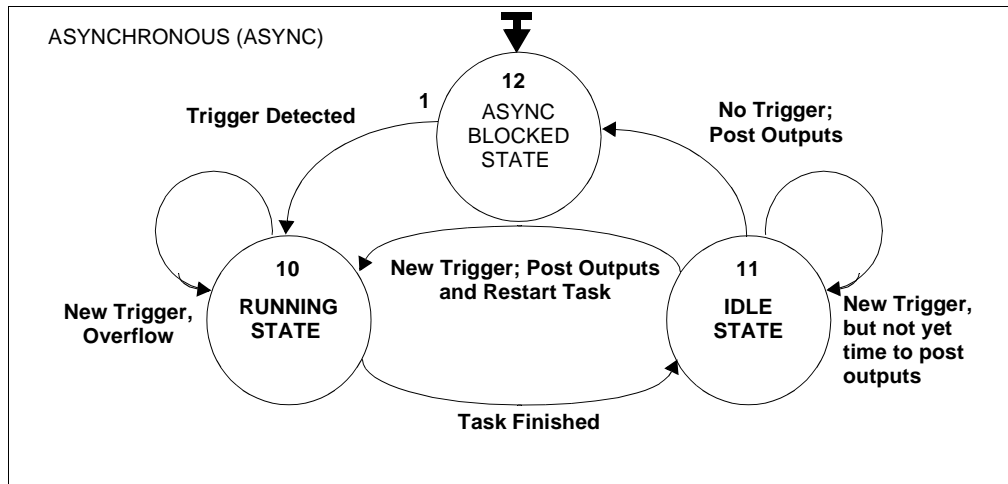
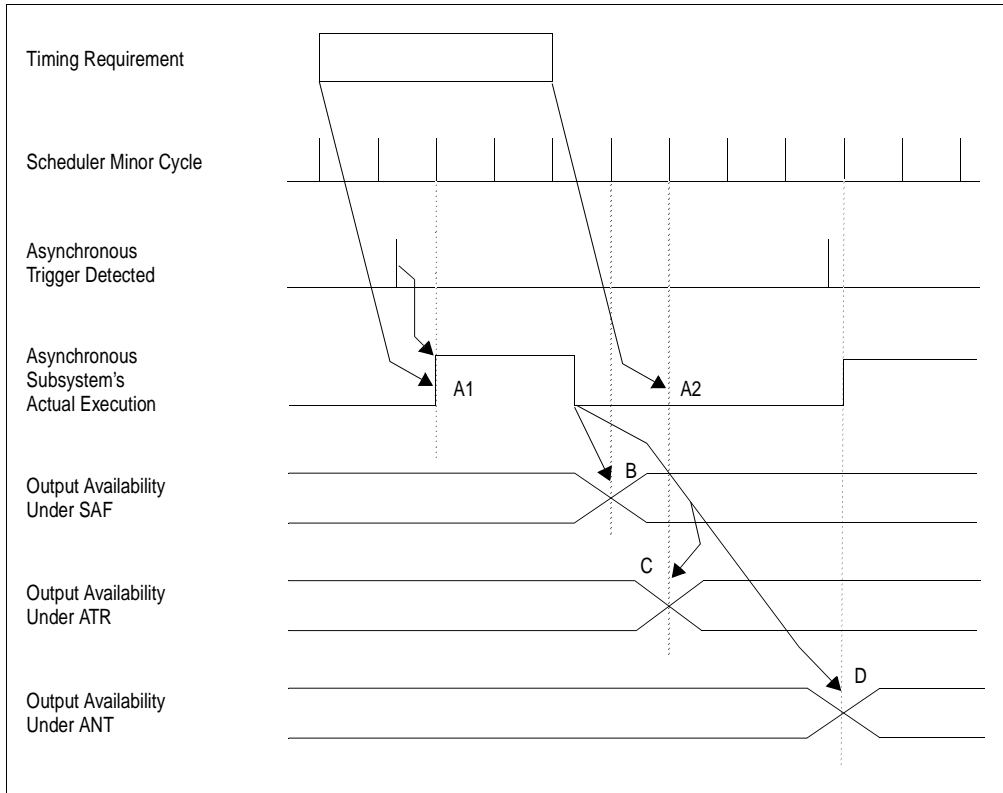


Figure 4-10 Timing of Triggered Subsystems



4.4 Properties of Asynchronous Subsystems

Asynchronous subsystems (procedures or asynchronous trigger subsystems) are so called because of their noncyclic or unscheduled execution in a real-time application. These subsystems should be viewed as special purpose entities and should be used accordingly.

The asynchronous subsystems are not directly managed or scheduled by the application scheduler as are the synchronous subsystems. Scheduled subsystems can execute only at the start of a scheduler minor cycle and not instantaneously

(for example, at the arrival of an external event). This adds latency to the execution of certain subsystems, which absolutely cannot wait until the next minor cycle for execution. Asynchronous subsystems are designed to solve this problem.

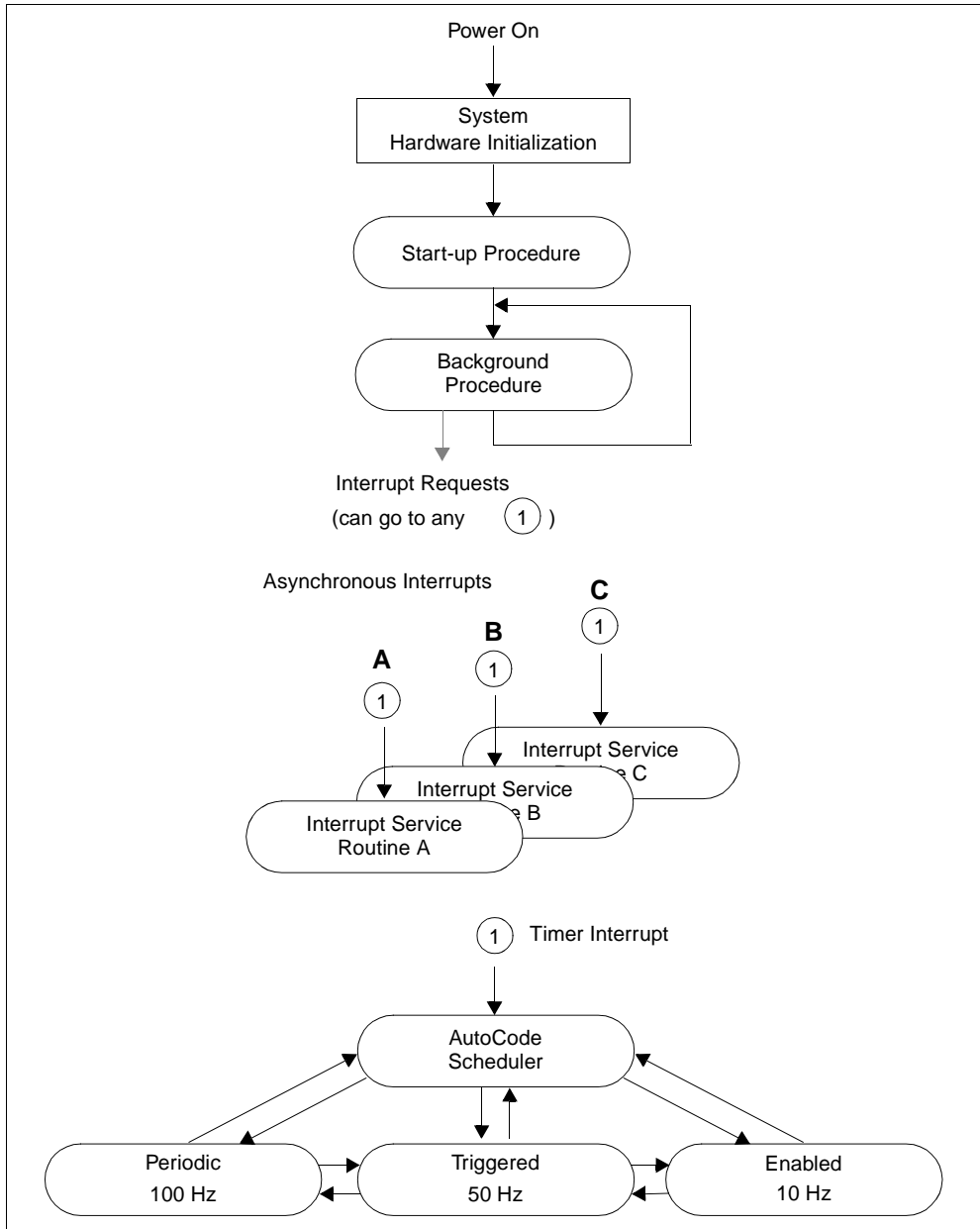
The kinds of asynchronous subsystems (see [Figure 4-11](#), p.80) that you can generate in an AutoCode application are:

- Start-up procedure
- Asynchronous subsystem
- Interrupt procedure
- Background procedure

4.4.1 Start-up Procedure

This procedure is defined in SystemBuild using the Start-up Procedure SuperBlock. The purpose of this start-up procedure is to initialize the application data at start-up time. This data includes variable block data and %variables represented by variable blocks. It is only via the Startup SuperBlock that you can initialize the %variables in SystemBuild at run time. Usually, the start-up procedure is called at the system initialization phase. Refer to the *SystemBuild User's Guide* for more details on the Startup SuperBlock description.

Figure 4-11 AutoCode Real-Time Application Execution Sequence



4.4.2 Asynchronous Trigger Subsystems

Subsystems formed by collections of Asynchronous Triggered SuperBlocks (ATSBs) with the same triggering signal are handled differently depending upon the source of the triggering signal.

If the triggering signal is internal to the model (that is, if the triggering signal is not an external input), these subsystems function similarly to the Triggered - Soon As Finished subsystems with the following exceptions:

- The ATSB subsystems have higher priority than the SAF triggered subsystems (that is, they are executed before other triggered subsystems).
- The triggering signal to ATSB subsystems is double-edged; the ATSB subsystem will be scheduled if its triggering signal transitions from low to high or from high to low during the previous scheduler cycle. This is to maintain compatibility with the use of these subsystems in simulation.

If the triggering signal is external to the model (that is, the triggering signal is an external input), then the ATSB subsystem is handled specially in the template. In this case, two pieces of code are generated; the regular (nonreentrant) triggered subsystem code, and a small (reentrant) wrapper that is designed to function as an interrupt service routine (ISR). This wrapper is wholly generated in the template, and can be customized for the user's application. In the templates provided by Wind River, the wrapper for the ATSB subsystem does the following:

- Checks for reentrancy, and reports an error if the wrapper is reentered. Note that the wrapper could instead queue calls to the subsystem code, but under no circumstances should the subsystem code be reentered.
- Gathers the subsystem inputs from external inputs and outputs from other subsystems.
- Invokes the ATSB subsystem.
- Posts the ATSB's subsystem outputs, updating the external output structures as necessary.

In the default template, the ATSB wrapper is a procedure requiring neither inputs nor return values.

Note that ATSB subsystems differ from Interrupt Procedure SuperBlocks (see [Asynchronous Trigger Subsystems](#)) in the following ways:

- The inputs and outputs to these subsystems can be represented in the model like those of other standard SuperBlocks without incurring processing overhead at interrupt time.

- These subsystems are simulatable by the SystemBuild simulator.
- States are supported in ATSB subsystems, but not in Interrupt Procedure SuperBlocks.
- Calls to Procedure SuperBlocks are supported on ATSB subsystems.
- These subsystems are not reentrant, even though the wrapper is reentrant, and thus can be designed in such a way to support asynchronous interrupts that may occur before the processing of the ATSB subsystem code is complete.

For more information on Asynchronous Trigger SuperBlocks, see the *SystemBuild User's Guide*.

4.4.3 Interrupt Procedure

This procedure is defined in SystemBuild using the Interrupt Procedure SuperBlock. The purpose of this SuperBlock is to model the Interrupt Service Routine (ISR). This ISR model in SystemBuild is generated as an interrupt procedure by AutoCode and can be executed on arrival of a specific interrupt signal. Using variable blocks, implicit communication can be established to any other subsystem in the application.

The execution of the interrupt procedure is done directly on arrival of an external interrupt or an event; the AutoCode scheduler does not manage, monitor, or schedule this procedure. However, since the AutoCode scheduler, as supplied, employs rate monotonic scheduling principle, it is possible that interrupt procedure execution will overflow the synchronous task execution. Interrupt procedure users should keep the interrupt procedure execution time to a minimum and should keep enough time buffer in each scheduler cycle for possible execution of interrupt procedures. Please refer to the *SystemBuild User's Guide* for more details on the Interrupt Procedure SuperBlock description.

4.4.4 Background Procedure

This procedure is defined in SystemBuild using the Background Procedure SuperBlock. The purpose of the background procedure is to represent the logic executed when the system is idle, in the background mode of operation. Typically, the background procedure should be executed as the lowest priority task and only if no other tasks need to be performed in the system. Please refer to the *SystemBuild User's Guide* for more details on the Background Procedure SuperBlock description.

4.5 Reentrancy and Preemption: The Dispatcher

The generated application program is interruptible except at the critical section of the scheduler. The scheduler is automatically created by AutoCode, using the template file, to provide input/output calls, scheduling, error handling, and dispatching services for the generated application program. All the services, except for dispatching of the subsystems, are performed in the critical section. The critical section is kept as brief as possible; one of several reasons for this minimization is to allow maximum time for the subsystems to execute.

The subsystems operate under different constraints as compared to those of the time-critical scheduler. A subsystem cannot execute more frequently than the scheduler does and in many cases, it will run far less often. However, the subsystem may require considerable time for each execution pass. Consequently, it can be interrupted repeatedly by scheduler execution. Thus, the subsystem code must be completely interruptible. It must be able to be interrupted by the scheduler and thus to be pre-empted by higher priority subsystems. It must also be able to restart at any point in its operations.

The only timing requirement of the subsystem is that it must finish executing before the next time it is to be queued for execution. A subsystem being ready to run and simultaneously not finished running, defines the condition called “subsystem timing overflow.” This is a catastrophic error in any system that requires deterministic operation.

The scheduler can add subsystems to the dispatch list at any cycle of the scheduler’s operations. However, the dispatcher only removes the subroutine from the list when the subsystem has begun its operations. Determinacy and proper operation of a pre-empted subsystem both demand that the inputs receive a sample-and-hold to a subsystem only once per execution, when first queued. As a result, a second list, the ready queue, is employed to determine which subsystems will have their inputs sampled and held this cycle. The ready queue is cleared and set up by the application scheduler once every minor cycle.

When the ready queue is cleared, the scheduler determines which subsystems are to be queued for dispatch this cycle and places the subsystems into the ready queue and the dispatch list. Note that the scheduler never removes a subsystem from the dispatch list; only the dispatcher has that duty. The following examples explore these ideas graphically, following the scheduler and dispatcher through a few cycles of operation, showing how the subsystem states, the ready queue, and the dispatch list change over time.

Other examples show the operation of a scheduler with more complex timing requirements (“pseudo-rate scheduler”) and its special conditions and error conditions.

4.6 Scheduler Examples

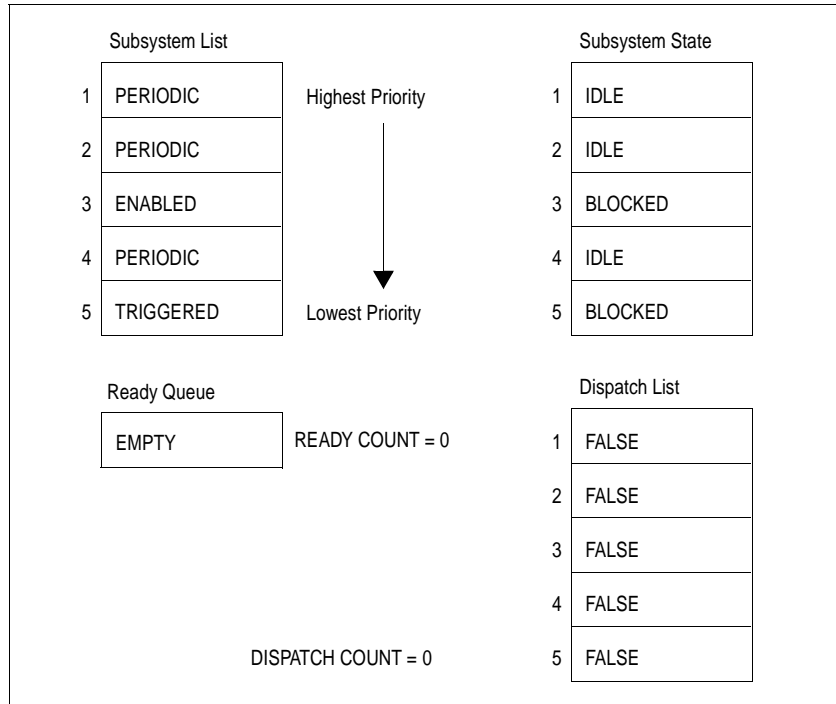
The examples presented in the subsections that follow describe scheduler operation both when sampling rates and timing requirements for all subsystems are common multiples and when they are not common multiples, thus requiring a pseudo-rate for the scheduler. Operating with skew is also discussed.

4.6.1 Dispatching and Pre-emption Example

The illustrations, starting with [Figure 4-12](#), show a system with five subsystems: three periodic, one enabled, and one triggered. The subsystem list identifies

which subsystems are periodic, enabled, or triggered, and shows them in priority order.

Figure 4-12 Scheduler Data Structures at Initialization: 1

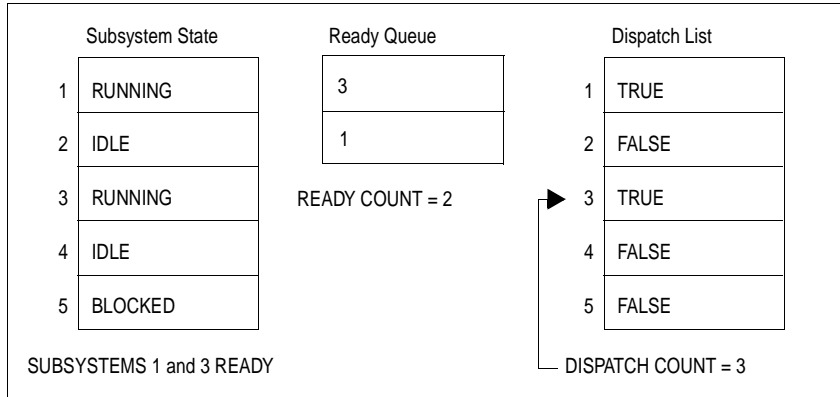


The subsystem state list identifies the state of the subsystem. The ready queue is used by the scheduler to determine which subsystems are to have their inputs sampled and held. When the scheduler determines that a subsystem is to be dispatched for execution, the subsystem is placed into this list. When a subsystem is dispatched for execution, it is removed from the list by the dispatcher. In [Figure 4-12](#), the subsystems are idle or blocked, each awaiting its condition to start running.

In [Figure 4-13](#), the time has arrived for subsystem 1 to execute and the enable signal for subsystem 3 has been received. The numbers of the two subsystems are entered into the ready queue in reverse order of priority and the ready count is set to equal the number of subsystems that are ready (that is, 2). The corresponding entries in the dispatch list are marked true. The dispatch count (which is the pointer to the lowest priority subsystem that is ready for dispatch) is set to 3. The

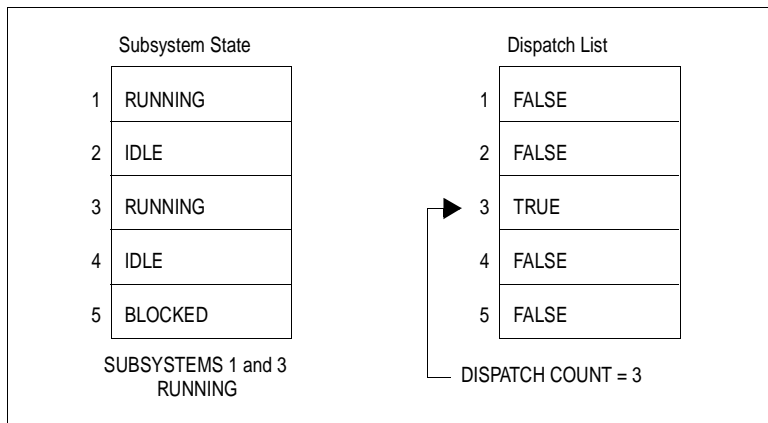
immediate effect of this is that the inputs for subsystems 1 and 3 are sampled and held, and control is passed to the dispatcher.

Figure 4-13 **Scheduler Data Structures: 2**



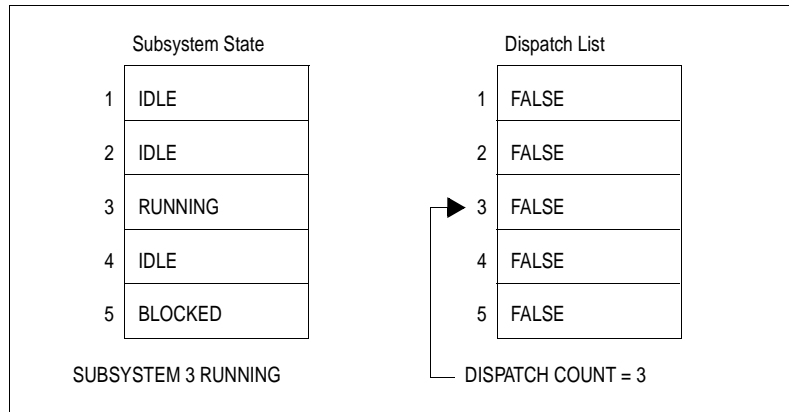
The action of the dispatcher is shown in [Figure 4-14](#), where subsystem 1 (the highest priority subsystem) is actually executing and subsystem 3 is waiting to be dispatched. The dispatch count is still equal to 3, pointing to the subsystem of lowest priority that is still either running or in the dispatch list. When subsystem 1 completes its operations (is no longer running), the system moves to the state shown in [Figure 4-15](#).

Figure 4-14 **Scheduler Data Structures: 3**



In [Figure 4-15](#), subsystem 1 is marked idle in the subsystem state table and subsystem 3 is dispatched and running. All the entries in the dispatch list are marked false, indicating that no subsystems currently need to be dispatched. However, the dispatch count pointer is still pointing to subsystem 3, indicating that it is not finished executing. Now, while subsystem 3 is still working, a scheduler interruption occurs.

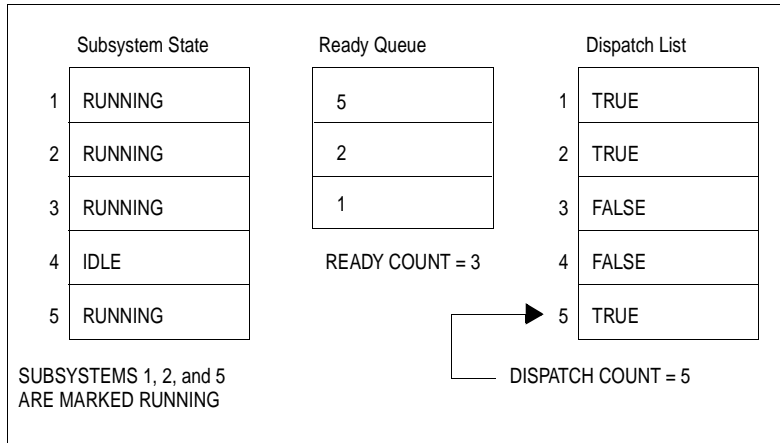
Figure 4-15 **Scheduler Data Structures: 4**



The scheduler is re-entered and, as a part of the scheduling loop, subsystems 1, 2, and 5 are all marked ready (see [Figure 4-16](#)). Subsystems 1, 2, and 5 have been entered into the subsystem state list as running and the interrupted subsystem 3 remains in a running state. Therefore, subsystems 1, 2, and 5 are placed in the ready queue to have their inputs sampled and held. Note that subsystem 3 cannot be placed in the ready queue, because it has already received its inputs for this

operations cycle. Now, subsystem 1 and then subsystem 2 will be dispatched for execution.

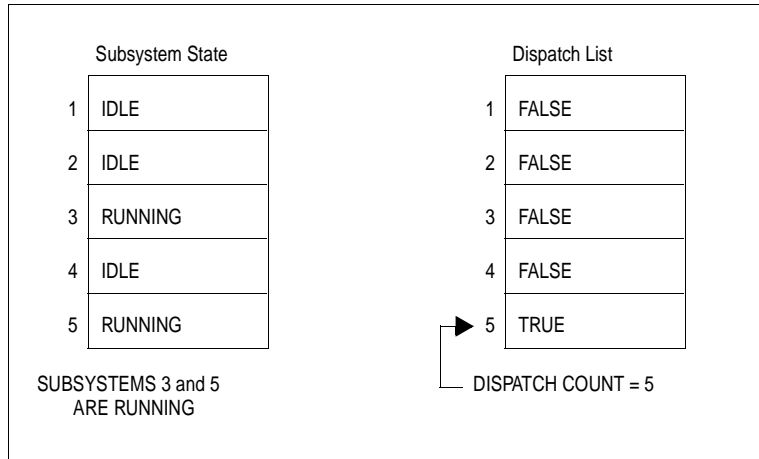
Figure 4-16 Scheduler Data Structures: 5



When subsystems 1 and 2 are both finished (see [Figure 4-17](#)), the dispatcher notes that subsystem 3 is in a running state, but is not in the dispatch list. The dispatcher checks its records and determines that subsystem 3 was actually in the process of executing when the next scheduler cycle occurred. Thus, the subsystem needs to have its context restored by the interrupt handler, not by the real-time application scheduler. Therefore, the scheduler dispatcher performs an

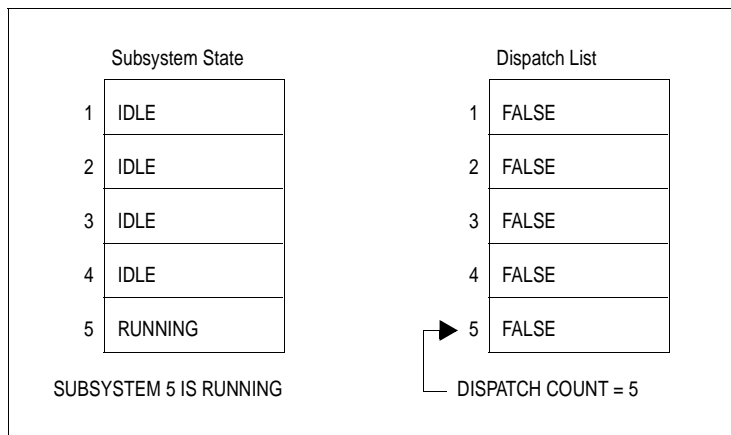
exit, which passes control back to the operating system interrupt handler. The interrupted subsystem is restored from there.

Figure 4-17 **Scheduler Data Structures: 6**



When subsystem 3 is finally finished executing, as shown in [Figure 4-18](#), subsystem 5 is dispatched and executes.

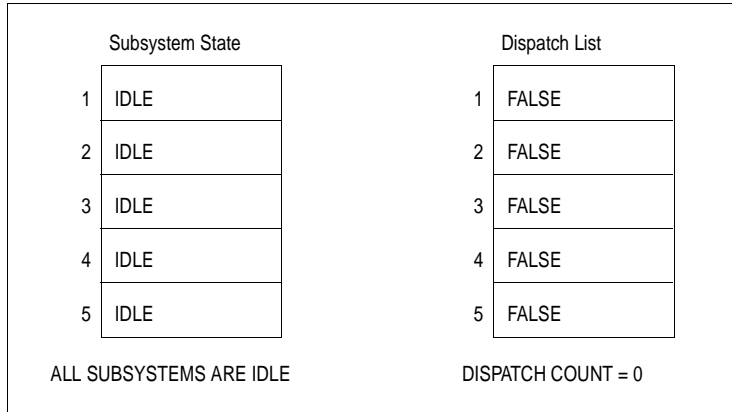
Figure 4-18 **Scheduler Data Structures: 7**



Finally, when subsystem 5 is also finished, the situation is as represented in [Figure 4-19](#). At this point, no subsystems are running, nothing is to be dispatched, and the next interruption for scheduler operation is sometime in the future. The

dispatcher responds by passing control to the operating system interrupt handler, which restores and passes control to the background subsystem.

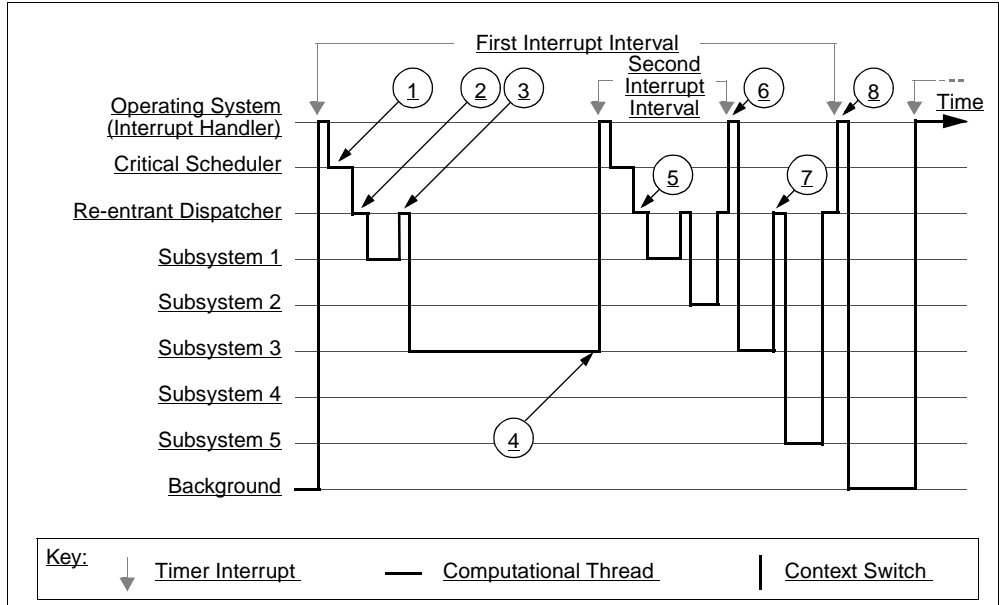
Figure 4-19 Scheduler Data Structures: 8



This background subsystem consists of interruptible code that does not return, but waits to be interrupted. It might be nothing but a loop that waits to be interrupted, or it might perform any of a range of low-priority program tasks, such as self-diagnosis and updating displays. For example, some Interactive Animation displays are updated by the background subsystem.

The operation of the scheduler and the subsystems in this example are shown in the form of a timing diagram in Figure 4-20. The numbers in circles correspond to the numbers at the end of each figure title associated with each of the figures for this example.

Figure 4-20 **Dispatcher Example as a Timing Diagram**

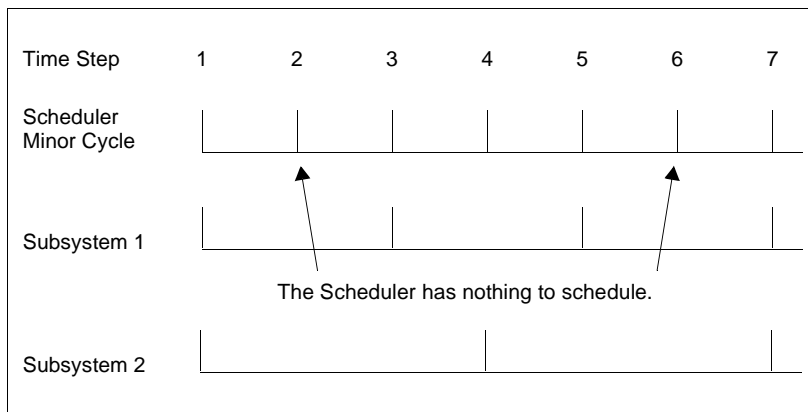


4.6.2 Pseudo-Rate Scheduler

The previous example assumed that the repetition rate of the scheduler (the scheduler minor cycle) was the same as the sampling rate of the fastest subsystem, subsystem 1. This correspondence holds true only if the sampling rates and timing requirements of all of the subsystems are common multiples. Thus, in the example, the sampling rate of subsystem 1 might be 1 unit, the sampling rate of subsystem 2 might be 2 units, and that of subsystem 4 might be 3 or more units.

However, if the sampling rates or timing requirements of the subsystems are not even multiples, AutoCode establishes a “pseudo-rate” for the scheduler, based on the least common multiple of the rates of the subsystems. The simplest case of a pseudo-rate is shown in Figure 4-21, where there are two free-running subsystems: subsystem 1 with a sampling rate of 2 and subsystem 2 with a rate of 3. At time step 1, the scheduler places both subsystems into the dispatch list and they are both executed. But at step 2, the time to execute has arrived for neither of them and nothing runs; the same thing is true at step 6. However, the scheduler still must complete its cycle of operations, that is, the first 7 steps of Figure 4-3, p.65, even though there is nothing ready to be scheduled. The cycles thus wasted might have a negative impact on system performance. For this reason, use sampling rates that are all even multiples in systems where performance is an issue.

Figure 4-21 Dispatcher Operation with a Pseudo-Rate Scheduler



What happens at the beginning of step 2 depends on whether or not the subsystems had completed running before the end of the last cycle. If a subsystem (subsystem 2, presumably) had still been running when the scheduler interruption occurred, at the time the scheduler completed its cycle with nothing in the dispatch list, the dispatcher would pass control back to the interrupt handler. The interrupt handler would invoke the operating system to restore the interrupted subsystem and pass control to it. If all subsystems had finished operation, then it would have been the background subsystem that was interrupted. The operating system would restore this subsystem instead and pass control to the background.

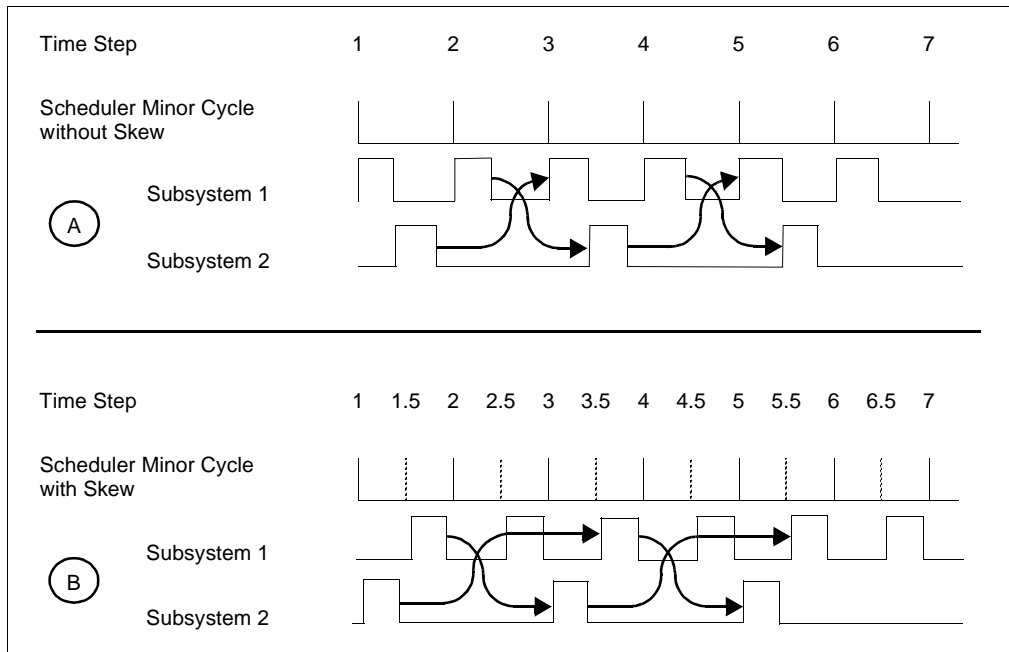
4.6.3 Operating with Skew

Skew, or First Sample in the SuperBlock block form, is a method for controlling the operation of the subsystems on the time line. The sampling interval lets you specify the periodicity of the subsystem, and the first sample lets you establish an offset from the beginning of the minor cycle on which the subsystem first becomes eligible to execute. One of the uses for skew is to force a slower-rate subsystem to execute before a faster-rate one, when either could run on the same cycle.

As shown in [Figure 4-22](#), at A, subsystem 1 is running at twice the sampling interval of subsystem 2. Therefore, subsystem 1 has priority. It is assumed that both subsystems receive external input data and that each subsystem posts outputs to the other. At time step 1, when the system is started, both subsystems receive as internal, sampled-and-held inputs whatever initial states you might have defined. But at time step 3, the outputs of subsystem 2 from its first major cycle are latched as inputs to subsystem 1 and the system is running in sync. The outputs from subsystem 1's operations during its secondary cycle are latched and fed to subsystem 2 to serve as its inputs at time step 3.

From that time forward, the outputs of every major cycle of subsystem 2's operations are presented as inputs for the next (odd-numbered) cycle of subsystem 1's operations (steps 3, 5, ...). Those same inputs are still visible to subsystem 1 on its next even-numbered cycle (steps 4, 6, ...). And subsystem 1's outputs from the even-numbered (minor) cycle serve as inputs to subsystem 2 on the odd-numbered (minor) cycle (latched at steps 3, 5, ...).

Figure 4-22 Operation of Skew



Subsystem 2 never sees the outputs of subsystem 1 from its odd-numbered cycle, for there is no sample-and-hold performed between the end of subsystem 1's odd-numbered step and the beginning of subsystem 2's operations. In the bottom part of the figure, at B, a skew of 0.5 has been added to the timing properties of subsystem 1 and no other change has been made. But the system will operate quite differently from A. First, the scheduler minor cycle is now a pseudo-rate, introduced so that the start-up of subsystem 1 can be scheduled and dispatched correctly. Also, characteristic of pseudo-rate schedulers, there are steps where the scheduler has no scheduling to do (steps 2, 4, 6, ...). Even so, the critical part of the scheduler must go through its full cycle of operations. This, of course, has an impact on overall system performance.

Observe that even though subsystem 1 would have priority, subsystem 2 starts executing before subsystem 1 in this example, because subsystem 2's time to run arrives before that of subsystem 1. Thus, at step 1, subsystem 2 starts up with initial states as its inputs and posts its outputs at time step 3 to be latched and made inputs to subsystem 1 at time step 3.5.

Now, as at A, subsystem 1 runs through two full cycles of its operations before subsystem 2 can run again. When subsystem 1 starts at timestep 1.5, its outputs are posted at timestep 2.5, and when it starts at timestep 2.5, its outputs are posted at timestep 3.5, and so on. Unlike example A, the outputs from subsystem 1's odd numbered cycles are visible to subsystem 2 at time steps 3, 5, ..., and the outputs of subsystem 2 posted at time steps 3, 5, ... are presented to subsystem 1 at time steps 3.5, 5.5, Consequently, the subsystems are synchronized differently in the two examples and the two systems can be expected to behave in very different ways at the micro-level.

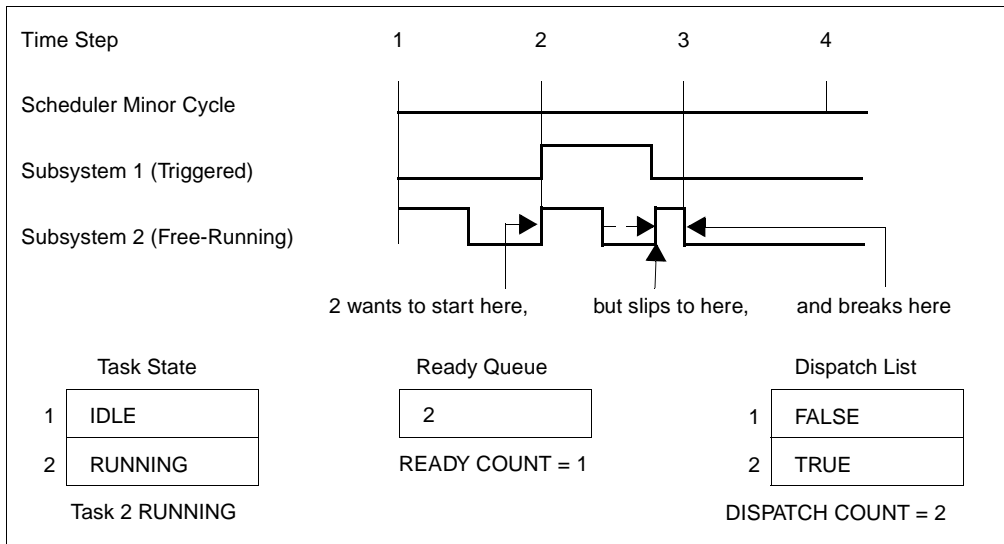
4.7 Scheduler Errors

The exact method for controlling the scheduler minor cycle interruptions is implementation-dependent. In the absence of standardization of the hardware and software for this and other functions within an embedded system, Wind River has not attempted to furnish a timing simulator, choosing instead to emphasize functional simulation. This is one major reason for our emphasizing the development of rapid prototyping or test bed systems, which can help you evaluate the performance aspects of systems where simulation cannot easily reach. However, we can postulate two kinds of timing problems that could be detected. The application scheduler traps both: scheduler overflow and subsystem overflow.

4.7.1 Scheduler or Subsystem Overflow

- Scheduler Overflow - If the non-interruptible critical section of the scheduler is running and an interruption for the scheduler occurs, then the scheduler is receiving more interrupts than it can handle. To prevent this, the length of the scheduler minor cycle must be increased, the interrupt timer rate must be decreased, or a faster processor must be obtained.
- Subsystem Overflow - Might be intermittent or rare, might or might not be a catastrophe in the context of a given system, and flexible means for dealing with it are provided. Subsystem overflow is defined as a subsystem ready to run and still not finished running. [Figure 4-23](#) shows a graphical representation of this condition.

Figure 4-23 Subsystem Overflow Example



In [Figure 4-23](#), subsystem 2 is free-running with an intermediate sampling rate. Subsystem 1, with a shorter timing requirement and therefore higher priority than subsystem 2, is triggered and runs only occasionally. When it does run, at step 2, it takes a considerable amount of time. When it finishes and subsystem 2 starts, there is not enough time for subsystem 2 to finish before the scheduler's interruption for step 3 is received. This would be acceptable, because subsystem 2 is required to be interruptible. However, at step 3, the scheduler notes that it is time for subsystem 2 to run again and enters it into the ready queue and the dispatch list. The scheduler also notes that subsystem 2 is not finished and that is where the problem begins. The scheduler cannot post sample-and-hold inputs for a subsystem that is not finished, resulting in a subsystem overflow. For some systems, the subsystem overflow is not critical.

4.7.2 Examples Where Overflow is Irrelevant or Cannot Happen

- A triggered subsystem with output posting As Soon As Finished cannot overflow.
- The background cannot overflow.

- In a subsystem where timing is not really critical, you might wish to disable the overflow indication, or to give it a slower sampling rate. Provision is made for customizing handling of the subsystem overflow error in the template files.

Most blocks in SystemBuild operate in a largely synchronous manner, executing once each time the subsystem is dispatched and contributing little to the generation of intermittent overflows. Even so, several conditions can contribute to the generation of subsystem overflows:

- A heavy load of triggered asynchronous events (see [Figure 4-23](#), p.96).
- While blocks execute a user-defined number of times in a given subsystem. If the number of iterations is variable, the amount of time to execute the subsystem becomes nondeterministic.
- If/else blocks have different logic depending on which branch is selected. If some branches have significantly more processing than others, the amount of time to execute the subsystem becomes less deterministic.
- User-supplied I/O drivers, which have variable execution time, such as a pulse-width-modulation driver that immediately returns if the duty cycle has not changed. Wind River avoids this practice in its implementation systems to the greatest degree possible.
- Your system might be over-extended. The code that is generated for SystemBuild blocks is optimized for performance, but any system can be overloaded by too many tasks doing too much work in a given cycle. Naturally, if this kind of overload occurs, the situation is likely to be catastrophic and reasonably easy to detect. But on a heavily loaded system, minor perturbations such as triggered subsystems or heavily loaded if/else constructs could cause an occasional overflow, which would be hard to debug.

5

Code Generation for Discrete Systems

This chapter introduces features of the generated code for discrete systems. This includes scheduler architecture as it relates to discrete code generation.

5.1 Introduction

A *discrete system* is a model that does not contain any Continuous SuperBlocks. The general categories are single-rate, multi-rate, and procedural discrete systems.

Single-rate discrete system — Contains SuperBlocks that use the exact same timing attributes.

Multi-rate discrete system — Contains SuperBlocks with different timing attributes.

Procedural discrete system — Contains only procedure SuperBlocks and therefore has no timing attributes.

A table describing all the options that control code generation can be found within [Appendix A, AutoCode Options](#). For more information about the structure and content of the generated code, see the *AutoCode Reference*.

5.2 How to Generate Code for Discrete Systems

The minimum options required to generate code for a discrete subsystem are: choice of language and top-level SuperBlock or real-time file (.rtf). This is shown in [Chapter 2.1 , Using AutoCode](#). Additional options are specified along with the required options. A separate options file can be used as a replacement to specify options directly to AutoCode.

5.3 Introduction to Vectorized Code

A vectorized discrete system is a discrete system that uses array variables in the generated code to implement vectors for the purposes of bundling signals together to enable loops within the generated code. The resultant vectorized code is more efficient in terms of code size and performance as compared to the nonvectorized equivalent. There is no need to generate vectorized code unless you are interested in gaining performance improvements and reducing code size on your target hardware. In other words, there is no difference in numerical results between vectorized and non-vectorized code, but *how* those results are obtained is significantly different.

By default, AutoCode generates nonvectorized code. You must specify an option to produce vectorized code (see the **-Ov** entry in [Table A-1](#), p. 161). That option controls two variations of vectorization which are summarized below.

Maximal Vectorization — This option directs AutoCode to create vectors everywhere possible. Traceability in the generated code is reduced as only one name can be used to represent many signals from the same block.

Label-based Vectorization — This option directs AutoCode to selectively create vectors for only those signals that have a vector name or label as specified in the diagram. This variation lets you specify exactly what signals are to be generated as a vector and exactly what the name of the array is within the generated code. Any signal that does not have a vector name or label will be generated as a scalar variable.

5.4 Introduction to Optimized Code

AutoCode produces generated code that is nearly one-to-one compared to the blocks used in the diagram. However, performance constraints of target hardware require optimization of the generated code. One would expect the target compiler to optimize the code, but many target compilers provide minimal optimization capabilities. Therefore, AutoCode can be directed to perform some optimizations that favor better executable code. Of course, there is a price to be paid; traceability back to the model's diagram is significantly reduced when optimized.

By default, AutoCode does not perform any special optimizations. You must specify which type of optimizations you desire. Some of the optimizations are summarized below.

Variable reuse — Reuse local variables within the code as the outputs of more than one block.

No restart — Generate code that cannot be restarted on the target unless the object code is reloaded.

Variable Block read propagation — Directly reference the Variable Block variable for read operations.

Constant propagation — Blocks that compute a constant are eliminated and the constant value is used directly.

5.5 Introduction to Procedural Code

Procedural code is the generated code for only the Procedure SuperBlocks within a model. The code is typically used for two purposes: 1) to subsequently treat the generated code as a module that is plugged into a much larger code stream; 2) the first step toward linking generated code back into the SystemBuild Simulator to improve its performance as a UserCode Block.

5.6 Sample Generated Code

The following section contains sample generated code. The code was generated with maximal vectorization and the no-restart optimization. Examples have been edited to eliminate the scheduler and other code not relevant for this example.

[Example 5-1](#) shows sample C code and [Example 5-2](#) shows sample Ada code.

5.6.1 Sample C Code

Example 5-1 SAMPLE_MODEL.c

```

/*****
|
|           AutoCode/C (TM) Code Generator V7.X
|           WIND RIVER SYSTEMS INC., SUNNYVALE, CALIFORNIA
|
*****/
rtf filename      : SAMPLE_MODEL.rtf
Filename         : SAMPLE_MODEL.c
Dac filename     : c_sim.dac
Generated on    : Mon Mar 17 18:26:36 2000
Dac file created on : Thu Mar 6 12:09:32 2000
--
--   Number of External Inputs : 4
--   Number of External Outputs: 8
--
--   Scheduler Frequency:      10.0
--
--   SUBSYSTEM  FREQUENCY  TIME_SKEW  OUTPUT_TIME  TASK_TYPE
--   - - - - -  - - - - -  - - - - -  - - - - -  - - - - -
--   1          10.0000   0.00000   0.00000     PERIODIC
*/

#include <stdio.h>
#include <math.h>
#include "sa_sys.h"
#include "sa_defn.h"
#include "sa_types.h"
#include "sa_math.h"
#include "sa_user.h"
#include "sa_utils.h"
#include "sa_time.h"
#include "sa_fuzzy.h"

/***** System Ext I/O type declarations. *****/
struct _Subsys_1_out {
    RT_FLOAT limited_values_1[4];
    RT_FLOAT limited_values_1_1[4];
};

struct _Sys_ExtIn {
```

```

    RT_FLOAT SAMPLE_MODEL_1[4];
};

/***** System Ext I/O type definitions. *****/
struct _Subsys_1_out subsys_1_out = {{-EPSILON, -EPSILON, -EPSILON, -EPSILON},
{-EPSILON, -EPSILON, -EPSILON, -EPSILON}};
struct _Sys_ExtIn sys_extin;

/***** Procedures' declarations *****/

/***** Procedure: value_added *****/

/***** Inputs type declaration. *****/
struct _value_added_u {
    RT_FLOAT gainfactor_1[4];
};

/***** Outputs type declaration. *****/
struct _value_added_y {
    RT_FLOAT limited_values_1[4];
};

/***** Info type declaration. *****/

struct _value_added_info {
    RT_INTEGER iinfo[5];
    RT_FLOAT RP[16];
};

/***** Procedures' definitions *****/

/***** Procedure: value_added *****/

void value_added(U, Y, I)
    struct _value_added_u *U;
    struct _value_added_y *Y;
    struct _value_added_info *I;
{
    RT_INTEGER *iinfo = &I->iinfo[0];

    /***** Parameters. *****/
    RT_FLOAT *R_P = &I->RP[0];

    /***** Algorithmic Local Variables. *****/
    RT_INTEGER ilower;
    RT_INTEGER iupper;
    RT_FLOAT uval;
    RT_INTEGER i;
    RT_INTEGER k;
    RT_FLOAT alpha;

```

```

/***** Output Update. *****/
/* ----- Linear Interp */
/* {value_added..2} */
for (i=1; i<=4; i++) {
    if (U->gainfactor_1[-1+i] < R_P[-2+2*i]) {
        ilower = 1;
        iupper = 0;
    }
    else if (U->gainfactor_1[-1+i] >= R_P[-1+2*i]) {
        ilower = 0;
        iupper = 1;
    }
    else {
        ilower = (RT_INTEGER)((U->gainfactor_1[-1+i] - R_P[-2+2*i])/(R_P[-1+2*i] - R_P[-2+2*i]));
        iupper = ilower + 1;
    }
    alpha = (U->gainfactor_1[-1+i] - R_P[-2+ilower+2*i])/(R_P[-2+iupper+2*i] - R_P[-2+ilower+2*i]);
    Y->limited_values_1[-1+i] = (1.0 - alpha)*R_P[6+ilower+2*i] + alpha*
    R_P[6+iupper+2*i];
}

iinfo[1] = 0;

EXEC_ERROR: return;
}

/***** Tasks declarations *****/

/***** Tasks code *****/

/***** Subsystem 1 *****/

void subsys_1(U, Y)
    struct _Sys_ExtIn *U;
    struct _Subsys_1_out *Y;
{
    static RT_INTEGER iinfo[4] = {0, 1, 1, 1};

    /***** Parameters. *****/
    static RT_FLOAT R_P[8] = {4.3, 5.2, 3.5, 2.3, -4.3, -5.2, -3.5, -2.3};

    /***** Local Block Outputs. *****/

    RT_FLOAT gainfactor_1[4];
    RT_FLOAT inverse_factor_1[4];

    /***** Algorithmic Local Variables. *****/

    RT_INTEGER i;
    static struct _value_added_u value_added_4_u;
    static struct _value_added_y value_added_4_y;
    static struct _value_added_info value_added_4_i = {{0, 1, 1, 1, 1},

```

```

    {-10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -1.5, 1.5, -1.5,
     1.5, -1.5, 1.5, -1.5, 1.5}}};
static struct _value_added_u value_added_14_u;
static struct _value_added_y value_added_14_y;
static struct _value_added_info value_added_14_i = {{0, 1, 1, 1, 1},
    {-10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -1.5, 1.5, -1.5,
     1.5, -1.5, 1.5, -1.5, 1.5}}};

/***** Output Update. *****/
/* ----- Gain Block */
/* {SAMPLE_MODEL.gf1.1} */
for (i=1; i<=4; i++) {
    gainfactor_1[-1+i] = R_P[-1+i]*U->SAMPLE_MODEL_1[-1+i];
}

/* ----- Procedure SuperBlock */
/* {value_added.4} */
{
    RT_INTEGER k = 0;
    for( k=0;k<4;k++ ) {
        value_added_4_u.gainfactor_1[k] = gainfactor_1[k];
    }
}
value_added(&value_added_4_u, &value_added_4_y, &value_added_4_i);
{
    RT_INTEGER k = 0;
    for( k=0;k<4;k++ ) {
        Y->limited_values_1[k] = value_added_4_y.limited_values_1[k];
    }
}
iinfo[0] = value_added_4_i.iinfo[0];
if( iinfo[0] != 0 ) {
    value_added_4_i.iinfo[0] = 0; goto EXEC_ERROR;
}
/* ----- Gain Block */
/* {SAMPLE_MODEL.gf2.2} */
for (i=1; i<=4; i++) {
    inverse_factor_1[-1+i] = R_P[3+i]*U->SAMPLE_MODEL_1[-1+i];
}
/* ----- Procedure SuperBlock */
/* {value_added.14} */
{
    RT_INTEGER k = 0;
    for( k=0;k<4;k++ ) {
        value_added_14_u.gainfactor_1[k] = inverse_factor_1[k];
    }
}
value_added(&value_added_14_u, &value_added_14_y, &value_added_14_i);
{
    RT_INTEGER k = 0;
    for( k=0;k<4;k++ ) {
        Y->limited_values_1_1[k] = value_added_14_y.limited_values_1[k];
    }
}
iinfo[0] = value_added_14_i.iinfo[0];
if( iinfo[0] != 0 ) {

```

```

    value_added_14_i.iinfo[0] = 0; goto EXEC_ERROR;
}

if(iinfo[1]) {
    SUBSYS_INIT[1] = FALSE;
    iinfo[1] = 0;
}
return;

EXEC_ERROR: ERROR_FLAG[1] = iinfo[0];
    iinfo[0]=0;
}

```

5.6.2 Sample Ada Code

Example 5-2 SAMPLE_MODEL.a

```

-----
--          AutoCode/Ada (TM) Code Generator V7.X          -
--          WIND RIVER SYSTEMS INC., SUNNYVALE, CALIFORNIA  -
-----
-- rtf filename      : SAMPLE_MODEL.rtf
-- Filename          : SAMPLE_MODEL.a
-- Dac filename      : ada_rt.dac
-- Generated on      : Mon Mar 17 18:27:44 2000
-- Dac file created on : Mon Mar 10 17:03:32 2000
--
-- Number of External Inputs : 4
-- Number of External Outputs: 8
--
-- Scheduler Frequency:    10.0
--
-- SUBSYSTEM  FREQUENCY  TIME_SKEW  OUTPUT_TIME  TASK_TYPE
-- -----
-- 1          10.0000    0.00000   0.00000      PERIODIC
-----

-----
--- System Data ---
-----

with SYSTEM;
with UNCHECKED_CONVERSION;
with SA_TYPES;           use SA_TYPES;
with SA_DEFN;           use SA_DEFN;
with SA_TIME;           use SA_TIME;
package SYSTEM_DATA is
    NUMIN      : constant RT_INTEGER := 4;
    NUMOUT     : constant RT_INTEGER := 8;
    ExtIn      : RT_FLOAT_AY(0..NUMIN);
    ExtOut     : RT_FLOAT_AY(0..NUMOUT) := (others => -EPSILON);
    SUBSYS_PREINIT : RT_BOOLEAN_AY(0..NTASKS);

----- System Ext I/O type declarations. -----

```

```

type Subsys_1_out_t is record
  limited_values_1 : RT_FLOAT_AY(0..3);
  limited_values_1_1 : RT_FLOAT_AY(0..3);
end record;

type Sys_ExtIn_t is record
  SAMPLE_MODEL_1 : RT_FLOAT_AY(0..3);
end record;

----- System Ext I/O type definitions. -----
subsys_1_out : Subsys_1_out_t := ((-EPSILON, -EPSILON, -EPSILON, -EPSILON),
  (-EPSILON, -EPSILON, -EPSILON, -EPSILON));
sys_extin : Sys_ExtIn_t;

end SYSTEM_DATA;

----- Procedures package declarations -----
with SYSTEM;
with UNCHECKED_CONVERSION;
with SA_TYPES;
with SYSTEM_DATA;
package value_added_pkg is
  use SA_TYPES;
  use SYSTEM_DATA;

  ----- Inputs type declaration. -----
  type value_added_u_t is record
    gainfactor_1 : RT_FLOAT_AY(0..3);
  end record;

  ----- Outputs type declaration. -----
  type value_added_y_t is record
    limited_values_1 : RT_FLOAT_AY(0..3);
  end record;

  ----- Info type declaration. -----
  type value_added_info_t is record
    iinfo : RT_INTEGER_AY(0..4);
    RP : RT_FLOAT_AY(0..15);
  end record;

  type value_added_u_t_P is access value_added_u_t;
  function ptr_of is new UNCHECKED_CONVERSION
    (SOURCE => SYSTEM.ADDRESS, TARGET => value_added_u_t_P);
  type value_added_y_t_P is access value_added_y_t;
  function ptr_of is new UNCHECKED_CONVERSION
    (SOURCE => SYSTEM.ADDRESS, TARGET => value_added_y_t_P);
  type value_added_info_t_P is access value_added_info_t;
  function ptr_of is new UNCHECKED_CONVERSION
    (SOURCE => SYSTEM.ADDRESS, TARGET => value_added_info_t_P);

  ----- Procedure: value_added -----
  procedure value_added(U : value_added_u_t_P;
    Y : value_added_y_t_P;
    I : value_added_info_t_P
  );
end value_added_pkg;

```

```

----- Subsystems' declarations -----
with SYSTEM;
with UNCHECKED_CONVERSION;
with SA_TYPES;                               use SA_TYPES;
with SYSTEM_DATA;                             use SYSTEM_DATA;
with value_added_pkg;                         use value_added_pkg;
package SUBSYSTEMS is

    ----- Subsystem 1 Package -----
    package subsys_1_pkg is
        type Sys_ExtIn_t_P is access Sys_ExtIn_t;
        function ptr_of is new UNCHECKED_CONVERSION
            (SOURCE => SYSTEM.ADDRESS, TARGET => Sys_ExtIn_t_P);
        type Subsys_1_out_t_P is access Subsys_1_out_t;
        function ptr_of is new UNCHECKED_CONVERSION
            (SOURCE => SYSTEM.ADDRESS, TARGET => Subsys_1_out_t_P);
        U : Sys_ExtIn_t_P := ptr_of(sys_extin'address);
        Y : Subsys_1_out_t_P := ptr_of(subsys_1_out'address);

        procedure subsys_1;
    end subsys_1_pkg;
end SUBSYSTEMS;

with SA_DEFN;                               use SA_DEFN;
with SA_TYPES;                               use SA_TYPES;
with SYSTEM_DATA;                             use SYSTEM_DATA;
package body SUBSYSTEMS is
    package body subsys_1_pkg is separate;
end SUBSYSTEMS;

with SA_TYPES;                               use SA_TYPES;
with SYSTEM_DATA;                             use SYSTEM_DATA;
with SA_UTILITIES;                           use SA_UTILITIES;
separate (SUBSYSTEMS)
package body subsys_1_pkg is

    SUBSYS_ID : constant := 1;

    ----- Tasks code -----
    iinfo : RT_INTEGER_AY(0..3) := (0, 1, 1, 1);

    ----- Parameters. -----
    R_P : RT_FLOAT_AY(0..7) := (4.3, 5.2, 3.5, 2.3, -4.3, -5.2, -3.5, -2.3);
    value_added_4_u : value_added_u_t;
    value_added_4_y : value_added_y_t;
    value_added_4_i : value_added_info_t := ((0, 1, 1, 1, 1), (-10.5, 20.5,
        -10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -1.5, 1.5, -1.5, 1.5, -1.5, 1.5,
        -1.5, 1.5));
    value_added_14_u : value_added_u_t;
    value_added_14_y : value_added_y_t;
    value_added_14_i : value_added_info_t := ((0, 1, 1, 1, 1), (-10.5, 20.5,
        -10.5, 20.5, -10.5, 20.5, -10.5, 20.5, -1.5, 1.5, -1.5, 1.5, -1.5, 1.5,
        -1.5, 1.5));

    procedure subsys_1 is

```

```

----- Local Block Outputs. -----
gainfactor_1 : RT_FLOAT_AY(0..3);
inverse_factor_1 : RT_FLOAT_AY(0..3);

----- Algorithmic Local Variables. -----
i_2 : RT_INTEGER;

begin
----- Output Update. -----
-- ----- Gain Block --
-- {SAMPLE_MODEL.gf1.1} --
for i_2 in RT_INTEGER range 1..4 loop
    gainfactor_1(-1+i_2) := R_P(-1+i_2)*U.SAMPLE_MODEL_1(-1+i_2);
end loop;
-- ----- Procedure Super Block --
-- {value_added.4} --
value_added_4_u.gainfactor_1(0..3) := gainfactor_1(0..3);
value_added(ptr_of(value_added_4_u'address), ptr_of(
    value_added_4_y'address), ptr_of(value_added_4_i'address));
Y.limited_values_1(0..3) := value_added_4_y.limited_values_1(0..3);
iinfo(0) := value_added_4_i.iinfo(0);
if iinfo(0) /= 0 then
    value_added_4_i.iinfo(0) := 0; raise EXEC_ERROR;
end if;
-- ----- Gain Block --
-- {SAMPLE_MODEL.gf2.2} --
for i_2 in RT_INTEGER range 1..4 loop
    inverse_factor_1(-1+i_2) := R_P(3+i_2)*U.SAMPLE_MODEL_1(-1+i_2);
end loop;
-- ----- Procedure Super Block --
-- {value_added.14} --
value_added_14_u.gainfactor_1(0..3) := inverse_factor_1(0..3);
value_added(ptr_of(value_added_14_u'address), ptr_of(
    value_added_14_y'address), ptr_of(value_added_14_i'address));
Y.limited_values_1_1(0..3) := value_added_14_y.limited_values_1(0..3);
iinfo(0) := value_added_14_i.iinfo(0);
if iinfo(0) /= 0 then
    value_added_14_i.iinfo(0) := 0; raise EXEC_ERROR;
end if;

if iinfo(1) > 0 then
    iinfo(1) := 0;
    SUBSYS_INIT(1) := false;
end if;

exception
    when EXEC_ERROR =>
        ERROR_FLAG(1) := iinfo(0); iinfo(0) := 0;
    when NUMERIC_ERROR | CONSTRAINT_ERROR =>
        ERROR_FLAG(1) := MATH_ERROR;
    when OTHERS =>
        ERROR_FLAG(1) := UNKNOWN_ERROR;
end subsys_1;
end subsys_1_pkg;

----- Procedures package bodies -----

```

```

with SA_TYPES;
with SA_DEFN;
with SYSTEM_DATA;
package body value_added_pkg is
----- Procedure: value_added -----
  procedure value_added(U : value_added_u_t_P;
    Y : value_added_y_t_P;
    I : value_added_info_t_P
  ) is
    iinfo : RT_INTEGER_AY_5_P := ptr_of(I.iinfo'address);

    ----- Parameters. -----
    R_P : RT_FLOAT_AY_16_P := ptr_of(I.RP'address);

    ----- Algorithmic Local Variables. -----
    ilower : RT_INTEGER;
    iupper : RT_INTEGER;
    uval : RT_FLOAT;
    i_1 : RT_INTEGER;
    k_1 : RT_INTEGER;
    alpha_1 : RT_FLOAT;

begin
  ----- Output Update. -----
  -- ----- Linear Interp -----
  -- {value_added..2} --
  for i_1 in RT_INTEGER range 1..4 loop
    if U.gainfactor_1(-1+i_1) < R_P(-2+2*i_1) then
      ilower := 1;
      iupper := 0;
    elsif U.gainfactor_1(-1+i_1) >= R_P(-1+2*i_1) then
      ilower := 0;
      iupper := 1;
    else
      ilower := ITRUNCATE((U.gainfactor_1(-1+i_1) - R_P(-2+2*i_1))/(R_P(
-1+2*i_1) - R_P(-2+2*i_1)));
      iupper := ilower + 1;
    end if;
    alpha_1 := (U.gainfactor_1(-1+i_1) - R_P(-2+ilower+2*i_1))/(R_P(
-2+iupper+2*i_1) - R_P(-2+ilower+2*i_1));
    Y.limited_values_1(-1+i_1) := (1.0 - alpha_1)*R_P(6+ilower+2*i_1) +
alpha_1*R_P(6+iupper+2*i_1);
  end loop;

  iinfo(1) := 0;

exception
  when EXEC_ERROR =>
    null;
  when NUMERIC_ERROR | CONSTRAINT_ERROR =>
    iinfo(0) := MATH_ERROR;
  when OTHERS =>
    iinfo(0) := UNKNOWN_ERROR;
end value_added;
end value_added_pkg;

```

6

Code Generation for Continuous Systems

This chapter discusses the scheduler architecture as it relates to continuous code generation. Topics include fixed-step integrators, user-defined integrators, and how to generate code for continuous and hybrid systems.

6.1 Introduction

AutoCode supports code generation for continuous or hybrid (continuous and discrete) systems. The AutoCode scheduler supports continuous subsystems in the same manner in which it supports discrete subsystems.

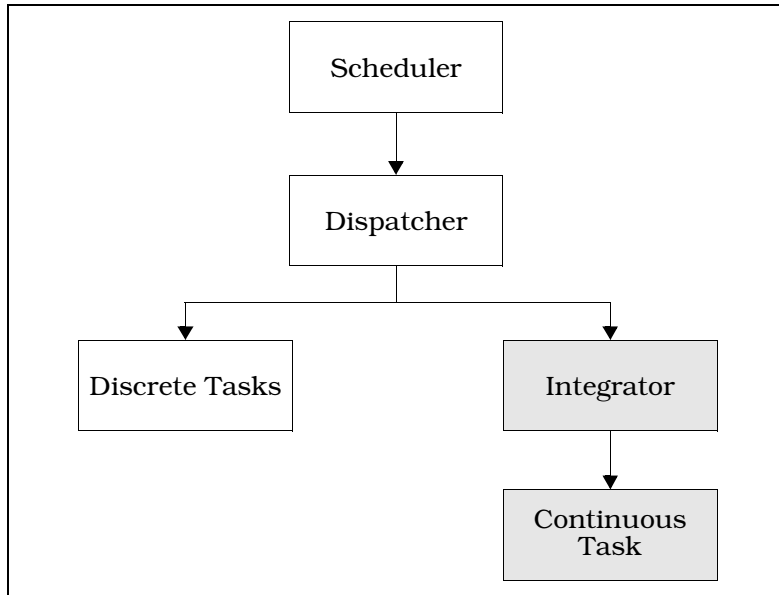
For continuous subsystems, at each minor cycle, the scheduler:

- Schedules the continuous subsystem to run.
- Posts continuous subsystem outputs.
- Performs sample and hold on the continuous subsystem inputs.
- Dispatches the continuous subsystem if ready.
- Handles vectorization and optimization the same as for discrete systems.

An element of the scheduler is the Integrator (see [Figure 6-1](#), p.112). It performs continuous, fixed-step integration of states and implicitly dispatches the continuous subsystem to perform the state and output updates. The integrator/

continuous task pair is, by default, treated as the fastest task to be dispatched by the scheduler.

Figure 6-1 Scheduler Architecture



6.2 Integrators

AutoCode supplies four fixed-step integrators:

- First order Runge-Kutta (Euler)
- Second order Runge-Kutta (Modified Euler)
- Fourth order Runge-Kutta (Simpson's 2nd rule)
- Kutta-Merson

All of these integrators are located in the templates directory in the integrator template file *language_intgr.tpl*. There is also the capability to insert a user-supplied integrator. Instructions for using your own integrator are provided in *6.4.2 Xmath Command Options for Continuous Code Generation*, p. 114.

6.3 Limitations

When using continuous code generation, keep these limitations in mind:

- Only fixed-step integrators are supported.
- There is a slight mismatch of sim and continuous application outputs (i.e., the subsystem external inputs at **time t** and at **time t+h**, where **h** is the integration step, are assumed to be unchanged inside AutoCode integrator algorithms).
- Continuous task states and derivatives are always of float data type.
- Algebraic loops are not supported.
- Only sim initialization mode 0 (initmode 0; see sim help for details) is supported.
- You cannot generate procedures-only continuous code (procedure around a top-level continuous hierarchy).

6.4 How to Generate Code for Continuous or Hybrid Systems

As described in 2.1 *How to Generate Real-Time Code*, p. 15, using AutoCode, you can generate C high-level language code from SystemBuild, the Xmath Commands window, or from the operating system prompt. The subsections that follow discuss each of these methods of code generation in terms of those options that are unique to generating code for continuous or hybrid systems.

You need both `c_sim.tpl` and `c_intgr.tpl` template files for C or `ada_rt.tpl` and `ada_intgr.tpl` for Ada (supplied in the `templates` directory). The `c_sim.tpl` and `ada_rt.tpl` template files include continuous subsystems-related parameters and the integrator template file. The integrator template file contains the code for the four integrators and a stubbed routine, `usrintegrator`, which provides the means for user-defined integrator implementation.

6.4.1 Generating Code for Continuous Systems from SystemBuild

To use AutoCode while inside SystemBuild, select Tools→AutoCode on the Catalog Browser to open the dialog. Instructions for using this dialog are in the MATRIX_x Help.

Depending on the template file used, the code generated can be either C code or Ada code.

6.4.2 Xmath Command Options for Continuous Code Generation

The method for generating code for a continuous or hybrid system using the Xmath command line follows the procedure described in 2.1.2 *Generating Code from Xmath*, p.16. Two command line options that are unique to continuous code generation are **ialg** and **csi**.¹ Although not for exclusive use in continuous code generation, the **minsf** option is useful for increasing the rate of a continuous task. See [Table 6-1](#) for a summary of these options.

Table 6-1 **Xmath Command Options for Continuous Code Generation**

Option	Description
ialg	Specifies the integrator selection 0 = user-defined integrator 1 = first order Runge-Kutta integrator 2 = second order Runge-Kutta integrator (default) 3 = fourth order Runge-Kutta integrator 4 = Kutta-Merson integrator
csi	Specifies the continuous task sample interval
minsf	Specifies the minimum AutoCode scheduler frequency in seconds (0.0 is the default)

As indicated in [Table 6-1](#), **ialg** specifies the selected integrator. The option takes an integer argument of 0, 1, 2, 3, or 4. The default integrator is the second order Runge-Kutta.

1. For standalone AutoCode, results for generated code will not match sim unless the **csi** option is not zero. Typically, set **csi** to 0.01, the time vector for standalone sim. Then, results will match.

When using 0 (user-defined integrator) for this command-line option, the integrator function should be implemented inside the function `usrintegrator()` located in `c_intgr.tpl` for C or `ada_intgr.tpl` for Ada.

Because the integrator is invoked at each scheduler interval and the continuous task is dispatched via the integrator, an implicit frequency (that of the scheduler) is associated with the continuous task. If the system is all continuous, the scheduler cycle is 1 Hz. For hybrid systems, the implicit frequency of the continuous task is always the least common multiple of all of the frequencies of the discrete tasks. For continuous only modes, the implicit frequency of the single continuous defaults to 1 Hz. The command option `csi` specifies the sample interval for a continuous task. This option is useful for adjusting the rate of the continuous task.

The command option `minsf` specifies the minimum AutoCode scheduler frequency. This option is useful for increasing the rate of a continuous task. The real-time scheduler frequency is set to the larger value of the frequency determined by the block diagram application and the value specified by the `minsf` option. The default value for this option is 0.0, which allows the application to set its own scheduler frequency. Deviation from this default should be approached with caution, as a consistent scheduler frequency should normally be based on a least common multiple of the application timing requirements.

For example, to generate code for a model with a continuous subsystem, using the fourth order Runge-Kutta integrator method, and minimum scheduler frequency of 300.0 Hz in file `model.c`, use the following Xmath command:

```
autocode, model="model", {ialg=3, minsf=300.0}
```

In this case, the `autocode` command automatically generates the file `model.c` in the directory from which Xmath was invoked.

6.4.3 OS Command Options for Continuous Code Generation

The method for generating code for a continuous or hybrid system using the operating system command line follows the procedure described in 2.1.3 *Generating Code from the Operating System*, p.17. Two command options that are unique to continuous code generation are `-i` and `-csi`. Although not for

exclusive use in continuous code generation, the **-minsf** option may be useful for increasing the rate of a continuous task. [Table 6-2](#) summarizes these options.

Table 6-2 **Operating System Command Options for Continuous Code Generation**

Option	Description
-i	Specifies the integrator selection 0 = user-defined integrator 1 = first order Runge-Kutta integrator 2 = second order Runge-Kutta integrator (default) 3 = fourth order Runge-Kutta integrator 4 = Kutta-Merson integrator
-csi	Specifies the continuous task sample interval
-minsf	Specifies the minimum AutoCode scheduler frequency in seconds (0.0 is the default)

As indicated in [Table 6-2](#), **-i** specifies the selected integrator. The option takes an integer argument of 0, 1, 2, 3, or 4. The default integrator is the second order Runge-Kutta.

When using 0 (user-defined integrator) for this command option, the integrator function should be implemented inside the function **usrintegrator()** located in **c_intgr.tpl** for C or **ada_intgr.tpl** for Ada.

Because the integrator is invoked at each scheduler interval and the continuous task is dispatched via the integrator, an implicit frequency (that of the scheduler) is associated with the continuous task. For hybrid systems, the implicit frequency of the continuous task is always the least common multiple of all the frequencies of the discrete tasks. The command option **-csi** specifies the sample interval for a continuous task.

The command option **-minsf** specifies the minimum AutoCode scheduler frequency. This option is useful for increasing the rate of a continuous task. The real-time scheduler frequency is set to the larger value of the frequency determined by the block diagram application and the value specified by the **minsf** option. The default value for this option is **0.0**, which allows the application to set its own scheduler frequency. Deviation from this default should be approached with caution, as a consistent scheduler frequency should normally be based on a least common multiple of the application timing requirements.

To generate code for a model with a continuous subsystem, using the fourth order Runge-Kutta integrator method, and minimum scheduler frequency of 300.0 Hz, use the operating system command shown in [Example 6-1 \(C\)](#) or [Example 6-2 \(Ada\)](#):

Example 6-1 **Sample Operating System Command for C**

```
% autostar -l c -i 3 -minsf 300.0 -o model.c model.rtf
```

Example 6-2 **Sample Operating System Command for Ada**

```
% autostar -l a -i 3 -minsf 300.0 -o model.a model.rtf
```

6

6.5 Sample Generated C Code

The following example is a file that lists the generated model and default integrator (Runge-Kutta 2) code for the block diagram model located in the file `mws_demo.dat` in the `classical_demo` directory located in the SystemBuild `demo` distribution directory. The block diagram is shown in [Figure 6-2](#).

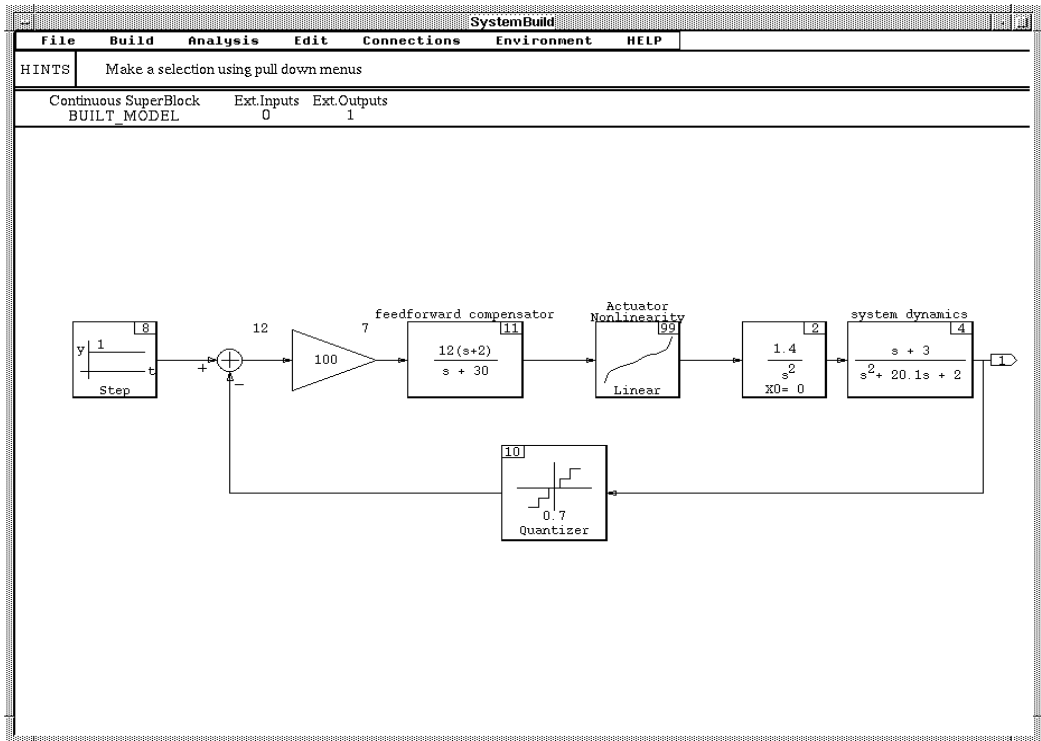
The sample generated code ([Example 6-3](#)) has been edited for brevity, showing only the most important features.

As code can change slightly from one release to the next, please refer to the current example in your demo directory for an exact code listing.



NOTE: If you need to review the steps required to create an executable, refer to [2.5.1 Standalone Simulation](#), p.26.

Figure 6-2 Built_Model SuperBlock



Example 6-3 File built_model.c

```

/*****
| AutoGen/C (TM) Code Generator V7.x |
| WIND RIVER SYSTEMS INC., SUNNYVALE, CALIFORNIA |
*****/
Modelname : built_model
Filename : built_model.c
Generated on : Wed Aug 4 16:43:28 1999
Dac file created on : Tue Jul 27 20:48:17 1999
*/

#include <stdio.h>
#include "sa_intgr.h"
...
/**** System Data ****/

#define SCHEDULER_FREQ 300.0
#define NTASKS 1
#define NUMIN 0

```

```

#define NUMOUT 1
#define IALG 2

enum TASK_STATE_TYPE { IDLE, RUNNING, BLOCKED, UNALLOCATED };

enum SUBSYSTEM_TYPE { CONTINUOUS, PERIODIC, ENABLED_PERIODIC,
                     TRIGGERED_ANT, TRIGGERED_ATR, TRIGGERED_SAF, NONE };
...
/***** Global declarations. *****/
...
/***** System Ext I/O structs. declarations.*****/
...
/***** System Ext I/O structs. definitions.*****/
...
/**Continuous Subsystem states and info structs. declarations.**/
struct _Subsys_1_states {
    RT_FLOAT system_dynamics_S1;
    RT_FLOAT system_dynamics_S2;
    RT_FLOAT BUILT_MODEL_2_S1;
    RT_FLOAT BUILT_MODEL_2_S2;
    RT_FLOAT feedforward_compensator_S1;
};
struct _Subsys_1_info {
    RT_INTEGER iinfo[5];
    RT_FLOAT rinfo[5];
};

/**Continuous Subsystem states and info structs. definitions.**/struct
_Subsys_1_states subsys_1_states[2] = {0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0.};

struct _Subsys_1_info subsys_1_info = {0, 1, 1, 1, 0, 1., 0.};
...
/***** Task's declarations. *****/

/***** (Continuous) Subsystem 1 *****/
extern void subsys_1();

/***** Task's code. *****/

/***** (Continuous) Subsystem 1 *****/
void subsys_1(Y, S, I)
    struct _Subsys_1_out *Y;
    struct _Subsys_1_states *S;
    struct _Subsys_1_info *I;
{
    RT_INTEGER *iinfo = &I->iinfo[0];
    RT_FLOAT *rinfo = &I->rinfo[0];
    RT_INTEGER INIT = iinfo[1];
    RT_INTEGER STATES = iinfo[2];
    RT_INTEGER OUTPUTS = iinfo[3];
    RT_INTEGER CALLER = iinfo[4];
    const RT_DURATION TIME = rinfo[0];

    /**** Current and Next States Pointers. ****/

```

```

struct _Subsys_1_states *X = &S[0];
struct _Subsys_1_states *XD = &S[1];

/***** Parameters. *****/
...

/***** Local Block Outputs. *****/
...

if(OUTPUTS) { /* Output Update. */
/*----- Num - Den Coeffs. */
/* {BUILT_MODEL.system dynamics.4} */
Y->system_output = 0.5*X->system_dynamics_S1;
Y->system_output = Y->system_output + 1.5*X->system_dynamics_S2;
/*----- Nth Order Integrator */
/* {BUILT_MODEL..2} */
BUILT_MODEL_2_1 = 1.4*X->BUILT_MODEL_2_S1;
/*----- Step Function */
BUILT_MODEL_8_1= 1;
/*----- Quantization */
/* {BUILT_MODEL..10} */
BUILT_MODEL_10_1 = 0.7*ROUND(Y->system_output/0.7);
sgn = SGN(Y->system_output);
ushift = fabs(Y->system_output) + 0.35;
remain = fmod(ushift,0.7);
alpha = remain - (1.0 - RELTOL)*0.7;
if (alpha > 0.0) {
    BUILT_MODEL_10_1 = BUILT_MODEL_10_1 + sgn*alpha/RELTOL;
}
...
}
if(STATES) { /* State Update. */
/*----- Num - Den Coeffs. */
/* {BUILT_MODEL.system dynamics.4} */
XD->system_dynamics_S1 = 0.0;
XD->system_dynamics_S2 = 0.0;
XD->system_dynamics_S1 = XD->system_dynamics_S1 - 20.1
*X->system_dynamics_S1;
XD->system_dynamics_S1 = XD->system_dynamics_S1 - 2.0
*X->system_dynamics_S2;
XD->system_dynamics_S2 = XD->system_dynamics_S2 +
    X->system_dynamics_S1;
XD->system_dynamics_S1 = XD->system_dynamics_S1 +
    2.0*BUILT_MODEL_2_1;
/*----- Nth Order Integrator */
/* {BUILT_MODEL..2} */
XD->BUILT_MODEL_2_S1 = X->BUILT_MODEL_2_S2;
XD->BUILT_MODEL_2_S2 = Actuator_Nonlinearity_1;
...
}
INIT = 0;
iinfo[1] = 0;
return;
ERROR: ERROR_FLAG[1] = iinfo[0];
iinfo[0]=0;
}

```

```

/* The function rungekutta2 employs the second-order Runge-Kutta method with
Kutta's coefficients to integrate a system of n simultaneous first order
ordinary differential equations  $dxdt[j] = dx[j]/dt$ , ( $j=1,2,\dots,n$ ), across one
step of length h in the independent variable t, subject to initial conditions
 $x[j]$ , ( $j=1,2,\dots,n$ ). Each  $dxdt[j]$ , the derivative of  $x[j]$ , must be computed
two times per integration step by calling the state derivatives and output
equations function (sybsys_1()). savex(j) is used to save the initial value
of  $x(j)$  and phi(j) is the increment function for the j(th) equation. As
written, n may be no larger than 5. (Modified Euler)
*/
void rungekutta2(n,x,dxdt,t,h)
    RT_INTEGER n;
    RT_FLOAT *x,*dxdt,t,h;
{
    RT_FLOAT phi[5];
    RT_FLOAT savex[5];
    RT_INTEGER j, retval;
    RT_FLOAT hh = t;

    ss1_rinfo[0] = hh; ss1_rinfo[1] = 0.0;
    ss1_iinfo[2]=1; ss1_iinfo[3]=1; ss1_iinfo[4]=8;
    subsys_1(&subsys_1_out, subsys_1_states, &subsys_1_info);

    for (j=0; j<n; j++) {
        savex[j] = x[j];
        phi[j] = dxdt[j];
        x[j] = savex[j] + h*dxdt[j];
    }
    hh=t+h;

    ss1_rinfo[0] = hh; ss1_rinfo[1] = 0.0;
    ss1_iinfo[2]=1; ss1_iinfo[3]=1; ss1_iinfo[4]=0;
    subsys_1(&subsys_1_out, subsys_1_states, &subsys_1_info);

    for (j=0; j<n; j++) x[j] = savex[j] + (phi[j] + dxdt[j])*h/2.0;
}

/*-----*
*-- SCHEDULER --*
*-----*/
...
void Init_Scheduler()
{
    ...
}
void SCHEDULER()
{

    register RT_INTEGER NTSK;
    register RT_INTEGER J;
    RT_INTEGER ITSK;
    RT_INTEGER I;

    TIME_COUNT = TIME_COUNT + 1;
}

```

```

  /*** System Input ***/
  ...
  /*** Task Scheduling ***/

  for( NTSK=NTASKS; NTSK>=1; NTSK-- ){

    switch( TASK_STATE[NTSK] ){
      case IDLE :

        switch( TCB[NTSK].TASK_TYPE ){
          case CONTINUOUS :
          case PERIODIC :
            if( TCB[NTSK].START == 0 ){
              Queue_Task(NTSK);
              Update_Outputs(NTSK);
              TCB[NTSK].START =
TCB[NTSK].SCHEDULING_COUNT;
            }else{
              TCB[NTSK].START =
TCB[NTSK].START - 1;
            }
            break;

          case ENABLED_PERIODIC :
            ...
            break;

          case RUNNING :
            ...
        }
      }

    /*** System Output ***/
    ...
    /*** Update elapsed time ***/

    ELAPSED_TIME = ((RT_DURATION)TIME_COUNT)*SCHEDULER_INTERVAL;

    /*** Task Input Sample and Hold ***/
    ...

    /*** Signal End of Critical Section ***/
    ...

    /*** Task Dispatching ***/

    while( ITSK < CURRENT_PRIORITY && ITSK <= DISPATCH_COUNT ){
      Disable;
      if( DISPATCH[ITSK] ){
        LEVEL++;
        PRIORITY[LEVEL] = CURRENT_PRIORITY;
        CURRENT_PRIORITY = ITSK;
        DISPATCH[ITSK] = FALSE;
        Enable;
        switch (ITSK){

```

```

&subsys_1_states,
                                case 1:
subsys_1 (&subsys_1_out,
                                &subsys_1_info);
                                rungekutta2(5,
(RT_FLOAT *)(&subsys_1_states[0]),
(RT_FLOAT *)(&subsys_1_states[1]),
(RT_FLOAT)SUBSYS_TIME[1],
                                0.003);

                                break;
                                default : break;
}
...
}
..
}

```

6.6 Sample Generated Ada Code

The following example is a file that lists the generated model and default integrator (Runge-Kutta 2) code for the block diagram model located in the file **mws_demo.dat** in the **classical_demo** directory located in the SystemBuild **demo** distribution directory. The block diagram is shown in [Figure 6-2](#), p.118.

AutoCode automatically generates **built_model.a** in the directory from which Xmath was invoked. The sample generated code ([Example 6-4](#)) has been edited for brevity, showing only the most important features.

As code can change slightly from one release to the next, be sure to refer to the current example in your demo directory for an exact code listing.



NOTE: If you need to review the steps required to create an executable, refer to [2.5.1 Standalone Simulation](#), p.26.

Example 6-4 **File built_model.a**

```

-----
--                               AutoCode/Ada (TM) Code Generator V7.x
--                               WIND RIVER SYSTEMS INC., SUNNYVALE, CALIFORNIA
-----
Modelname           : built_model
-- Filename         : built_model.ada
-- Dac filename     : ada_rt.dac

```

```

-- Generated on           : Wed Dec  1 20:59:39 1999
-- Dac file created on   : Wed Dec  1 18:01:31 1999
-----...
package SUBSYSTEMS is

  ----- (Continuous) Subsystem 1 Package -----
  package subsys_1_pkg is
    ...

    procedure subsys_1;

    procedure rungekutta2(n :in RT_INTEGER;
                          x :in out RT_FLOAT_AY_5_P;
                          dxdt :RT_FLOAT_AY_5_P;
                          t :in RT_FLOAT;
                          h :in RT_FLOAT);

  end subsys_1_pkg;

end SUBSYSTEMS;

...
package body Subsys_1_pkg is

  SUBSYS_ID : constant := 1;

  ----- Task's code. -----
  ...
  procedure subsys_1 is
    ...
    ----- Local Block Outputs. -----
    ...
    ----- Algorithmic Local Variables. -----
    ...
  begin
    if iinfo(1) > 0 then
      INIT := TRUE; iinfo(1) := 0;
    end if;
    if iinfo(2) > 0 then
      STATES := TRUE; iinfo(2) := 0;
    end if;
    if iinfo(3) > 0 then
      OUTPUTS := TRUE; iinfo(3) := 0;
    end if;

    ----- Output Update. -----

    if OUTPUTS then
      -- ----- Num - Den Coeffs. --
      -- {BUILT_MODEL.system_dynamics.4} --
      Y.system_output := 0.5*X.system_dynamics_S1 + 1.5*
        X.system_dynamics_S2;

      -- ----- Nth Order Integrator --
      -- {BUILT_MODEL..2} --
      BUILT_MODEL_2_1 := 1.4*X.BUILT_MODEL_2_S1;
      -- ----- Step Function --

```

```

-- {BUILT_MODEL..8} --
      BUILT_MODEL_2_1:=1;
-- ----- Quantization --
-- {BUILT_MODEL..10} --
BUILT_MODEL_10_1 := 0.7*ROUND(Y.system_output/0.7);
sgn := SGN(Y.system_output);
ushift := ABS(Y.system_output) + 0.35;
remain := ((ushift)MOD(0.7));
alpha_1 := remain - (1.0 - RELTOL)*0.7;
if alpha_1 > 0.0 then
  BUILT_MODEL_10_1 := BUILT_MODEL_10_1 + sgn*alpha_1/RELTOL;
end if;
...

end if;

----- State Update. -----
if STATES then
  -- ----- Num - Den Coeffs. --
  -- {BUILT_MODEL.system dynamics.4} --
  XD.system_dynamics_S1 := 0.0;
  XD.system_dynamics_S2 := 0.0;
  XD.system_dynamics_S1 := XD.system_dynamics_S1 - 20.1*
    X.system_dynamics_S1;
  XD.system_dynamics_S1 := XD.system_dynamics_S1 - 2.0*
    X.system_dynamics_S2;
  XD.system_dynamics_S2 := XD.system_dynamics_S2 +
    X.system_dynamics_S1;
  XD.system_dynamics_S1 := XD.system_dynamics_S1 + 2.0*
    BUILT_MODEL_2_1;
  -- ----- Nth Order Integrator --
  -- {BUILT_MODEL..2} --
  XD.BUILT_MODEL_2_S1 := X.BUILT_MODEL_2_S2;
  XD.BUILT_MODEL_2_S2 := Actuator_Nonlinearity_1;
  ...
end if;

INIT := FALSE;
exception
when EXEC_ERROR =>
  ERROR_FLAG(1) := iinfo(0); iinfo(0) := 0;
when NUMERIC_ERROR | CONSTRAINT_ERROR =>
  ERROR_FLAG(1) := MATH_ERROR;
when OTHERS =>
  ERROR_FLAG(1) := UNKNOWN_ERROR;
end subsys_1;

-- The function rungekutta2 employs the second-order Runge-Kutta method
-- with Kutta's coefficients to integrate a system of n simultaneous
-- first order ordinary differential equations dxdt(j) = dx(j)/dt,
-- (j=1,2,...,n), across one step of length h in the independent
-- variable t, subject to initial conditions x(j), (j=1,2,...,n). Each
-- dxdt(j), the derivative of x(j), must be computed two times per
-- integration step by calling the state derivatives and output
-- equations function (subsys_1()). savex(j) is used to save the
-- initial value of x(j) and phi(j) is the increment function for the

```

```
-- j(th) equation. As written, n may be no larger than 5.
-- (Modified Euler)
procedure rungekutta2(n      :in RT_INTEGER;
                     x      :in out RT_FLOAT_AY_5_P;
                     dxdt  :in RT_FLOAT_AY_5_P;
                     t      :in RT_FLOAT;
                     h      :in RT_FLOAT) is

    phi      : RT_FLOAT_AY(0..5);
    savex    : RT_FLOAT_AY(0..5);
    j        : RT_INTEGER;
    retval   : RT_INTEGER;
    hh       : RT_FLOAT := t;

begin

    I.rinfo(0) := hh;  -- TIME
    I.rinfo(1) := h;  -- SAMPLE INTERVAL
    I.rinfo(2) := 0.0; -- SKEW
    I.rinfo(3) := 0.0; -- START TIME
    I.iinfo(2):=1; I.iinfo(3):=1; I.iinfo(4):=1;
    subsys_1;

    for j in 0..n loop
        savex(j) := x(j);
        phi(j) := dxdt(j);
        x(j) := savex(j) + h*dxdt(j);
    end loop;
    hh:=t+h;

    I.rinfo(0) := hh;  -- TIME
    I.rinfo(1) := h;  -- SAMPLE INTERVAL
    I.rinfo(2) := 0.0; -- SKEW
    I.rinfo(3) := 0.0; -- START TIME
    I.iinfo(2):=1; I.iinfo(3):=1; I.iinfo(4):=1;
    subsys_1;

    for j in 0..n -1 loop
        x(j) := savex(j) + (phi(j) + dxdt(j))*h/2.0;
    end loop;
end rungekutta2;
...

end Subsys_1_pkg;
```

6.7 Hints

When dealing with a system containing a single continuous subsystem, AutoCode generates a **SCHEDULER_FREQ** of 1.0 (that is, the inherent rate of the continuous subsystem is that of the scheduler, 1.0). Additionally, when dealing with a hybrid system, AutoCode, by default, treats the continuous subsystem as the fastest task to be dispatched (again, the inherent rate of the continuous subsystem is that of the scheduler). This value might not reflect the true dynamics of the system. In order to obtain an approximate rate for the continuous task, you need to use `sim` iteratively (or `lin` for predominantly linear systems) to arrive at an optimal step size for the integration algorithm, and thus, an approximate sampling interval for the continuous task. For a continuous system (represented by differential equations), the step size is related to its eigenvalues (the eigenvalues vary in time for nonlinear systems). Therefore, AutoCode cannot calculate the average step size.

Typically, a continuous system needs to be sampled 5 to 15 times faster than the smallest time constant in the system (depending on the order of the integration algorithm). This time constant is the reciprocal of the largest eigenvalue in the system and this information can be obtained with `lin`.

7

Using VxWorks with AutoCode

This chapter describes the VxWorks AutoCode C template package with MATRIX_X[®] release 7.X and Tornado 2. This includes a description of generating the real-time application source code such as **super_cruise.c** using the VxWorks template. It also provides a way to run and test this application code using a sample application driver program. For both of the examples in this chapter, you need to have Tornado 2 installed on your host and a target running VxWorks 5.4. Currently, we support the following target CPU types:

- SIMNT - VxWorks simulator (see [Increasing SIMNT Memory Size](#))
- PPC604 - single processor (see [For PPC604 targets](#))
- I80486 - single processor (see [For I80486 targets](#))



NOTE: Other x86 CPU types and their corresponding BSPs could easily be used.

7.1 Template Features

The VxWorks template provides the following features:

- Scheduler and task overflow limit have been implemented. If the scheduler or any task is unable to complete its execution within the set number of clock cycles, then it is allowed to take a few more clock cycles to finish. You can change this value in the **usrData.h** file. If this limit is exceeded, then the application reports an error and terminates.

- The stop and restart capability is fully functional.
- Supports BetterState related code generation.
- The **gencode.bat** file can generate application code for any model. For example, typing **gencode lander** at the appropriate command prompt, generates code for the model **lander**. Typing **gencode only**, generates the default Supercruise code.
- Similarly, **makefile.cmdline** can build the downloadable for any model. For example, typing **make -f makefile.cmdline PROJ=lander** starts the build for the **lander** model on the default I80486 CPU type.
- The real-time application shuts down whenever an AutoCode related error is encountered.
- The **super_cruise.c** file is no longer instrumented with **printf** statements.
- The AutoCode generated code uses semaphores instead of message queues as the IPC mechanism. This improves the run-time performance of the real-time application.
- The application can take inputs from an Xmath format file and run until all inputs are processed. Upon shut-down, an output file is generated containing the computed results of the input data.

7.2 Generating Code

In order to use the VxWorks template, you need the following files:

<code>vxworks.tpl</code>	VxWorks template source file
<code>vxworks.dac</code>	Compiled template file - needed if <code>vxworks.tpl</code> can't be compiled
<code>gencode.bat</code>	Batch file for generating the application code
<code>appl.rtf</code>	A real-time file of a SystemBuild model, for example, super_cruise.rtf .
<code>appl.data</code>	An Xmath formatted input data file, for example, super_cruise.data

To generate the code for the given model file, do the following:

1. Open a Command Prompt window.
2. Copy all provided files to your working directory.
3. Change the variables ISIHOME and MATRIXXVER in the **gencode.bat** file to point to your MATRIX_X installation directory and release, respectively.
4. Run the batch file **gencode.bat** from the command prompt.

This generates the following files in your working directory:

<code>super_cruise.c</code>	The real-time application source code
<code>vxworks.dac</code>	Compiled template file - needed if <code>vxworks.tpl</code> cannot be compiled
<code>gencode.bat</code>	Batch file for generating the application code
<code>appl.rtf</code>	A real-time file of a SystemBuild model, for example, super_cruise.rtf
<code>appl.data</code>	An Xmath formatted input data file, for example, super_cruise.data

7.3 Code Testing Method

In order to test this code, make sure that you have copied the following files to your working directory:

<code>makefile.cmdline</code>	Sample makefile to build a downloadable object.
<code>usrAppInit.c</code>	Sample driver program.
<code>auxClk.c</code>	File containing aux clock related functions file.
<code>sa_utils.c</code>	Utility functions that super_cruise.c calls.
<code>super_cruise.c</code>	Application file generated by you.
<code>sa_defn.h,</code> <code>sa_utils.h</code>	Include files used by the application.
<code>appl.h</code>	Include file for the application source code.

<code>usrData.h</code>	Include file with settings you can change.
<code>super_cruise.data</code>	Input data file.

The inputs to the application can now be read from an input file. The computed results will be saved in an output file. This I/O capability is available if the symbol `FILE_IO` is defined. Without this, the code will work like before where you must initialize the hard-coded inputs in the `sa_utils.c` file with appropriate values according to your model needs. If you are running the `super_cruise` model, then please take a look at the `SA_External_Input()` function in the `sa_utils.c` file and follow the instructions given there.

This application is by default built for an 80486 target PC. If your target differs from this, please follow the instructions in the *VxWorks Programmer's Guide* (Chapter: Configuration and Build; Section: Defining the CPU Type) for making changes to the given makefile. Also, follow the *Tornado User's Guide* (Chapter: Projects; Section: Building a Downloadable Application) if you want to use the IDE for building the downloadable image. The makefile supplied is for use at the Command Prompt, and *not* in a Tornado IDE project.

If you are using the aux clock, then you need to custom-build your VxWorks image and boot your target with this. Details for building a custom image can be found in the *Tornado User's Guide* (Chapter: Projects; Section: Creating a Custom VxWorks Image). Changes needed are as follows:

For PPC604 targets

1. Increase `AUX_CLK_RATE_MAX` from 5000 to 50000.
2. Compile with the `-O2` or `-O3` optimization level.

For I80486 targets

Compile with the `-O2` or `-O3` optimization level.

If you are building VxWorks manually, then edit `mv2600.h` in your `ppc604` BSP directory to change the value of `AUX_CLK_RATE_MAX`. Then add the following line in the makefile found in the same location (or in the `pc486` BSP directory):

```
ADDED_CFLAGS = -O2
```

Do a `make clean` followed by `make` to rebuild VxWorks.

Follow these steps for building and loading the test case, assuming you have installed Tornado in `c:\Tornado` and are in your working directory. We also assume that you have your target hooked up to your host through one of the methods mentioned in the *Tornado User's Guide*.

From a Command Prompt:

1. Enter the following to activate all Tornado-related environment variables:

```
c:\Tornado\host\x86-win32\bin\torvars.exe
```

2. Edit `makefile.cmdline` to change:

CPU to your default target type and `EXTRA_INCLUDE` and `VPATH` to point to the correct drive for the Wind River distribution of AutoCode.

Also, note the usage of `EXTRA_DEFINE` which sets the `RS_VXWORKS` symbol. You must set this symbol, the extra include path, and the macro `VPATH` if you are using the Tornado project facility for make.

Add `-DVX_FP` to the list of `EXTRA_DEFINE` if your target supports a floating-point coprocessor. For example:

```
EXTRA_DEFINE=-DRS_VXWORKS -DVX_FP
```

The floating-point define is not needed when using the VxWorks simulator.

Remove `-DUSE_AUX_CLK` if you want your application to free-run. By default the makefile has this symbol defined which implies that your application will be driven by aux clock interrupts. Attempting to use the aux clock with a `SIMNT` target will result in a run-time error.

Add `-DFILE_IO` if you want the application to read its input from the `super_cruise.data` file. By default, this symbol is not defined.

3. If you are using the aux clock, then edit the `usrData.h` file to make changes to the clock frequency (`CLK_FREQ`). See [Usage Notes](#) for the appropriate values. You can also change the value of `OVERFLOW_LIMIT` to a different number.
4. Initialize the input values in the `SA_External_Input()` function in the `sa_utils.c` file if not using `FILE_IO`. Otherwise, edit the `usrData.h` file and change the value of `DATA_FILE_LOCATION` to point to the directory where you have stored `super_cruise.data`.
5. Save all files that have been edited.

6. Issue the command:

```
make -f makefile.cmdline CPU=CPU_TYPE
```

For example:

```
make -f makefile.cmdline CPU=PPC604
```

This compiles all source files and produces a linked object called **super_cruise**. You can ignore the warnings given by the compiler.

Examples of CPU_TYPE are explained in the makefile. If none is given, the default I80486 is used.

7. Start a WindShell in the Tornado IDE. Make sure that you are in the current working directory in this shell and that an appropriate target server is running. For example:

```
cd "d:\users\myname\vxworks\projects\super_cruise"
```

8. Issue the following command to download the linked object **super_cruise** to your target:

```
ld 1,0, "super_cruise"
```

9. Spawn a task to execute the real-time application:

```
sp usrAppInit
```



NOTE: You might want to experiment with the code in **usrAppInit.c**.

10. If you have defined FILE_IO, then you will be prompted to enter the input and output file names. Enter **super_cruise.data** as the input file and a similar name for the output file name at the target console or the hyperterminal.

7.4 Increasing SIMNT Memory Size

If the VxWorks built-in simulator runs out of memory, follow this procedure to increase the available memory:

1. Open a Command Prompt window and change to the following directory:

```
C:\Tornado\host\x86-win32\bin
```

2. Run **torvars** from the command prompt to set up the environment.
3. Start the VxWorks simulator with increased memory (for example, approximately 3 MB):

```
C:\Tornado\target\config\simpc\vxWorks.exe -r3000000
```



NOTE: There is no space between **-r** and the memory size.

4. Configure and launch a target server from the Tornado 2 toolbar
-or-
Configure and launch a target server from the Command Prompt window by changing directory to:

```
C:\Tornado\target\host\x86-win32\bin
```

and then entering this command:

```
set WIND_UID=0
```

followed by:

```
tgtsvr.exe -V -B wdbpipe -R C:/TEMP/tsfs -RW -n vxsim -c C:/Tornado/  
target/config/simpc/vxWorks yourNode
```

where *yourNode* is the name of your machine.

5. Launch a WindShell and communicate with the target as usual. You can launch the WindShell from the Tornado 2 toolbar or from the Command Prompt:

```
windsh.exe vxsim@yourNode
```



WARNING: When a simulator target is booted, a number of tasks have a nonzero **Errno**. These errors occur before **super_cruise** is loaded, but cause no harm.

7.5 Usage Notes

When you are using the VxWorks AutoCode C template package with MATRIX_X 7.X and Tornado 2, the following usage notes may be of interest:

- The code in **usrAppInit.c** is for demonstration and test purposes only.
- Recommended values to use for the aux clock frequency are:
 - For I80486 targets Values between 2 and 1024 Hz.
 - For PPC604 targets Values between 40 and 10000 Hz.
 - For SIMNT Not applicable.
- The total expected printed output of the `super_cruise` application driven by the aux clock and without any interference from run-time print messages is:

```
SA_Background: Starting aux clock INT...
Enter Xmath {matrixx, ASCII} formatted input file name:

Enter output file name:
```

If, at any time the scheduler or a subsystem task exceeds the number of clock cycles set by `OVERFLOW_LIMIT` in **usrData.h**, while trying to complete its execution, the system reports this situation and terminates by deleting all the tasks and cleaning up after them.

- The aux clock used by the PC486 BSP is the CMOS RTC. It is limited to using one of only 13 clock speeds which are powers of two and in the range [2,8192]. By contrast, the PPC604 aux clock value is limited to the range of [40,50000], but not constrained to be a power of two. Setting the aux clock speed to any frequency outside of these ranges and constraints results in the failure of the `sysAuxClkRateSet()` call and the application will not be able to run.

All tasks will just be in a pending state. At this point, you will need to issue a **progStop** command to get out of such a situation.

- You can use WindView to observe the application graphically and confirm its correctness as far as scheduling goes. However, it is not advisable to interfere with the target in any way (for example, typing a command at a WindShell, using WindView, using the browser's spy chart) when the application is running at a significantly high clock speed. The program can only tolerate such activities (without causing any scheduler or task overruns) at low clock speeds. For information about WindView, see the *WindView User's Guide*.

- The browser's spy chart and WindView (with time stamping) both use the aux clock - hence they will interfere with the application if either of them is used while the application is running. However, we have not observed any significant interference.
- The application free-runs when the aux clock is not used. This means that its period is equal to the speed of the slowest task in the system. The SIMNT target supports only the free-run mode of operation. If you attempt to use the aux clock with the SIMNT target, you will get an error message.
- The include file `sa_defn.h` has been merged into the standard AutoCode 7.X distribution.



WARNING: When using FILE_IO, it is essential that you spawn `usrAppInit` rather than just invoke `usrAppInit`. By spawning it, all I/O gets redirected to the target's console or the hyperterminal. Hence, anything that you enter from the keyboard gets echoed back on to the screen. If you don't spawn, then I/O is done at the WindShell, where the things that you type are not echoed.

- MATRIX[®] 7.X support files for Tornado 2 are included in the AutoCode 7.X distribution in the `%MATRIX%\case\acc\templates\apps\vxworks` directory. This includes the following files:

```
appl.h
auxclk.c
gencode.bat
makefile.cmdline
readme.txt
sa_defn.h
sa_utils.c
sa_utils.h
super_cruise.c
super_cruise.data
super_cruise.rtf
usrappinit.c
usrdata.h
vxworks.dac
vxworks.tpl
```



NOTE: The above files, including possible updates, may be available on the Wind River FTP site in the form of a `.zip` file.

8

Customizing AutoCode and Generated Code

This chapter provides advanced methods for customizing AutoCode and its output real-time code using AutoCode configuration options, templates, BlockScript, and %variables.

8.1 Introduction

You can customize the AutoCode process and the generated output code to suit your specific needs. The different ways you can do this are listed below and described in detail in the sections that follow:

- AutoCode configuration options allow you to specify indentation, coding of significant digits for numeric literals, minimum scheduler frequency, and output file name as described in [AutoCode Configuration Options](#), p.140.
- Templates allow you to modify the overall architecture of generated code, customize the scheduler, modify data structures and external I/O calls, add user codes described in [Templates](#), p.140.
- BlockScript enables you to create your custom block algorithm and generate it in-line in the output source files described in [BlockScript Block](#), p.140.
- Data parameterization (%variable) allows the numeric literals in the block algorithms to be represented by named variables (%variables) as described in [Data Parameterization](#), p.142.

- Using existing code libraries and interfaces to hardware are accomplished by using a UserCode Block as described on [p.143](#) or Macro Procedure as described on [p.143](#).

8.2 AutoCode Configuration Options

You can specify the AutoCode configuration from a SystemBuild form (see [Chapter 2](#)), by using Xmath Commands window options or by using operating system command options. For information on Xmath Commands window or operating system command options, see [Appendix A, AutoCode Options](#).

8.3 Templates

Templates serve as the front end to AutoCode. They determine completely what the output code should be for a given model (.rtf file) and command options. You use the template programming language (TPL) to specify the templates, which are merely TPL programs. We provide templates for both C and Ada code generation that, when compiled, will produce what is called a standalone simulation executable.

Templates and the TPL are described in the *Template Programming Language User's Guide*.

8.4 BlockScript Block

The block algorithms for supplied blocks cannot be modified or customized through AutoCode templates. However, you can create your own block by specifying the algorithm in a BlockScript block. A BlockScript block uses a

scripting language called BlockScript that is translated into C or Ada code and it is generated along with the other blocks in the system. BlockScript provides a generalized programming capability for defining SystemBuild blocks for simulation and code generation, and can also be accessed from BetterState. BlockScript allows you to write the update equations that process the inputs and parameters to produce the outputs. BlockScript I/O can be read by the Data Dictionary. BlockScript is documented in the *AutoCode Reference* and the *BlockScript User's Guide*.

[Example 8-1](#) shows a user-defined BlockScript algorithm calculating the average of 5 numbers (using a BlockScript WHILE loop).

Example 8-1 Example of BlockScript Using WHILE Loop

```

Outputs: y;
parameters: p;
Float y, p(5);
Float sum;

sum=0.0;
k=1;
While k<p.size Do
    sum=sum+p(k);
    k=k+1;
EndWhile;
y=sum/p.size;

```

Resulting generated C Code segment:

```

/*----- BlockScript */
/* {gplvar.Thru_Var.3} */
sum = 0.0;
k = 1;
while (k < 5) {
    sum = sum + myvar[-1+k];
    k = k + 1;
}
Y->Thru_Var_1 = sum/5;

```

Resulting generated Ada Code segment:

```

----- BlockScript --
-- {gplvar.Thru_Var1.3} --
sum := 0.0;
k := 1;
while k < 5 loop
    sum := sum + myvar(-1+k);
    k := k + 1;
end loop;
Y.Thru_Var_1 := sum/RT_FLOAT(5);

```

The **parameters: p;** statement causes the Parameters View in the BlockScript form to include a new 5-by-1 **Parameter p**. A %variable **%myvar** has been defined for this parameter. This causes AutoCode to replace all occurrences of **p** in the previous and following scripts to be replaced by **myvar**.

For more information about programming with BlockScript, see the *AutoCode Reference* and the *BlockScript User's Guide*.

8.5 Data Parameterization

AutoCode users have a choice of generating block data with constant values entered in the Block form or to use Xmath variables (%variables) to represent the data symbolically. While generating code from the SystemBuild menu, this choice is made via the **Block Parameters** option, which can have values of % **Xmath vars** or **Block Defaults**. [Example 8-2](#) shows generated code for a gain block using block default data, and [Example 8-3](#) shows generated code for a gain block using an Xmath variable called **gainvar**, which is initialized to 5.6 in the Xmath partition.

Example 8-2 **Generated Code for a Gain Block Using Block Default Data**

```
y = 2.3 * u;
```

Example 8-3 **Generated Code for a Gain Block Using Xmath Initialized Variable**

```
0 generated code:  
VAR_FLOAT gainvar = 5.6;  
.  
.  
y = gainvar * U->gainvar_1;
```



CAUTION: Changing %variables can in certain cases, such as feedback loops, cause the blocks to be executed out of order. The result of the application might not match the SystemBuild simulation.

8.6 UserCode Block

A UserCode Block (UCB) provides a strict interface between an AutoCode-generated system and some other code. The idea is that you can implement a particular functionality more efficiently by supplying code rather than attempting to model it within SystemBuild. Such functionality includes operations dealing with hardware or reusing existing code found in libraries.

A UCB can also be used to increase the performance of the SystemBuild Simulator by linking back code within a UCB directly to the simulator. The code can be either handwritten or discrete procedural code generated by AutoCode.

For more information about the UCB interface and linking back into the Simulator, see the *AutoCode Reference*.

8

8.7 Macro Procedure Block

A Macro Procedure block provides a C-macro like capability in the generated code. The macro's functionality is to be modeled within SystemBuild so that the simulation matches, but within the generated code, only a macro name will be generated. For example, you can model a function that returns the maximum of 2 numbers, but it is much more efficient to use the MAX macro provided in C. Therefore, the implementation of a macro procedure must be supplied by you in the generated code for the code to compile. Macro procedures provide an inline capability for small code fragments. See the *AutoCode Reference* for more details about the code generated for a Macro Procedure block.



NOTE: AutoCode supports macro procedure blocks for Ada as well as C. However, there is no standard C-macro like capability in Ada. Therefore, we recommend that you implement the Macro Procedure as a standard procedure and use the `INLINE` pragma.

8.8 ZeroCrossing Blocks and Resettable Integrators

AutoCode can generate code for ZeroCrossing blocks and the resettable integrator. These representations are an approximation of the simulator implementation given the real-time nature of the generated code. You must use **actiming** and the fixed-point algorithm during simulation to ensure that your generated code will approximate the simulation results obtained with the fixed-step integration algorithms supported by AutoCode. Your AutoCode results, however, will not exactly match the simulation results when your model uses ZeroCrossing blocks.



NOTE: Event detection is not supported.

For a complete discussion of ZeroCrossing blocks and resettable integrators, see the *SystemBuild User's Guide*.

8.9 User-Defined Code Comments

AutoCode provides the following tokens for adding comments within generated code:

- **blk_code_cmt** for blocks
- **sb_code_cmt** for SuperBlocks
- **ds_code_cmt** for DataStores

These code-comment tokens are predefined user parameters, which are especially useful if you plan to generate documentation as described in the *DocumentIt User's Guide*. For additional information on user parameters, see the *SystemBuild User's Guide*.

8.9.1 Using a User-Defined Code Comment

To use a code-comment token, take the steps described in the following phases.

Phase One

Before code generation:

1. Create an appropriate user-parameter.
2. Within a block, create a user-parameter with the name **blk_code_cmt_s**.



NOTE: Within a SuperBlock, create the user-parameter with the name **sb_code_cmt_s** and for a DataStore, create a user-parameter with the name **ds_code_cmt_s**.

3. Repeat this process for each of the blocks, SuperBlocks, and DataStores where you want comments to appear in the generated code.

Phase Two

The second phase occurs during code generation. To insert the comments you created within the user-parameters into the generated code, do one of the following:

- Generate code by selecting the Enable DocumentIt Block Comments option from the Formatting tab of the Advanced Dialog of the AutoCode code generation dialog.
- Use the **docit** AutoCode keyword.
- Use the **-doc** Xmath command option.

After code is generated, the comments placed within a block's **blk_code_cmt_s** user-parameter appear where the block appears in the generated code. For SuperBlocks, only Procedure SuperBlocks have the comments placed within the generated code. Those comments from the **sb_code_cmt_s** user-parameter are placed at the definition of the function that represents the Procedure SuperBlock. For DataStores, the comments within the **ds_code_cmt_s** are placed at the definition of the DataStore.

Wind River recommends that the content of the user-parameters be plain-text, rather than Rich Text Format or other formatted text content, because the contents are placed within code.

8.9.2 Limitations

The code-comment tokens have the following limitations:

- Any “basic block” can use the **blk_code_cmt_s** user-parameter, including the SuperBlock Block (that is, a SuperBlock reference).
- Any DataStore can use the **ds_code_cmt_s** user-parameter.
- Any SuperBlock definition can use the **sb_code_cmt_s**, but only for Procedure SuperBlocks (all variations) will the comments appear in the code.

9

Introduction to Software Constructs with AutoCode

This chapter is an introduction to the blocks that implement typical software constructs just as loops and decision statements. This includes UserCode Blocks, Macro Procedure Blocks, and Procedure SuperBlocks.

9.1 Introduction

We call blocks that implement typical software logic (such as loops and decision statements) *software constructs* to differentiate from other blocks that compute a result (that is, functional blocks). In other words, software construct blocks deal with the control flow of the program rather than the data flow of the model, allowing the automatically generated code to more closely mimic handwritten code.



NOTE: Unless otherwise noted, these blocks are not supported in Continuous SuperBlocks.

9.2 Standard Procedure SuperBlocks

Standard Procedure SuperBlocks are included in the discussion of software constructs because a procedure represents good software engineering by creating a modular piece of code that can be reused throughout the model. When procedures are reused, code size can be greatly reduced. Maintenance of your design is made easier as fixes are made only in one place. Testing is more tractable as a procedure defines an encapsulated unit that can be independently tested, validated, and verified. Wind River recommends that you use Standard Procedures within your model.

9.3 Variable Blocks

Variable Blocks represent actual variables within the generated code. SystemBuild has two types of Variable Blocks, one that represents a global variable and the other that represents a local variable.

9.3.1 Global

A Global Variable Block is used for a variety of purposes. Traditionally, it has been used to communicate information between Asynchronous Procedure SuperBlocks and subsystems. Also, data shared across multiple processors can be easily accessed. Another usage of Global Variable Blocks is to provide persistent data during execution of the system. Standard Procedures can use Global Variable Blocks as well.

Global Variable Blocks represent global variables in the code. Therefore, Global Variable Blocks are implemented to preserve determinacy. This is only an issue for multi-rate and multi-processor systems. However, even for single-rate systems, overhead and special semantics are associated with a Global Variable Block when data is read and written.

Generally speaking, all reads from a Global Variable Block occur at the beginning of the subsystem for that time point or activation frame, while all writes to the Global Variable Block occur at the end of the subsystem or activation frame. See the *AutoCode Reference* for more details.

9.3.2 Local

A Local Variable Block is similar to a Global Variable Block, except Local Variable Blocks represent local variables in the generated code. Therefore, a Local Variable Block *cannot* hold persistent data, and *cannot* be used to communicate information across processors, subsystems, and procedures. Local Variable Blocks provide efficient communication within a subsystem, and are used with the Iterator and IfThenElse Blocks.

9.4 Graphical Software Constructs

Software constructs are graphical representations of typical software functions. These functions include:

- IfThenElse blocks
- Looping
- Ordering or sequencing the flow of data and calculations
- Using local or global variables

These are basic functions used in pseudo-code for software design. Instead of using pseudo-code, a designer can use the function blocks to design the procedure, and then automatically generate code.

Vectors

You can use vectors to reduce code size when vector type data is being passed or operated on. Example x and example y illustrate the code size reduction for two 5-element vectors, through the use of arrays in the code. You can tailor generated code through options in the Optimization tab of the Advanced AutoCode dialog (as shown in [Optimizations](#)).

9.5 IfThenElse Block

The IfThenElse block implements a decision within the generated code, and then executes one sequence of blocks. This block is like an if statement in C or Ada. The IfThenElse block has output ports (pins) and a prolog section. The output ports are available from the face of the first condition block in the IfThenElse block chain. You can connect to these ports as if they were the output ports of any other standard block. Output from each of the block sections is connectable to the output ports the same way as the output of the content of a SuperBlock is connected to its external output.

9.5.1 IfThenElse Block Example

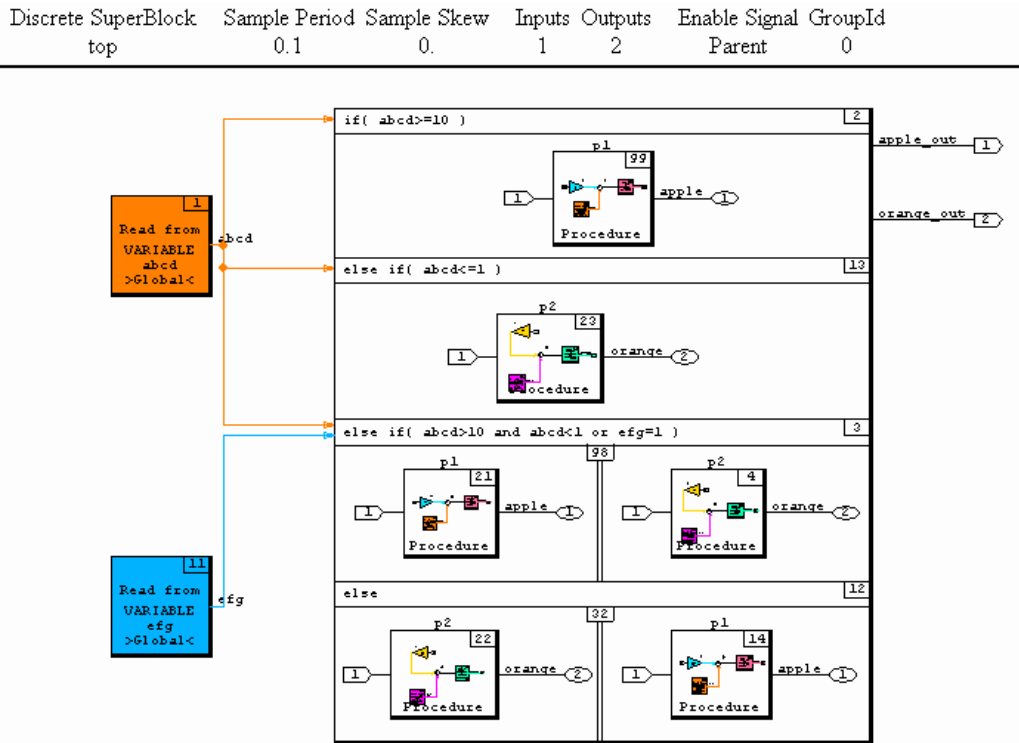
In our IfThenElse block example, we have the following problem requirements:

- 2 procedures named P1 and P2, which are implemented in a particular order, based on the values of inputs **abcd** and **efg**.
- The procedures to be executed for specific values of the input criteria as defined as follows:

Input Criteria	Procedures Executed
abcd > 10	P1
abcd < 1	P2
abcd > 5 or efg = 1	P1, P2
efg > 6	P2, P1

Given the above problem description, the generated C code makes use of the IfThenElse clause to implement the requirement.

SystemBuild is used the graphical coding tool to implement the requirements of the Process Activation Table (PAT), using predefined software constructs. The top-level block (shown below) is a discrete SuperBlock, and the software construct blocks are used to structure the code. This example is a grouping of IfThenElse blocks where the required procedures, P1 and P2, are executed based on the input criteria of **abcd** and **efg**. The P1 and P2 procedures are standard procedure SuperBlocks. They represent code that is used multiple times, and is easily re-used.



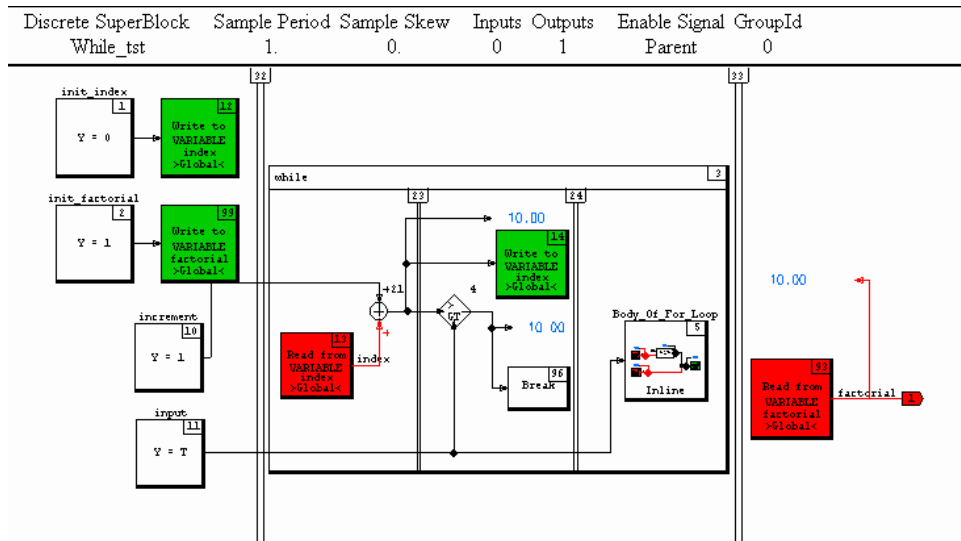
When multiple procedures (or any blocks) must be executed in a specific order the Sequencer can be used to indicate the order. In the above top SuperBlock, notice that the sequencers are the sets of vertical parallel lines. All blocks to the left of a sequencer are executed first, followed by all blocks to the right. The last two “else if” paths in the example actually specify the order of execution of procedures P1 and P2, as indicated by the sequencers.

The outputs of the procedures are then written to variables, which can be accessed from the rest of the system. When using the IfThenElse construct, only the segment whose conditions are met is actually executed. All other segments are not updated, which reduces simulation time. Also, if once the IfThenElse construct is complete, the output **myvar** of the executed procedure is then read.

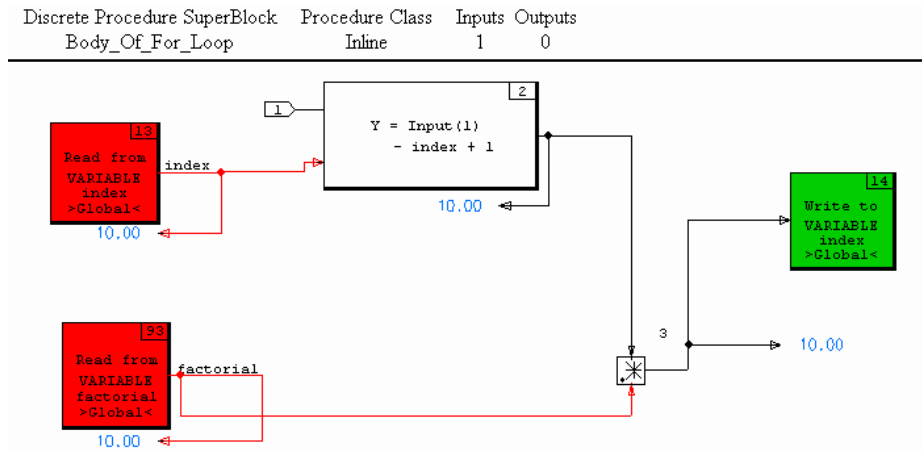
For the generated code from this model, see [Appendix C, Sample AutoCode Output](#).

9.5.2 Looping

Looping is another important software method that is easily accomplished graphically within SystemBuild with the While, Break, and Continue icons. The model shown below uses a For loop to calculate the factorial for each time input. The model uses blocks within the While container that are executed in the current time sample. The Break block allows an exit from the While, when the conditional statement feeding it is true.



In the Body Of For Loop block (inline procedure), we have the following logic:



Note that in the looping example, the inputs and outputs are a mix of direct inputs (increment and input) and local variables (index, factorial).

9.5.3 Ordering or Sequencing the Flow of Data and Calculations

The IfThenElse block example and in looping example illustrate logical flow from software constructs and also with the sequencers. For transferring data and for generating desired calculations, controlling program flow is easily established and then modified with SystemBuild models and generated code.

9.5.4 Using Local or Global Variables

In the looping example shown in 9.5.2, p.152, the local variable is not persistent across time samples. Once the current iteration of the software is complete, the values of the local variables are lost. It is also necessary to initialize all local variables upon entering a SuperBlock or procedure that uses them.

9.5.5 Other Coding Considerations

The procedure Body Of For Loop is an inline procedure. Therefore, the primitive blocks nested within the inline procedure are merged into the subsystem of the parent SuperBlock. As a result, use of inline procedures can result in a different block execution order and can help eliminate potential algebraic loops. An

alternate choice is to use a standard procedure, and the corresponding code would contain a call to the procedure, rather than the actual code.

An important automatic code generation feature is self-documentation. You can use the Comment tab on the Properties dialog of any block to describe the purpose of the block and how it fits into the overall system. Your comment is then automatically incorporated into the generated code by selecting the enable flag at code generation time. The code shown in example xxx illustrates typical comments. A recommended practice is to have a one-to-one correspondence between lines of code and comment lines.

9.6 *Iterator Block*

The While block provides a container that defines blocks that will continue to execute until a condition or set of conditions is met. This type of block is like a **while** loop in C and Ada.

9.7 *Explicit Block Sequencing*

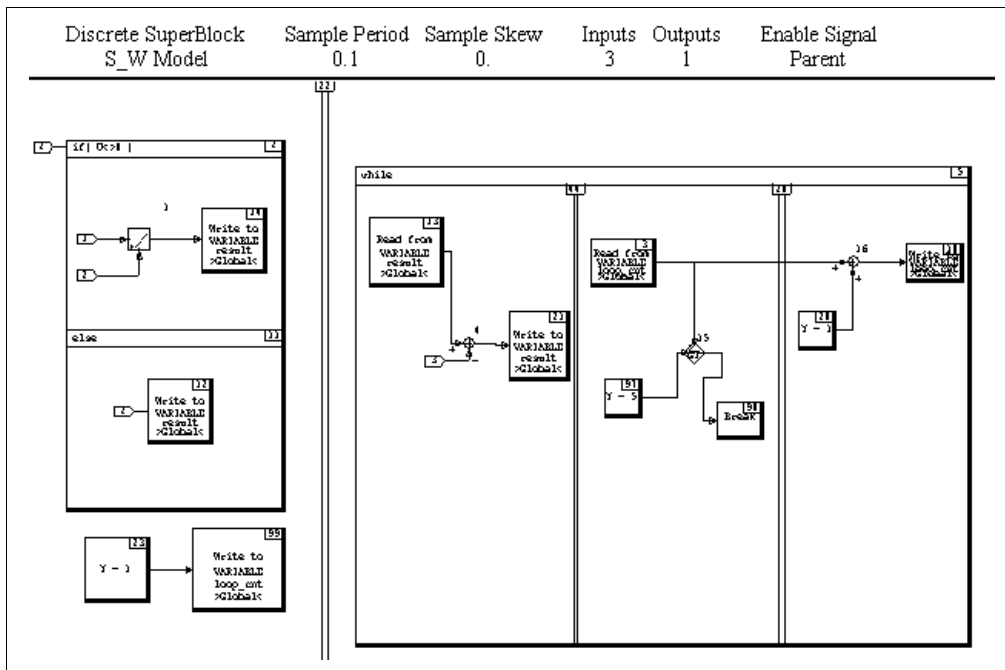
The SystemBuild Analyzer and AutoCode automatically determine the sequence in which blocks are executed. However, there may be algorithms you design that require a set of blocks to be executed before another set of blocks. You can control the sequencing of blocks by using the Sequencer block to divide the diagram into frames; the frame on the left side executes before the frame on the right side. No code is generated for a Sequencer block.

Explicit sequencing is critical for managing blocks such as Global and Local Variable blocks, IfThenElse blocks (see [IfThenElse Block](#)), and possibly Standard Procedure SuperBlocks.

9.8 Example Model

Figure 9-1 shows a model with software constructs, and Example 9-1 shows the code generated for this model.

Figure 9-1 **Sample Model with Software Constructs**



The following C code (Example 9-1) was generated from the model in Figure 9-1 with the Variable Block Read and Constant Propagation optimizations.

Example 9-1 **Generated Software from Software Constructs Model (excerpt)**

```

/*****
|                               AutoCode/C (TM) Code Generator V7.x
|                               WIND RIVER SYSTEMS INC., SUNNYVALE, CALIFORNIA
|                               *****/
rtf filename      : S_W_Model.rtf
Filename         : S_W_Model.c
Dac filename     : c_sim.dac
Generated on     : Wed Jun  2 19:10:16 2000
Dac file created on : Thu Mar 25 10:10:09 2000
Options         : -l c
--

```

```

-- Number of External Input : 3
-- Number of External Output: 1
--
-- Scheduler Frequency:      10.0
--
-- SUBSYSTEM  FREQUENCY  TIME_SKEW  OUTPUT_TIME  TASK_TYPE
-- -----
-- 1          10.0      0.0        0.0          PERIODIC
*/

#include <stdio.h>
#include <math.h>
#include "sa_sys.h"
#include "sa_defn.h"
#include "sa_types.h"
#include "sa_math.h"
#include "sa_matrix.h"
#include "sa_user.h"
#include "sa_utils.h"
#include "sa_time.h"
#include "sa_fuzzy.h"

/** System Data **/

/***** Structure to drive disconnected input/output. *****/

struct _DcZero {
    RT_FLOAT dzero;
};

static const struct _DcZero dczero = {0.0};

#define EPSILON                1.49011611938476562E-08
#define EPS                    (4.0 * EPSILON)
#define ABSTOL                 EPSILON
#define XREMAP                  1

#define SCHEDULER_FREQ        10.0
#define NTASKS                 1
#define NUMIN                   3
#define NUMOUT                  1
#define SCHEDULER_ID           0
#define PREEMPTABLE            2

enum TASK_STATE_TYPE { IDLE, RUNNING, BLOCKED, UNALLOCATED };

static RT_INTEGER              ERROR_FLAG [NTASKS+1];
static RT_BOOLEAN              SUBSYS_PREINIT [NTASKS+1];
static RT_BOOLEAN              SUBSYS_INIT [NTASKS+1];
static enum TASK_STATE_TYPE    TASK_STATE [NTASKS+1];

/***** System Ext I/O and Sample-Hold type declarations. *****/
struct _Sys_ExtOut {
    RT_FLOAT dzero;
};

```

```

struct _Sys_ExtIn {
    RT_FLOAT S_W_Model_1;
    RT_FLOAT S_W_Model_2;
    RT_FLOAT S_W_Model_3;
};

struct _Subsys_1_in {
    RT_FLOAT S_W_Model_1;
    RT_FLOAT S_W_Model_2;
    RT_FLOAT S_W_Model_3;
};

/**** System Ext I/O and Subsystem I/O type definitions and ****
**** Pointers to SubSystem Outputs ReadOnly/Work areas. ****/
struct _Sys_ExtOut sys_extout;
struct _Sys_ExtIn sys_extin;
struct _Subsys_1_in subsys_1_in;

static RT_FLOAT          ExtIn          [NUMIN+1];
static RT_FLOAT          ExtOut         [NUMOUT+1];

/* Model variable definitions. */
VAR_INTEGER loop_cnt;
VAR_FLOAT result;
/* Model variable declarations. */
extern VAR_INTEGER loop_cnt;
extern VAR_FLOAT result;

/***** Tasks declarations *****/

/***** Subsystem 1 *****/
extern void subsys_1( struct _Subsys_1_in *U);

/***** Tasks code *****/

/***** Subsystem 1 *****/

void subsys_1( struct _Subsys_1_in *U
)
{
    static RT_INTEGER iinfo[4];

    /***** Local Block Outputs. *****/

    RT_INTEGER S_W_Model_23_1;
    RT_FLOAT S_W_Model_1_1;
    RT_FLOAT S_W_Model_13_1;
    RT_FLOAT S_W_Model_4_1;
    RT_INTEGER S_W_Model_3_1;
    RT_INTEGER S_W_Model_97_1;
    RT_FLOAT S_W_Model_15_1;
    RT_INTEGER S_W_Model_20_1;
    RT_INTEGER S_W_Model_16_1;

    /***** Initialization. *****/

```

```

if (SUBSYS_PREINIT[1]) {
    iinfo[0] = 0;
    iinfo[1] = 1;
    iinfo[2] = 1;
    iinfo[3] = 1;
    SUBSYS_PREINIT[1] = FALSE;
    return;
}

/***** Output Update. *****/
/* ----- IfThenElse */
/* {S_W Model..2} */
if( U->S_W_Model_2 != 0.0 ) {

    /* ----- ElementDivision */
    /* {S_W Model..1} */
    S_W_Model_1_1 = U->S_W_Model_1/U->S_W_Model_2;
    /* ----- Write to Variable */
    /* {S_W Model..14} */
    result = S_W_Model_1_1;

}
else {

    /* ----- Write to Variable */
    /* {S_W Model..12} */
    result = U->S_W_Model_2;

}
/* ----- Algebraic Expression */
/* {S_W Model..23} */
S_W_Model_23_1 = 1;

/* ----- Write to Variable */
/* {S_W Model..99} */
loop_cnt = S_W_Model_23_1;

/* ----- While */
/* {S_W Model..5} */
while (TRUE) {

    /* ----- Read from Variable */
    /* {S_W Model..13} */
    S_W_Model_13_1 = result;
    /* ----- Summer */
    /* {S_W Model..4} */
    S_W_Model_4_1 = S_W_Model_13_1 - U->S_W_Model_3;
    /* ----- Write to Variable */
    /* {S_W Model..21} */
    result = S_W_Model_4_1;
    /* ----- Read from Variable */
    /* {S_W Model..3} */
    S_W_Model_3_1 = loop_cnt;
    /* ----- Algebraic Expression */
    /* {S_W Model..97} */
    S_W_Model_97_1 = 5;
}

```

```

/* ----- Relational Operator -- LT-EQ-GT */
test = S_W_Model_3_1 > S_W_Model_97_1;
/* ----- Break */
/* {S_W Model..98} */
if( test ) {
    break;
}
/* ----- Algebraic Expression */
/* {S_W Model..20} */
S_W_Model_20_1 = 1;
/* ----- Summer */
/* {S_W Model..16} */
S_W_Model_16_1 = S_W_Model_3_1 + S_W_Model_20_1;
/* ----- Write to Variable */
/* {S_W Model..10} */
loop_cnt = S_W_Model_16_1;
}

if(iinfo[1]) {
    SUBSYS_INIT[1] = FALSE;
    iinfo[1] = 0;
}
return;
EXEC_ERROR: ERROR_FLAG[1] = iinfo[0];
    iinfo[0]=0;
}

```


A

AutoCode Options

This appendix describes options that can be used when invoking AutoCode from within Xmath or from the OS prompt. This appendix also describes how to use an `autostart.opt` file. This appendix supplements [Chapter 2, Using AutoCode](#).

A.1 Options When Invoking AutoCode

As described in [Chapter 2](#), AutoCode can be invoked from the Catalog Browser, the Xmath Commands window, or the operating system prompt. [Table A-1](#) lists the various AutoCode command options. The code generator is invoked by using the `autocode` command (Xmath) or the `autostar` command (from OS prompt).

Table A-1 Options When Invoking AutoCode

Xmath Option	OS Option	Description
<code>allgscope</code>	<code>-allgscope</code>	Force all Output Scopes to be Global and all procedure Input Scopes to be Local.
<code>arraymin</code>	<code>-Oarray n</code>	Minimum size of vectorized arrays. (default: 2)
<code>backpmap</code>	<code>-bmap map</code>	A string specifying the map associating background procedures with processors. The syntax parallels that specified for the <code>subsysmap</code> option, except procedure numbers are used rather than task numbers.

Table A-1 Options When Invoking AutoCode (Continued)

Xmath Option	OS Option	Description
<code>config</code>		Replaced by options .
<code>csi</code>	<code>-csi n</code>	csi specifies the continuous task sample interval (see 6.4.2 <i>Xmath Command Options for Continuous Code Generation</i> , p.114 for details.). NOTE: Use the csi option so generated code will match sim results for continuous systems.
<code>docit</code>	<code>-doc</code>	Enables the DocumentIt tokens as described in the <i>Template Programming Language User's Guide</i> .
<code>doublebuf</code>	<code>-doublebuf</code>	Force double-buffering for single-rate systems.
<code>epinfo</code>	<code>-epi</code>	Boolean (default=0). Generate extended procedure information data structures. Procedures generated with this option should not automatically be mixed with those generated without it.
<code>epsilon</code>	<code>-eps</code>	Float (default is machine epsilon). Set the value of epsilon used in the generated code. This option is used to initialize model outputs. epsilon is set to a very small value so that initial outputs will be near zero; this prevents division by zero problems. The AutoCode token epsilon_r is used to access the epsilon value. If epsilon is left blank, a default value is used.
<code>errcheck</code>	<code>-e</code>	Boolean (default=0). Enables error checking in the generated code. Default is 0, error checking disabled.
<code>file</code>	<code>-o file</code>	The default name is taken from the name of the model file; the default extension is .c or .a , depending on the language chosen.

Table A-1 **Options When Invoking AutoCode** (Continued)

Xmath Option	OS Option	Description
fmarker	-f	<p>Boolean (default=0). When true (1), this option's value forces single-precision floating-point markers for encoded numbers (C language option only).</p> <p>Example: no -f option, generated code looks like $y=2.3 * u;$ with -f option, generated code looks like $y=2.3f * u.$</p>
foverflow	-ovfp n	<p>Integer (default=2). Indicate state of overflow protection for integer and fixed-point calculations.</p> <p>0 = overflow protection disabled 1 = overflow protection forced 2 = overflow protection selected by block's option</p>
glbvarblkopt	-Ogvarblk	Optimize read-from global varblocks.
See ^a on a <i>There is no keyword, but has the command help autocode for Netscape help., p.169.</i>	-h	Obtains a help display.
ialg	-i n	<p>ialg specifies the selected integrator as one of the following:</p> <ol style="list-style-type: none"> 1. First order Runge-Kutta -- Euler 2. Second order Runge-Kutta -- Modified Euler 3. Fourth order Runge-Kutta -- Simpson's Second Rule) 4. Kutta-Merson 5. User Integrator <p>See 6.4.2 <i>Xmath Command Options for Continuous Code Generation</i>, p.114 for details.</p>
indent	-indent n	This integer value specifies the amount of indentation in output between levels. Default is 3.

Table A-1 Options When Invoking AutoCode (Continued)

Xmath Option	OS Option	Description
<code>initmerge</code>	<code>-Oinitmerge</code>	Merge block INIT sections into one INIT section, if possible.
<code>interpmap</code>	<code>-imap map</code>	A string specifying the map associating interrupt procedures with processors. The syntax parallels that of the <code>backpmap</code> option.
<code>ipath</code>	<code>-I pathname</code>	Adds a <i>pathname</i> to the list of directories in which to search for template @include files. Can be used multiple times on the command line (limit 10).
<code>krstyle</code>	<code>-kr</code>	Generate old-style (Kernighan and Ritchie) C function prototypes.
<code>language</code>	<code>-l lang</code>	The language for generating the code: C or Ada. The following are accepted: c, C; a, ada, Ada, ADA.
<code>linesz</code>	<code>-linesz n</code>	This integer value specifies the maximum number of output characters per line. The integer value must be ≥ 78 . The default is 80.
<code>loopmin</code>	<code>-Oloop n</code>	Loop threshold for vectorized code (default: 2).
<code>locvarblkopt</code>	<code>-Olvarblk</code>	Optimize read-from local varblocks.
<code>mapfile</code>	<code>-pfile file</code>	A string defining the map file associating subsystems and background, startup, and interrupt procedures with processors. The <code>subsystemmap</code> , <code>backpmap</code> , <code>startpmap</code> , and <code>interpmap</code> options override the specifications in this file, and if none of these options is supplied and the file doesn't exist, it's created using a default map. A single-line comment is indicated using <code>//</code> characters.

Table A-1 **Options When Invoking AutoCode** (Continued)

Xmath Option	OS Option	Description
minsf	-minsf n	Specifies minimum AutoCode scheduler frequency. The real-time scheduler frequency is set to the larger value of the frequency determined by the block diagram application and the value specified by -minsf . Normally, the default value of 0.0 should be used, which allows the application to set its own scheduler frequency. Deviation from this default should be approached with caution, as a consistent scheduler frequency should normally be based on a least common multiple of the inverse of the application timing requirements (that is, frequencies).
namelen	-nl n	This integer value adjusts the maximum variable length in the generated code. The integer value must be ≥ 20 (the default is 48).
nodiscon	-odiscnout	Optimize away disconnected outputs.
noerr	-noerr	Do not generate error detection code after a procedure call is made.
nogscope	-nogscope	Force all Output Scopes to be Local.
noinfo	-noinfo	If possible, eliminate a procedure's INFO structure.
noicmap	-noicmap	Sets the constant variable XREMAP to False, which prevents initial values of states from being set.
nomap	-nomap	Boolean (default=0). Turn off the structure map indicating subsystem and system external inputs and outputs by setting the nobusmap_b token to True.
norestart	-Onorestart	Optimize out the restart capability.
nosmooth	-nosmooth	Turn off floating-point constant number smoothing.
nouy	-nouy	Pass procedure input and outputs as actual arguments to the function.

A

Table A-1 Options When Invoking AutoCode (Continued)

Xmath Option	OS Option	Description
<code>numproc</code>	<code>-np n</code>	Integer (default=1). The number of processors to generate code for.
<code>options</code>	<code>-opt file</code>	Specifies the name of the options file. Options are entered in the file using the same syntax as if they were specified from Xmath or the OS. The exception is that map specifications are not enclosed between quotes. Options can be on one line, separate lines or a combination. Command options override all of the options in the <code>-opt</code> file. The <code>rtf</code> file name cannot be specified in the <code>-opt</code> file. Single line comments are done by using <code>//</code> characters. See <i>A.2 Using the autostar.opt File</i> , p.169 for more information about the options file.
<code>parname</code>	<code>-p</code>	Boolean (default=0). When true (1), specifies use of parameter names specified by scripts in language blocks instead of RP and IP arrays.
<code>priomap</code>	<code>-prio</code>	A string defining the priority of the subsystems. It has a form similar to that of the <code>skewmap</code> option (see above), except <code><skew value></code> is replaced by an integer priority. Provided that AutoCode can assign each subsystem a unique priority while obeying the <code>priomap</code> , range and list operators in the <code>priomap</code> are permitted for both subsystems and priorities.
<code>procs_only</code>	<code>-procs</code>	Sets the template parameter <code>procs_only_b</code> as true , and default template only generates Procedure SuperBlocks and generates UCBs and subsystem wrappers for each of these procedures.
	<code>-prompt</code>	Prompt for command options.
<code>propconst</code>	<code>-Opc</code>	Propagate constants across blocks.
<code>reuse</code>	<code>-Oreuse n</code>	Strategy for reusing subsystem local outputs. 0 : Do not reuse (default) 1 : Reuse by matching named outputs 2 : Reuse whenever possible.
<code>roundfloat</code>	<code>-round</code>	Force an implicit float-to-integer conversion to be rounded rather than truncated.

Table A-1 Options When Invoking AutoCode (Continued)

Xmath Option	OS Option	Description
<code>rtf</code>	See note ^b on p.169.	The name of the generated real-time file (<code>.rtf</code>). Default: modelname.rtf
<code>rtos</code>	<code>-rtos</code>	Boolean (default=0). Read the default Real-Time Operating System configuration file, ac_rtos.cfg , to obtain RTOS parameter information (or create ac_rtos.cfg if it doesn't exist). This option is incompatible with the rtosfile , subsysmap , startpmap , backpmap , interpmap , and priomap options. More information is given in the <i>AutoCode Reference</i> .
<code>rtosfile</code>	<code>-rtosf file</code>	A string specifying the name of the file from which to read the Real-Time Operating System configuration information. This option is incompatible with the RTOS , subsysmap , startpmap , backpmap , interpmap , and priomap options.
<code>scheduler</code>	<code>-sched n</code>	Choose a scheduler type: 0 : one-stage output posting (default) 1 : pre & post output posting
<code>sd</code>	<code>-sd n</code>	Integer. Specifies the number of significant digits for encoded numbers. Default: long constants are emitted to full machine width.
<code>skewmap</code>	<code>-skew n</code>	A string defining the skew of each subsystem. It has the form: <pre><skewmap>::= <subsystem #> <skew value> { <skewmap> ... } <subsystem #>::= An integer naming the subsystem <skew value>::= A float defining the skew</pre> Subsystem numbers and skew values must be separated by a single space in the string. Optionally, a range or list of subsystem numbers can be used instead of <subsystem #> above.
<code>smcallout</code>	<code>-smco</code>	Boolean (default=0). Generate call-outs for access to all elements in shared memory. Turn on the shared memory function call out.

A

Table A-1 Options When Invoking AutoCode (Continued)

Xmath Option	OS Option	Description
<code>startpmap</code>	<code>-smap map</code>	A string specifying the map associating startup procedures with processors. The syntax parallels that of the backpmap option.
<code>subsysmap</code>	<code>-pmap map</code>	A string specifying the map associating subsystems with processors. The map contained in the string has the form: <pre><map> ::= <prsr #> <list> { <map> ... } <prsr #> ::= a processor number (starting at 1) <list> ::= <task #>, { <task #> ... } <task #> ::= a task number (starting at 0)</pre> Processor numbers and lists must be separated by a single space, while elements of the list must be separated by a single comma.
<code>tpldac</code>	<code>-d file</code>	A direct access template file (see Chapter 5). Default: <code>\$CASE/ACC/templates/c_sim.dac</code> for C <code>\$CASE/ACA/templates/ada_rt.dac</code> for Ada
<code>tplsrc</code>	<code>-t file</code>	This is a template source file (see Chapter 5). Default: <code>\$CASE/ACC/templates/c_sim.tpl</code> for C <code>\$CASE/ACA/templates/ada_rt.tpl</code> for Ada
<code>typecheck</code>	See note ^c	Boolean (default=1). Enables data type checking. See the SystemBuild online help for simulation for details about data typing. For AutoCode the default is 1; type checking is enabled. If typecheck is set to false, all the variables in the model are hardcoded with float data type.
<code>ucbparams</code>	<code>-ucbparams</code>	Use RP/IP temporaries as actuals to UCB call instead of %var variables.

Table A-1 Options When Invoking AutoCode (Continued)

Xmath Option	OS Option	Description
vars	See note ^d on p.169.	Boolean (default=1). The default is 1, meaning that %vars are included in the model. Note that if the code is generated from a model, the model is processed using the vars keyword. However, if the AutoCode function is invoked from the operating system prompt using an existing .rtf , the vars keyword is ignored. To turn off vars , specify !vars . Same basic functionality as the vars simulation keyword as described in the <i>SystemBuild User's Guide</i> .
vbcallout	-vbco	Boolean (default=0). Generate callouts around the critical section of variable block accesses in the generated code.
vectormode	-Ov n	Vectorization option: 0 : Scalar code generation (default) 1 : Vectorize based on labeling 2 : Maximal vectorization

- a. There is no keyword, but has the command **help autocode** for Netscape help.
- b. The name of the **rtf** file must always be specified when invoking from Xmath or the OS.
- c. The **typecheck** feature applies only to the creation of the **rtf** file, thus there is no equivalent option from Xmath or the OS.
- d. This Xmath option is used for creation of the **rtf**. When invoking from the OS, the **rtf** must already exist; therefore, there is no OS option equivalent.

A.2 Using the *autostar.opt* File

If you invoke AutoCode with the same options consistently, you can put these options into an options file, saving a lot of error-prone, repetitive typing each time you invoke AutoCode. AutoCode reads the options file at startup, and performs the options as though you had entered them from the O/S prompt. Although you can use an options file whether you invoke AutoCode from the Xmath Commands window or the operating system prompt, the only options that you can specify in the options file are operating system command options.

The default options file is **autostar.opt**. If you have an **autostar.opt** file in the current working directory from which you invoke AutoCode, the options in that file will be executed when you invoke AutoCode. If you specify an option from the Xmath Commands window that is also in the options file, the command option overrides the same option in the options file (see [Example A-1](#) and the paragraphs following).

For different applications, you might need to invoke AutoCode differently. For this reason, you can have multiple options files. To invoke AutoCode with an options file other than **autostar.opt**, specify the name of the options file when you invoke AutoCode (see [Example A-2](#) and the paragraphs following).

Options are entered into the options file using the same syntax as if they were specified from the O/S prompt. The exception is that map specifications are not enclosed between quotes. Options can be on one line, separate lines or a combination. The **rtf** file name cannot be specified in the options file. Single line comments are done by using `//` characters.

[Example A-1](#) shows an options file.

Example A-1 **Example autostar.opt Options File**

```
// Sample options file //  
-l c  
-t c386_c860_mb2.tpl  
-o myoutput
```

To use this file, invoke AutoCode as follows:

```
autostar model.rtf
```

This invokes AutoCode with the **autostar.opt** options file. Output is directed to the file **myoutput**.

With the following command:

```
autostar -o myoutput3 model.rtf
```

The options file is again used, but the output file option (**-o**) is specified, so it overrides the corresponding command in the options file. The output documentation will be in file **myoutput3**, not in file **myoutput** as specified in the options file.

Example A-2 shows an options file called **myopt.opt**.

Example A-2 **Example Options File Called myopt.opt**

```
// Sample options file //  
-l c  
-t c386_c860_mb2.tpl  
-o myoutput2
```

To use this file, invoke AutoCode as follows:

```
autostar -opt myopt.opt model.rtf
```

This invokes AutoCode with the **myopt.opt** options file.

So, if you have both of the above option files (shown in Examples A-1 and A-2) in your directory, invoking **autostar** without the **-opt** option puts the generated code into file **myoutput** (the **autostar.opt** options file is used). Invoking **autostar** with the **-opt myopt.opt** option as shown above puts the generated code into file **myoutput2**.

A.3 Mapping Options

This section describes the options for controlling subsystem execution.

A.3.1 Setting Subsystem Priorities

If you do not specify any priorities, AutoCode allocates priorities to the subsystems in the following manner. For C it allocates priority 0 to the scheduler, and 1, 2, ..., total number of subsystems to subsystem 1, subsystem 2, ..., respectively. For Ada by default, it allocates the number equal to the total number of subsystems to the scheduler, and grows downward for subsystem 1, subsystem 2, ... , respectively.

For C or Ada, if you specify only the scheduler priority using the **-prio** option, the subsystems will be given the priority 1 plus the scheduler priority. Or, if at least one more subsystem is specified, depending on the sequence of priorities you establish in specifying subsystems, the priority order will be ascending or descending.

Depending on the operating system you are using, you might need to change the default mapping. In some operating systems, the smaller the number, the higher the priority, but in others, the opposite sequence might prevail. You can change the priorities by using the **prio** option. Syntax:

```
-prio subsystem# priority
```

When specified from the OS prompt, the syntax is

```
-prio "subsystem# priority"
```

Example:

```
-prio 0 30 1..3 29 3..n 28.
```

From the O/S prompt the above example would be:

```
-prio "0 30 1..3 29 4..n 28"
```

This example sets the priority 30 to scheduler subsystem 0 and 29 to subsystems 1,2 and 3 and priority 28 to subsystem 4 onwards. n is always 65535. Be sure to place the scheduler priority at the beginning of the **-prio** option. The subsystem# and priorities must be given in pairs and delimited by a space.

The priority of a subsystem, if not specified, depends on the previous and successor subsystems. The system will not allocate a priority greater than the successor subsystem. AutoCode will not allocate a negative subsystem ID. Every subsystem priority must be less than the scheduler subsystem priority, whether the sequence of priority numbers is ascending or descending. '..' is used to specify the range and ',' is for listing. In **-pfile** the option can be:

```
-prio 0 30
```

```
1,3 29
```

```
4..7 28
```

As no subsystem can have more than one priority, the priorities cannot have the range or list operators '..',''.

Example:

```
-prio 0 30
```

```
1 28..26
```

```
2 25,24 is INVALID, but
```

Example:

-prio 0 30

1..n 9..2 is VALID and subsystems 1 to n will be assigned priorities between 9 and 2.

A.3.2 Setting Subsystem Skews

With the **skew** option, you can change skews specified in SystemBuild. One use of this option is to reset the skews which have been applied to SuperBlocks in order to split a large subsystem into two or more parts. A reason for splitting up such a subsystem is that the parts can be run in parallel on multiple processors. Resetting the skew ensures that they all start at the same time.

The syntax of the **skew** option is similar to that of the **prio** option.

-skew subsystem# skew

Examples:

-skew 1 .1 2 .002	From Xmath
-skew "1 .1 2 .002"	From the O/S prompt

Only the subsystems can use the range and list operators '..' and ','.

Example:

```
-skew 1      .1
      2..5   .002
      6,8    .003
```

A.3.3 Setting Processor Subsystem Map

When generating code for multiple processors, a subsystem-to-processor map must be specified. If no mapping is specified, AutoCode will map the subsystems to different processors using the following rule:

$$\text{Processor No.} = ((\text{subsys_id}-1)\% \text{max. no. of processors}) + 1$$

For a system with six subsystems and 2 processors, this would assign subsystems 1, 3, and 5 to processor 1 and subsystems 2, 4, and 6 to processor 2. Scheduler 0 is always assigned to processor 1. When the default mapping rule is used, the mapping is saved in file **autocode.pmp** in the working directory. You can edit

this file and specify this file name for future invocations of AutoCode using the **-pfile** option.

If the **-pfile** option is specified but the file does not exist, AutoCode will create the file and save the default mapping to it rather than to **autocode.pmp**. Note that **-pmap** specifications are not saved to the file specified by the **-pfile** option.

To change the default processor mapping, use the **-pmap** option. The syntax is similar to that of **-prio** and **-skew** options.

```
-pmap processor# subsystem#
```

Example:

```
-pmap 1 0 2 2,3,4 3 5,6
```

```
-pmap "1 0 2 2,3,4 3 5,6" (Command Line)
```

The above example allocates subsystem 0 to processor 1 and 2, 3, and 4 to 2 and 5, 6 to 3.

```
-pmap 1 0,1  
      2 2..6
```

The subsystems can have the range and list operators '..', ''.

A.3.4 Processor Map Specification from the OS Prompt

This section describes the command processor mapping for subsystem tasks, background, startup and interrupt procedure SuperBlocks. The mapping is specified by using the **-pmap**, **-bmap**, **-smap**, and **-imap** options.

The following is the format of these options:

```
<option> "<map>"
```

```
where:<option> ::= [ -pmap, -bmap, -smap, -imap ]  
      <map> ::= <prsr #> <list> { <map> ... }  
      <prsr #> ::= <processor number starting at 1>  
      <list> ::= <task/procedure numbers>
```

example:

```
autostar -l c -pmap "1 0,1,3 2 2,4" -np 2 test.rtf
```

You can use the **-pfile** option to generate a default set of mappings. The example maps subsystems 0 (the scheduler), 1, and 3 to processor 1 and subsystems 2 and 4 to processor 2.

B

Software Development Kit

This appendix describes the AutoCode Procedure Software Development Kit (ACP SDK). This SDK provides users of AutoCode generated code an Application Programming Interface (API) to generated Standard and Startup Procedure SuperBlock code. An API to the generated procedures allows a user to integrate the generated code into a non-AutoCode generated system.

B.1 Scope

This appendix is for the MATRIX_x AutoCode user who has built a modular model using Procedure SuperBlocks and now needs to integrate the generated code into a complete system. This appendix assumes that you are familiar with the use of SystemBuild and AutoCode. If not, please see the MATRIX_x Help and the *SystemBuild User's Guide*. This appendix describes the basis for the SDK and provides some examples on the usage of the API.

B.2 Supported Versions and Languages

This SDK has been developed and tested using MATRIX_X 7.X. The kit contains templates to generate C, C++, and Ada code.

The contents of the AutoCode SDK are summarized in [Table B-1](#).

Table B-1 **AutoCode SDK Contents**

Filename/Parameter	Description
<code>ada_sdk.tpl</code>	AutoCode TPL file for generating the Ada version of the procedures and API functions. See <i>B.7.2 Physical Design (cpp_sdk.tpl)</i> , p. 189.
<code>c_sdk.tpl</code>	AutoCode TPL file for generating the C version of the procedures and API functions. See <i>B.4.5 Physical Design (c_sdk.tpl)</i> , p. 180.
<code>ialg</code> (for integration algorithm)	Specifies the integrator selection (corresponds to the IALG Options tag on the Advanced dialog as described in Chapter 2): 0 = user-defined integrator 1 = first order Runge-Kutta integrator 2 = second order Runge-Kutta integrator (default) 3 = fourth order Runge-Kutta integrator 4 = Kutta-Merson integrator
<code>csi</code>	Specifies the continuous task sample interval
<code>minsf</code>	Specifies the minimum AutoCode scheduler frequency in seconds (0.0 is the default).
<code>c_sdk_m.tpl</code>	Same as <code>c_sdk.tpl</code> , except each procedure is generated into a separate <code>.c</code> and <code>.h</code> file. See <i>B.4.6 Physical Design (c_sdk_m.tpl)</i> , p. 181.
<code>c_core.tpl</code>	Common TPL between <code>c_sdk.tpl</code> and <code>c_sdk_m.tpl</code> .
<code>c_sdkcore.tpl</code>	Common TPL code for C and C++ templates.
<code>cpp_sdk.tpl</code>	Same as <code>c_sdk.tpl</code> , except, C++ classes are used for the API. See <i>B.7.2 Physical Design (cpp_sdk.tpl)</i> , p. 189.

Table B-1 **AutoCode SDK Contents** (Continued)

Filename/Parameter	Description
<code>cpp_sdk_m.tpl</code>	Same as <code>cpp_sdk.tpl</code> , except each procedure is generated into a separate <code>.cpp</code> and <code>.h</code> file. See <i>B.7.3 Physical Design (cpp_sdk_m.tpl)</i> , p.190.
<code>cpp_core.tpl</code>	Common TPL between <code>cpp_sdk.tpl</code> and <code>cpp_sdk_m.tpl</code> .
<code>wheel.lib.cat</code>	Example SystemBuild model file. See <i>B.7.6 Example 4: Wheel Program (CPP-SDK)</i> , p.193.
<code>wheeldriver.c</code>	Example driver program (C). See Example B-6 , p.186.
<code>wheeldriver.cpp</code>	Example driver program (C++). See Example B-9 , p.195.

B

B.3 Overview

B.3.1 Procedures-Only SystemBuild Model

This SDK is based on the assumption that you want modular, reusable code to be included into your system. This is accomplished by creating your model using only Procedure SuperBlocks and a single top-level discrete SuperBlock as a *wrapper* for all of the top-level procedures. For each of the top-level procedures, the API interface is generated. Also, the API interface is generated for all Startup Procedure SuperBlocks in the model.

B.3.2 Limitations

The SDK cannot properly generate code for procedures that use Xmath partitions for %vars assigned to Procedure SuperBlock blocks.

B.3.3 Application Programming Interface

The procedures-only SystemBuild model is generated with AutoCode using the SDK template appropriate for the programming language you plan to use. The

template generates code for the procedures in the model and generates an interface to the top-level procedure(s) specified in the top-level discrete SuperBlock of the model. The interface is composed of three elements: interface structure, initialization function, and execute function.

Interface Structure

The interface structure is a data structure containing the private data needed by the AutoCode generated procedure. You will need to create an instance of this structure and pass it as an argument to the other API functions.

Initialization() Function

This initialization function initializes the interface structure. This function must be called once for each instance of an interface structure and called before any other API function.

Execute() Function

The **execute()** function is a wrapper function that interfaces your code to the AutoCode generated procedure. The function arguments vary depending on the procedure's inputs and outputs. Call this function to execute the procedure.

B.3.4 Driver Program

The driver program is code that you implement to call the procedure API functions. The driver is responsible for managing the input/output data of the procedures.

B.4 C API

This section describes the generated C code of the C SDK template.

B.4.1 Logical Design

The logical design of the API consists of one structure and two functions for each of the top-level procedures.

B.4.2 Interface Structure

The interface structure is a C struct containing the private data needed to support the execution of the procedure. The structure is based on the following “boilerplate” which describes the kinds of data found in this structure.

Example B-1 **C Interface Structure**

```
struct _procedurename_ext {
    procedurename_info  procedurename_i;
    procedurename_s     procedurename_s_s;
};
```

where *procedurename* is the name of the top-level procedure and *_ext* is a suffix to distinguish this structure from other structures of the procedure.

The interface structure contains the procedure’s INFO and STATE structures. For more information about these structures, see the *AutoCode Reference*. Depending on exactly what the procedure needs and/or optimizations you have used during code generation, these structures may or may not be present.

Since this is a private data structure, there is no need to discuss its purpose or content any further. The only requirement is that an instance of this structure be created for each instance of the procedure you want to call and that it is properly initialized.

B.4.3 Initialize() Function

This function initializes an instance of (instantiates) the procedure’s interface structure. The function has the following signature. This function must be called once for each instance of the interface structure and before any other API function.

```
void procedurename_initialize(
    struct _procedurename_ext *_idata
);
```

B.4.4 *Execute()* Function

The `execute()` function is the calling interface from your code to the generated procedure. The function has the “boilerplate” signature shown in [Example B-2](#), p.180.

Example B-2 `Execute()` Function

```
RT_INTEGER procedurename_Execute(  
    struct procedurename_u *U,  
    struct procedurename_y *Y,  
    RT_FLOAT TIME,  
    struct _procedurename_ext *_idata  
);
```

If you use the AutoCode `-nouy` option, which passes procedure input and outputs as actual arguments to the function, the code looks similar to [Example B-2](#).

Example B-3 `Execute()` Function (with `-nouy` option)

```
RT_INTEGER procedurename_Execute(  
    RT_datatype u1 [, RT_datatype un]  
    RT_datatype *y1 [, RT_datatype *yn]  
    RT_FLOAT TIME,  
    struct _procedurename_ext *_idata  
);
```

In both of the above examples, struct `procedurename_u` is the procedure’s input argument structure; struct `procedurename_y` is the procedure’s output argument structure; TIME is the current simulated time measured from the start of the simulation.

Depending on the exact number of inputs, outputs, and whether the current time is needed, the interface varies for each procedure. You must declare an instance of the input/output structures and pass all necessary arguments to this function to properly execute the procedure. The function returns an error code, 0 meaning no error; any other is an error that can be determined from predefined error codes in the `sa_defn.h` SA-Library header file.

B.4.5 *Physical Design (c_sdk.tpl)*

The physical design of the API is the grouping of functions and data structures into source files. The template generates two files: the header file and the source file.

Header File

The generated header file contains all of the data structures used by all of the procedures in the generated code. This includes the procedure's input, output, INFO and state data structures as well as the interface structure to the top-level procedure(s) and the function prototypes for the API functions. The header file name has the format *toplevel.h*, where *toplevel* is the name of the top-level discrete SuperBlock of the model.

Source File

The generated source file contains code for all of the procedures including the API functions for each of the top-level procedures. The source file name has the format *toplevel.c*, where *toplevel* is the name of the top-level discrete SuperBlock of the model.

B

B.4.6 Physical Design (*c_sdk_m.tpl*)

The physical design of the API is the grouping of functions and data structures into source files. This template generates multiple files.

Header Files

A header file is *created for each* Standard and Startup Procedure that contains all of the data structures used by that procedure. This includes the procedure's input, output, INFO, and state data structures. The name of each header file is based on the name of the Procedure SuperBlock. The function prototypes for the API functions for all top-level Standard Procedures are created in a separate header file. That header file name has the format *toplevel.h*, where *toplevel* is the name of the top-level discrete SuperBlock of the model.

Source File

A source file is *created for each* Standard and Startup Procedure that contains code for the procedure. The name of each source file is based on the name of the Procedure SuperBlock. The implementation of the API functions for all top-level Standard Procedures are created in a separate source file. That source file name

has the format *toplevel.c*, where *toplevel* is the name of the top-level discrete SuperBlock of the model.

B.4.7 Compilation and Link Details

Given that we are using AutoCode generated code, the same compilation configuration that is needed for the standalone simulation code will be needed when using the SDK functions. There are only three details related to the Standalone Library and machine platform. Of course, all of the individual source files will need to be compiled and then all linked together.

Standalone Library Header Files

You will need all of the Standalone Library's header files for proper compilation of the generated code. You can either copy the Standalone Library header files into your current working directory, or add the appropriate compiler option to specify the library's distribution directory in the compiler's include path.

Platform Indicator

When compiling the generated source file, you must define the appropriate platform symbol. For example, if you are using a Solaris platform, you need to define the pre-processor symbol at the compiler's command prompt, **--DSOLARIS**. See the *AutoCode Reference* for the symbol for your platform and other syntax information.

Standalone Library Source Files

The Standalone Library contains support routines for the generated code. Although mostly used for the standalone simulation, the generated procedure may need some of the Standalone Library's routines, especially if fixed-point is used. If you receive errors when linking, this is most likely a result of not compiling and linking the appropriate Standalone Library source file. Find the needed Standalone Library source file, compile it and link it along with the other object files for your system.

B.5 Sample Code (C-SDK) Example

Example B-4 illustrates the code generated in the header file and Example B-5 is a sample driver program that calls the procedure. The top-level discrete SuperBlock is named: **interfaces** and the top-level procedure is named: **feather**.

Example B-4 Header File (interfaces.h)

```

/*****
|
|           AutoCode/C (TM) Code Generator
|           WIND RIVER SYSTEMS INC.,  SUNNYVALE, CALIFORNIA
|
|*****
rtf filename      : interfaces.rtf
Filename         : interfaces.h
Dac filename     : c_sdk.dac
Generated on    : Tue Feb  3 11:59:07 2000
Dac file created on : Tue Feb  3 11:59:06 2000
                AutoCode Procedure Software Development Kit
Procedure Name   Inputs  Outputs  INFO  States
-----
feather          YES    YES     YES   NO

*/
#ifndef _INTERFACES_H_
#define _INTERFACES_H_

#include "sa_sys.h"
#include "sa_types.h"

/***** Procedure: feather *****/

/**** Inputs type declaration. ****/
struct _feather_u {
    RT_FLOAT input_val;
};

/**** Outputs type declaration. ****/
struct _feather_y {
    RT_FLOAT gain_out;
};

/**** Info type declaration. ****/
struct _feather_info {
    RT_INTEGER iinfo[5];};

/* *****
***** I N T E R F A C E S *****
*****
*/

/*
**
** Interfaces to: feather

```

B

```
**
*/

struct _feather_ext {
    struct _feather_info feather_i;
};

/* Function: feather_Initialize
**
** Abstract: Initialize procedure's private data
**
** Parameters: _idata (in) : ptr to an instance of
**                  procedure's interface data
**
*/
extern void feather_Initialize(struct _feather_ext *_idata);

/* Function: feather_Execute
**
** Abstract: Execute the procedure
**
** Parameters: Procedure's input and output variables
**              Instance of interface structure
**              Current time may be needed as well.
**              Interface varies for each procedure.
**
** Returns: error status, 0 = no error,
**           else see errors in sa_defn.h
**
*/
extern RT_INTEGER feather_Execute(
    struct _feather_u *U,
    struct _feather_y *Y,
    struct _feather_ext *_idata);

#endif
```

Example B-5 is a sample driver program.

Example B-5 Driver Program (main.c)

```
#include <stdlib.h>

#include <stdio.h>
#include "interfaces.h"
int main() {
    struct _feather_u  in;
    struct _feather_y  out;
    struct _feather_ext feather_inf;

    int i;

    srand(304);
    feather_Initialize(&feather_inf);
    for(i=0; i < 20; i++) {
        in.input_val = (RT_FLOAT)(rand() / 8.5);
```

```

    feather_Execute(&in, &out, &feather_inf);
    printf("input: %f\toutput: %f\n", in.input_val, out.gain_out);
}
return 0;
}

```

B.6 Wheel Program (C-SDK) Example

The wheel program example uses a supplied example model and source code to illustrate the steps needed to generate, compile, and execute the driver and SDK generated code.

Step 1: Set up

We need to set up a current working directory. Create a new subdirectory, called **wheeltest**. Change directory to **wheeltest** to make it the current working directory and copy all of the SDK files into it.

Step 2: Generate API

We need to generate the API structures and functions. This is accomplished by performing the following steps:

1. Launch Xmath and SystemBuild from within the current working directory.
2. Load the example model file: **wheellib.cat**.
3. Generate code using AutoCode. From the Xmath Commands window, enter:

```

autocode "wheel_library", { language="c", tplsrc="c_sdk.tpl", procs_only,
nouy }

```

AutoCode generates two source files: **wheel_library.h** and **wheel_library.c**.

Step 3: Compile and Link API Functions and Driver Program

With the API functions generated, all we need to do is compile and link those API functions with the supplied driver program (**wheeldriver.c**) that is our example system.

Perform the following steps from an operating system command prompt:

1. Copy SA Library header files into your current working directory. If you need other SA Library Files, especially the fixed-point implementations, copy those files. If you do not copy the header files, you can add **-I** include path directives to the compile line.
2. Compile and link the program as follows:

(UNIX - Solaris)

```
% acc -o wheel -DSOLARIS wheeldriver.c wheel_library.c
```

(Windows)

```
C:> CL -Op -O1 -W1 -DMSWIN32 -Fewheel_library.exe wheel_library.c  
wheeldriver.c
```



NOTE: This example does not require any of the Standalone Library source files to be compiled and linked. If it did, you could add those files to the above command.

3. If you get any link errors, specifically missing symbols, it is most likely that you need to compile and link an SA Library File. Determine which SA library file contains the function(s) (per the link error message), and then compile and link with those files.
4. Run the program and some results will be output to the screen.

Example B-6 Driver Program (wheeldriver.c)

```
#include <stdlib.h>  
#include <stdio.h>  
  
#include "wheel_library.h"  
  
typedef struct _wheeldata {  
    struct _whl_velocity_ext  vel_data;  
    struct _whl_motion_ext    motion_data;  
    struct _whl_recoil_ext    recoil_data;  
} WheelData;
```

```

int main()
{
    WheelData wheels[4];
    int i;
    RT_FLOAT in1, in2;
    RT_FLOAT vel, motion, recoil, area;
    RT_FLOAT curTime;

    srand(281);

    for(i=0; i<4; i++) {
        whl_velocity_Initialize(&wheels[i].vel_data);
        whl_motion_Initialize(&wheels[i].motion_data);
        whl_recoil_Initialize(&wheels[i].recoil_data);
    }

    curTime = 0.0;

    while (curTime <= 5.0) {
        in1 = (RT_FLOAT)(rand() / 450.5);
        in2 = (RT_FLOAT)(rand() / 1004.25);
        printf("TIME = %f\n", curTime);
        for(i=0; i<4; i++) {
            in1 = in1 + (0.15*i);
            in2 = in2 - (0.05*i);
            whl_velocity_Execute(in1, in2, &vel, curTime, &wheels[i].vel_data);
            whl_motion_Execute(vel, &motion, curTime, &wheels[i].motion_data);
            whl_recoil_Execute(vel, &recoil, &area, curTime, &wheels[i].recoil_data);

            printf("\tW%d\t%f\t%f\t%f\t%f\n",i,vel,motion,recoil,area);
        }

        curTime = curTime + 0.125;
    }

    return 0;
}

```

B.7 C++ API

This section describes the generated C++ code of the C++ SDK template.

B.7.1 Logical Design

The logical design of the API consists of a class for each of the top-level procedures. That class contains the private data and methods for properly invoking the AutoCode procedure.

Class

A class is created that presents an interface to the AutoCode generated object. The class name is of the form: *procedurename_proxy*.

Private Data Member

The private data of this class is a struct containing the private data needed to support the execution of the procedure. The structure is based on the following “boilerplate” which describes the kinds of data found in this structure.

```
struct _procedurename_ext {  
    procedurename_info  procedurename_i;  
    procedurename_s     procedurename_s_s;  
} m_procdata;
```

where *procedurename* is the name of the top-level procedure, and **m_procdata** is a member instance of the structure.

The structure contains the procedure’s INFO and STATE structures. For more information about these structures, see the *AutoCode Reference*. Depending on exactly what the procedure needs and/or optimizations you have used during code generation, these structures may or may not be present. Since this is a private data structure, there is no need to discuss its purpose or content any further.

Constructor Method

The purpose of this class constructor is to initialize an instance of the private data member structure.

Execute Method

This method is the calling interface from your code to the generated procedure. The function has the following “boilerplate” signature.

```
RT_INTEGER Execute(
    struct procedurename_u *U,
    struct procedurename_y *Y,
    RT_FLOAT TIME
);
```

or if using the AutoCode `-nouy` option:

```
RT_INTEGER Execute(
    RT_datatype u1 [, RT_datatype un]
    RT_datatype *y1 [, RT_datatype *yn]
    RT_FLOAT TIME
);
```

where struct *procedurename_u* is the procedure’s input argument structure; struct *procedurename_y* is the procedure’s output argument structure; TIME is the current simulated time measured from the start of the simulation.

Depending on the exact number of inputs and outputs, and whether the current time is needed, the interface varies for each procedure. You must declare an instance of the input/output structures and pass all necessary arguments to this function to properly execute the procedure. The function returns error code 0, which means no error. Any other code indicates an error that can be determined from predefined error codes in the `sa_defn.h` SA-Library header file.

B.7.2 Physical Design (cpp_sdk.tpl)

The physical design of the API is the grouping of functions and data structures into source files. There are two files generated by the template: a header file and a source file.

Header File

A header file is created that contains all of the data structures used by all of the procedures in the generated code. This includes the procedure's input, output, INFO and state data structures as well API class declarations. The header file name has the format *toplevel.h*, where *toplevel* is the name of the top-level discrete SuperBlock of the model.

Source File

A source file is created that contains code for all of the procedures including the API class methods for each of the top-level procedures. The source file name has the format *toplevel.cpp*, where *toplevel* is the name of the top-level discrete SuperBlock of the model.



NOTE: The actual AutoCode procedures are generated as free functions. This approach minimizes the amount of generated code. For example, if each of the procedure's descendent procedures were part of the class, then if more than one top-level procedure called the same procedure, that procedure would have to be implemented as member functions of both classes.

B.7.3 Physical Design (*cpp_sdk_m.tpl*)

The physical design of the API is the grouping of functions and data structures into source files. This template generates multiple files.

Header Files

A header file is *created for each* Standard and Startup Procedure that contains all of the data structures used by that procedure. This includes the procedure's input, output, INFO and state data structures. The name of each header file is based on the name of the Procedure SuperBlock. The proxy classes for the API functions for all top-level Standard Procedures are created in a separate header file. That header file name has the format *toplevel.h*, where *toplevel* is the name of the top-level discrete SuperBlock of the model.

Source File

A source file is *created for each* Standard and Startup Procedure that contains code for the procedure. The name of each source file is based on the name of the Procedure SuperBlock. The implementation of the API proxy classes for all top-level Standard Procedures are created in a separate source file. That source file name has the format *toplevel.cpp*, where *toplevel* is the name of the top-level discrete SuperBlock of the model.

B.7.4 Compilation and Link Details

The compilation and link details are the same as documented in the C API section.

B.7.5 Example 3: Sample Code (CPP-SDK)

Example B-7 illustrates the code generated in the header file and a sample driver program that calls the procedure. The top-level discrete SuperBlock is named **interfaces** and the top-level procedure is named **feather**.

Header File (*interfaces.h*)

Example B-7 Header File (*interfaces.h*)

```

/*****
|
|           AutoCode/C (TM) Code Generator
|           WIND RIVER SYSTEMS INC.,  SUNNYVALE, CALIFORNIA
|*****
rtf filename      : interfaces.rtf
Filename         : interfaces.h
Dac filename     : cpp_sdk.dac
Generated on    : Tue Feb  3 11:59:07 1998
Dac file created on : Tue Feb  3 11:59:06 1998

AutoCode Procedure Software Development Kit

Procedure Name      Inputs  Outputs  INFO  States
-----
feather             YES     YES     YES   NO

*/

#ifndef _INTERFACES_H_
#define _INTERFACES_H_

```

```
#include "sa_sys.h"
#include "sa_types.h"

/***** Procedure: feather *****/

/***** Inputs type declaration. *****/
struct _feather_u {
    RT_FLOAT input_val;
};

/***** Outputs type declaration. *****/
struct _feather_y {
    RT_FLOAT gain_out;
};

/***** Info type declaration. *****/
struct _feather_info {
    RT_INTEGER iinfo[5];
};

// //////////////////////////////////////

//
// CLASS: feather_proxy
//
class feather_proxy {
    //
    // methods
    //
public:
    //
    // Default Constructor
    //
    feather_proxy();

    // Function: Execute
    //
    // Abstract: Execute the procedure
    //
    // Parameters: Procedure's input and output variables
    //               Current time may be needed as well.
    //               Interface varies for each procedure.
    //
    // Returns: error status, 0 = no error,
    //           else see errors in sa_defn.h
    //
    RT_INTEGER Execute(
        struct _feather_u *U,
        struct _feather_y *Y);

    //
    // private data
    //
private:
    struct _feather_ext {
        struct _feather_info feather_i;
    } m_procdta;
};
```

```
};
#endif
```

Driver Program (main.cpp)

Example B-8 **Driver Program (main.cpp)**

```
#include <stdlib.h>
#include <stdio.h>
#include "interfaces.h"

int main() {
    struct _feather_u    in;
    struct _feather_y    out;
    feather_proxy        theFeather;
    int i;

    srand(304);
    for(i=0; i < 20; i++) {
        in.input_val = (RT_FLOAT)(rand() / 8.5);
        theFeather.Execute(&in, &out);
        printf("input: %f\toutput: %f\n", in.input_val, out.gain_out);
    }
    return 0;
}
```

B

B.7.6 Example 4: Wheel Program (CPP-SDK)

This example uses the supplied example model and source code to illustrate the steps needed to generate, compile, and execute the driver and SDK generated code.

Step 0: Set up

We need to set-up a current working directory. Create a new subdirectory, called **wheeltest**. Change directory to **wheeltest** to make it the current working directory and copy all of the SDK files into it.

Step 1: Generate API

We need to generate the API structures and functions. This is accomplished by performing the following steps:

1. Launch Xmath and SystemBuild from within the current working directory.
2. Load the example model file: **wheellib.cat**.
3. Generate code using AutoCode. From the Xmath Commands window, enter:

```
autocode "wheel_library",                                     {  
language="c",tplsrc="cpp_sdk.tpl",file="wheel_library.cpp",  
procs_only,nouy }
```

AutoCode should have generated two source files: **wheel_library.h** and **wheel_library.cpp**.

Step 2: Compile and Link API Functions and Driver Program

With the API functions generated, all we need to do is compile and link those API functions with the supplied driver program (**wheeldriver.cpp**) that is our example system. Perform the following steps from an operating system command prompt.

1. Copy SA Library header files into your current working directory. If you need other SA Library Files, especially the fixed-point implementations, copy those files. If you do not copy the header files, you can add **-I** include path directives to the compile line.
2. Compile and link the program

(UNIX – Solaris)

```
% CC -o wheel -DSOLARIS wheeldriver.cpp wheel_library.cpp
```

(Windows)

```
C:> CL -Op -O1 -W1 -DMSWIN32 -Fewheel_library.exe wheel_library.cpp  
wheeldriver.cpp
```



NOTE: This example does not require any of the Standalone Library source files to be compiled and linked. If it did, you could add those files to the above command.

3. If you get any link errors, specifically missing symbols, it is most likely that you need to compile and link an SA Library File. Determine which SA library file contains the function(s) (as per the link error message), and then compile and link with those files.
4. Run the program and some results will be output to the screen.

Driver Program (wheeldriver.cpp)

Example B-9 **Driver Program (wheeldriver.cpp)**

```

#include <stdlib.h>
#include <stdio.h>
#include "wheel_library.h"

typedef struct _wheeldata {
    whl_velocity_proxy vel;
    whl_motion_proxy motion;
    whl_recoil_proxy recoil_data;
} WheelData;

int main()
{
    WheelData wheels[4];
    int i;
    RT_FLOAT in1, in2;
    RT_FLOAT vel, motion, recoil, area;
    RT_FLOAT curTime;

    srand(281);

    curTime = 0.0;

    while (curTime <= 5.0) {
        in1 = (RT_FLOAT)(rand() / 450.5);
        in2 = (RT_FLOAT)(rand() / 1004.25);

        printf("TIME = %f\n", curTime);
        for(i=0; i<4; i++) {
            WheelData &whl = wheels[i];
            in1 = in1 + (0.15*i);
            in2 = in2 - (0.05*i);

            whl.vel.Execute(in1, in2, &vel, curTime);
            whl.motion.Execute(vel, &motion, curTime);
            whl.recoil.Execute(vel, &recoil, &area, curTime);

            printf("\tW%d\t%f\t%f\t%f\t%f\n",i,vel,motion,recoil,area);

```

```
    }  
    curTime = curTime + 0.125;  
  }  
  
  return 0;  
}
```

B.8 Ada API

This section describes the generated Ada code of the Ada SDK template.

B.8.1 Logical Design

The logical design of the API consists of a package specification and body for each of the top-level procedures.

Package

A package is created that presents an API to the AutoCode generated procedure. The package name is of the form: *procedurename_ext_pkg*.

Interface Record

The record datatype is defined to support the execution of the procedure. The record based on the following “boilerplate” which describes the kinds of data found in this record:

```
type procedurename_ext_t is record  
  procedurename_i : procedurename_info_t;  
  procedurename_s_s : procedurename_s_t;  
end record;  
  
type procedurename_ext_t_P is  
  access procedurename_ext_t;  
  
function ptr_of is new UNCHECKED_CONVERSION  
(SOURCE => SYSTEM.ADDRESS,  
  TARGET => procedurename_ext_t_P);
```

where *procedurename* is the name of the top-level procedure;

The structure contains the procedure's INFO and STATE structures. For more information about these structures, see the *AutoCode Reference*. Depending on exactly what the procedure needs and/or optimizations you have used during code generation, these structures may or may not be present. Since this is a private data structure, there is no need to discuss its purpose or content any further.

Initialize Procedure

The purpose of this procedure is to initialize an instance of the procedure's interface record. The procedure has the following signature. This procedure must be called once for each instance of the interface record and before any other API function.

```
procedure procedurename_Initialize(
    idata : procedurename_ext_t_P
);
```

Execute() Function

This function is the calling interface from your code to the generated procedure. The function has the following "boilerplate" signature.

```
function procedurename_Execute(
    U : in procedurename_u_t_P;
    Y : in procedurename_y_t_P;
    TIME : in RT_FLOAT;
    idata : in procedurename_ext_t_P
) return RT_INTEGER;
```

or if using the AutoCode `-nouy` option

```
function procedurename_Execute(
    u1 : in RT_datatype; [un : in RT_datatype;]
    y1 : in RT_datatype_P; [yn : in RT_datatype_P;]
    TIME : in RT_FLOAT;
    idata : in procedurename_ext_t_P
) return RT_INTEGER;
```

where *procedurename_u_t_P* is the access pointer to the procedure's input argument record, and *procedurename_y_t_P* is the access pointer to the procedure's output argument record; **TIME** is the current simulated time measured from the start of the simulation.

Depending on the exact number of inputs, outputs and if the current time is needed, the interface varies for each procedure. You must declare an instance of the input/output records and pass all necessary argument to this function to properly execute the procedure. The function returns an error code, 0 meaning no error, any other is an error that can be determined from predefined error codes in the `sa_defn.a` SA-Library header file.

B.8.2 Physical Design (package `feather_ext_pkg`)

All package specifications and bodies for the AutoCode procedures and the API packages are generated in one output file.

B.8.3 Compilation and Link Details

The compilation and link details are specific to each Ada compiler. The process follows relatively the same steps used to compile the standalone simulation.

B.8.4 Example 5: Sample Code (Ada-SDK)

[Example B-10](#) illustrates the API package specification and a sample driver program ([Example B-11](#)) that calls the procedure. The top-level discrete SuperBlock is named: **interfaces** and the top-level procedure is named: **feather**.

Package Specification

Example B-10 **API Package Specification for `feather_ext_pkg`**

```
with SA_TYPES;      use SA_TYPES;
with SA_DEFN;       use SA_DEFN;
with SYSTEM_DATA;  use SYSTEM_DATA;
with SYSTEM;
with UNCHECKED_CONVERSION;
with feather_pkg;  use feather_pkg;

package feather_ext_pkg is

    type feather_ext_t is record
        feather_i : feather_info_t;
    end record;

    type feather_ext_t_P is access feather_ext_t;
    function ptr_of is new UNCHECKED_CONVERSION
        (SOURCE => SYSTEM.ADDRESS, TARGET => feather_ext_t_P);
```

```

procedure feather_Initialize( idata : in feather_ext_t_P );
function feather_Execute(
    U : in feather_u_t_P;
    Y : in feather_y_t_P;
    idata : in feather_ext_t_P) return RT_INTEGER;

end feather_ext_pkg;

```

Driver Program

Example B-11 Driver Program for API Package Specification

```

with SA_TYPES;          use SA_TYPES;
with feather_pkg;      use feather_pkg;
with feather_ext_pkg;  use feather_ext_pkg;
with TEXT_IO;         use TEXT_IO;
procedure main is
    package FLT_IO is new Float_IO(RT_FLOAT);
    in   : feather_u_t;
    out  : feather_y_t;
    f_ext : feather_ext_t;
begin
    feather_Initialize( ptr_of(f_ext'ADDRESS) );

    for i in 1..20 loop
        in.input_val := RT_FLOAT(i) / 8.5;
        feather_Execute( ptr_of(in'ADDRESS), ptr_of(out'ADDRESS),
            ptr_of(f_ext'ADDRESS));
        FLT_IO.put(in.input_val);
        FLT_IO.put(out.gain_out);
    end loop;
end main;

```


C

Sample AutoCode Output

This appendix shows the generated AutoCode output (in [Example C-1](#)) from the IfThenElse example shown in [Chapter 9](#).

Example C-1 Generated Software from IfThenElse Model

```
/*
|
|           AutoCode/C (TM) Code Generator 70mx0912
|           WIND RIVER INC., SUNNYVALE, CALIFORNIA
|
|*****
RTF filename      : C:\WINNT\top.rtf
Filename         : C:\WINNT\top.c
Generated on     : Fri Sep 15 11:03:58 2000

Dac filename     :
\\torque\mx70\NT\70mx0912\wini_70mx0912\case\ACC\templates\c_sim.dac
Dac file created on : Tue Sep 12 03:08:49 2000
Options         : -l c

*****
--
--   Number of External Input : 1
--   Number of External Output: 2
--
--   Scheduler Frequency:      10.0
--
--   SUBSYSTEM  FREQUENCY  TIME_SKEW  OUTPUT_TIME  TASK_TYPE
--   -----
--   1          10.0      0.0        0.0          PERIODIC
--
--   Number of Procedures : 2
--
--   Procedure Name           Inputs  Outputs  INFO  States
--   -----
--   p1                       YES     YES     YES   NO
```

```
-- p2                YES      YES      YES      NO
--
*****/

#include <stdio.h>
#include <math.h>
#include "sa_sys.h"
#include "sa_defn.h"
#include "sa_types.h"
#include "sa_math.h"
#include "sa_user.h"
#include "sa_utils.h"
#include "sa_time.h"
#include "sa_fuzzy.h"

/** System Data **/

/***** Structure to drive disconnected input/output. *****/

#define EPSILON          1.4901161193847656E-008
#define EPS              (4.0 * EPSILON)
#define ABSTOL           EPSILON
#define XREMAP           1

#define SCHEDULER_FREQ  10.0
#define NTASKS          1
#define NUMIN            1
#define NUMOUT           2
#define SCHEDULER_ID    0
#define PREEMPTABLE     2

enum TASK_STATE_TYPE { IDLE, RUNNING, BLOCKED, UNALLOCATED };

static RT_INTEGER          ERROR_FLAG [NTASKS+1];
static RT_BOOLEAN         SUBSYS_PREINIT [NTASKS+1];
static RT_BOOLEAN         SUBSYS_INIT [NTASKS+1];
static enum TASK_STATE_TYPE TASK_STATE [NTASKS+1];

/***** System Ext I/O type declarations. *****/
struct _Subsys_1_out {
    RT_FLOAT apple_out;
    RT_FLOAT orange_out;
};

struct _Sys_ExtIn {
    RT_FLOAT top_1;
};

/***** System Ext I/O type definitions. *****/
struct _Subsys_1_out subsys_1_out;
struct _Sys_ExtIn sys_extin;

static RT_FLOAT          ExtIn      [NUMIN+1];
static RT_FLOAT          ExtOut     [NUMOUT+1];
```

```
/* Model variable definitions. */
VAR_FLOAT abcd;
VAR_FLOAT efg;

/***** Procedures' declarations *****/

#ifndef p1_STRUCTS
#define p1_STRUCTS

/***** Procedure: p1 *****/

/***** Inputs type declaration. *****/
struct _p1_u {
    RT_FLOAT pl_1;
};

/***** Outputs type declaration. *****/
struct _p1_y {
    RT_FLOAT apple;
};

/***** Info type declaration. *****/
struct _p1_info {
    RT_INTEGER iinfo[5];
};

#endif

void p1 (
    struct _p1_u *U
    , struct _p1_y *Y
    , struct _p1_info *I
);

/*****

#ifndef p2_STRUCTS
#define p2_STRUCTS

/***** Procedure: p2 *****/

/***** Inputs type declaration. *****/
struct _p2_u {
    RT_FLOAT p2_1;
};

/***** Outputs type declaration. *****/

struct _p2_y {

    RT_FLOAT orange;

};

/***** Info type declaration. *****/
struct _p2_info {
```

```

    RT_INTEGER iinfo[5];
};

#endif

void p2 (
    struct _p2_u *U
    , struct _p2_y *Y
    , struct _p2_info *I
    );

/*****
/***** Procedures' definitions *****/

/***** Procedure: p1 *****/

void p1(    struct _p1_u *U
    ,struct _p1_y *Y
    ,struct _p1_info *I
    )
{

    RT_INTEGER *iinfo = &I->iinfo[0];

    /**** Local Block Outputs. *****/

    RT_FLOAT abcd_1;
    RT_FLOAT p1_1_1;
    RT_FLOAT p1_2_1;

    /**** Output Update. *****/
    /* ----- Read from Variable */
    /* {p1..3} */
    abcd_1 = abcd;
    /* ----- Gain Block */
    /* {p1..1} */
    p1_1_1 = 22.0*U->p1_1;
    /* ----- Summer */
    /* {p1..2} */
    p1_2_1 = p1_1_1 + abcd_1;
    /* ----- Saturation */
    /* {p1..13} */
    Y->apple = MIN(MAX(-50.0,p1_2_1),50.0);

    iinfo[1] = 0;

EXEC_ERROR: return;
}

/***** Procedure: p2 *****/
void p2(    struct _p2_u *U
    ,struct _p2_y *Y
    ,struct _p2_info *I
    )

```

```

{
    RT_INTEGER *iinfo = &I->iinfo[0];

    /***** Local Block Outputs. *****/

    RT_FLOAT abcd_1;
    RT_FLOAT p2_1_1;
    RT_FLOAT p2_2_1;

    /***** Output Update. *****/
    /* ----- Read from Variable */
    /* {p2..3} */
    abcd_1 = efg;
    /* ----- Gain Block */
    /* {p2..1} */
    p2_1_1 = 15.0*U->p2_1;
    /* ----- Summer */
    /* {p2..2} */
    p2_2_1 = p2_1_1 + abcd_1;
    /* ----- Saturation */
    /* {p2..13} */
    Y->orange = MIN(MAX(-50.0,p2_2_1),50.0);

    iinfo[1] = 0;
EXEC_ERROR: return;
}

/***** Tasks declarations *****/

/***** Subsystem 1 *****/
void subsys_1( struct _Sys_ExtIn *U, struct _Subsys_1_out *Y);

/***** Tasks code *****/

/***** Subsystem 1 *****/
void subsys_1( struct _Sys_ExtIn *U
,struct _Subsys_1_out *Y
)
{

    static RT_INTEGER iinfo[4];

    /***** Local Block Outputs. *****/

    RT_FLOAT abcd_1;
    RT_FLOAT efg_1;
    static struct _p1_u p1_99_u;
    static struct _p1_y p1_99_y;
    static struct _p1_info p1_99_i;
    static struct _p2_u p2_23_u;
    static struct _p2_y p2_23_y;
    static struct _p2_info p2_23_i;

```

```

static struct _p1_u p1_21_u;
static struct _p1_y p1_21_y;
static struct _p1_info p1_21_i;
static struct _p2_u p2_4_u;
static struct _p2_y p2_4_y;
static struct _p2_info p2_4_i;
static struct _p2_u p2_22_u;
static struct _p2_y p2_22_y;
static struct _p2_info p2_22_i;
static struct _p1_u p1_14_u;
static struct _p1_y p1_14_y;
static struct _p1_info p1_14_i;

/***** Initialization. *****/

if (SUBSYS_PREINIT[1]) {
  iinfo[0] = 0;
  iinfo[1] = 1;
  iinfo[2] = 1;
  iinfo[3] = 1;
  p1_99_i.iinfo[0] = iinfo[0];
  p1_99_i.iinfo[3] = iinfo[3];
  p2_23_i.iinfo[0] = iinfo[0];
  p2_23_i.iinfo[3] = iinfo[3];
  p1_21_i.iinfo[0] = iinfo[0];
  p1_21_i.iinfo[3] = iinfo[3];
  p2_4_i.iinfo[0] = iinfo[0];
  p2_4_i.iinfo[3] = iinfo[3];
  p2_22_i.iinfo[0] = iinfo[0];
  p2_22_i.iinfo[3] = iinfo[3];
  p1_14_i.iinfo[0] = iinfo[0];
  p1_14_i.iinfo[3] = iinfo[3];
  SUBSYS_PREINIT[1] = FALSE;
  return;
}

/**** Output Update. *****/
/* ----- Read from Variable */
/* {top..1} */
abcd_1 = abcd;
/* ----- Read from Variable */
/* {top..11} */
efg_1 = efg;

/* ----- IfThenElse */
/* {top..2} */
if( abcd_1 >= 10.0 ) {

  /* ----- Procedure Super Block */
  /* {p1.99} */
  p1_99_u.p1_1 = U->top_1;
  p1(&p1_99_u, &p1_99_y, &p1_99_i);
  Y->apple_out = p1_99_y.apple;
  iinfo[0] = p1_99_i.iinfo[0];
  if( iinfo[0] != 0 ) {

```

```

        p1_99_i.iinfo[0] = 0; goto EXEC_ERROR;
    }
}

else if( abcd_1 <= 1.0 ) {

    /* ----- Procedure Super Block */
    /* {p2.23} */
    p2_23_u.p2_1 = U->top_1;
    p2(&p2_23_u, &p2_23_y, &p2_23_i);
    Y->orange_out = p2_23_y.orange;
    iinfo[0] = p2_23_i.iinfo[0];
    if( iinfo[0] != 0 ) {
        p2_23_i.iinfo[0] = 0; goto EXEC_ERROR;
    }
}

else if( abcd_1 > 10.0 && abcd_1 < 1.0 || (efg_1 == 1.0) ) {

    /* ----- Procedure Super Block */
    /* {p1.21} */
    p1_21_u.p1_1 = U->top_1;
    p1(&p1_21_u, &p1_21_y, &p1_21_i);
    Y->apple_out = p1_21_y.apple;
    iinfo[0] = p1_21_i.iinfo[0];
    if( iinfo[0] != 0 ) {
        p1_21_i.iinfo[0] = 0; goto EXEC_ERROR;
    }

    /* ----- Procedure Super Block */
    /* {p2.4} */
    p2_4_u.p2_1 = U->top_1;
    p2(&p2_4_u, &p2_4_y, &p2_4_i);
    Y->orange_out = p2_4_y.orange;
    iinfo[0] = p2_4_i.iinfo[0];
    if( iinfo[0] != 0 ) {
        p2_4_i.iinfo[0] = 0; goto EXEC_ERROR;
    }
}

else {

    /* ----- Procedure Super Block */
    /* {p2.22} */
    p2_22_u.p2_1 = U->top_1;
    p2(&p2_22_u, &p2_22_y, &p2_22_i);
    Y->orange_out = p2_22_y.orange;

    iinfo[0] = p2_22_i.iinfo[0];
    if( iinfo[0] != 0 ) {
        p2_22_i.iinfo[0] = 0; goto EXEC_ERROR;
    }
}

/* ----- Procedure Super Block */
/* {p1.14} */
p1_14_u.p1_1 = U->top_1;

```

```
        pl(&pl_14_u, &pl_14_y, &pl_14_i);
        Y->apple_out = pl_14_y.apple;
        iinfo[0] = pl_14_i.iinfo[0];
        if( iinfo[0] != 0 ) {
            pl_14_i.iinfo[0] = 0; goto EXEC_ERROR;
        }
    }

    if(iinfo[1]) {
        SUBSYS_INIT[1] = FALSE;
        iinfo[1] = 0;
    }
    return;

EXEC_ERROR: ERROR_FLAG[1] = iinfo[0];
            iinfo[0]=0;
}

/*****
** Initialize global data such as:
**     Scheduler data (subsystem code only)
**     Variable Blocks
**     %vars
**     Subsystem I/O
*****/
void Init_Application_Data ()
{
    RT_INTEGER cnt;

    /* Declare %var/varblk initialization data */

    /* Variable blocks initialization. */
    abcd = 0.0;
    efg = 0.0;

    /* Subsystem outputs initialization. */
    subsys_1_out.apple_out = -EPSILON;
    subsys_1_out.orange_out = -EPSILON;

    for( cnt=0; cnt<NUMOUT; cnt++ ){
        ExtOut[cnt] = -EPSILON;
    }

    SUBSYS_PREINIT[1] = SUBSYS_INIT[1];
    subsys_1(&sys_extin, &subsys_1_out);
}

/*-----*
*-- SCHEDULER --*
*-----*/

/**** Scheduler Data ****/
```

```

enum SUBSYSTEM_TYPE { CONTINUOUS, PERIODIC, ENABLED_PERIODIC,
                      TRIGGERED_ASYNC, TRIGGERED_ANT, TRIGGERED_ATR,
                      TRIGGERED_SAF, ECHART, NONE };

static const enum SUBSYSTEM_TYPE    TASK_TYPE           [NTASKS+1] =
    {NONE, PERIODIC};
static const enum TASK_STATE_TYPE   INITIAL_TASK_STATE [NTASKS+1] =
    {UNALLOCATED, IDLE};
static const RT_INTEGER              START_COUNT        [NTASKS+1] =
    {0, 0};
static const RT_INTEGER              SCHEDULING_COUNT    [NTASKS+1] =
    {0, 0};
static const RT_INTEGER              OUTPUT_COUNT        [NTASKS+1] =
    {0, 0};

static long int                      TIME_COUNT;
static RT_INTEGER                    TSK;
static RT_INTEGER                    SCHEDULER_STATUS;
static RT_INTEGER                    CURRENT_PRIORITY = NTASKS+1;
static RT_INTEGER                    LEVEL           = 0;
static RT_INTEGER                    READY_COUNT;
static RT_INTEGER                    READY_QUEUE [NTASKS+1];
static RT_BOOLEAN                    DISPATCH      [NTASKS+1];
static RT_INTEGER                    PRIORITY     [NTASKS+1];
static volatile RT_INTEGER           DISPATCH_COUNT;

struct TCB_TYPE
{
    enum SUBSYSTEM_TYPE    TASK_TYPE;
    RT_BOOLEAN             ENABLED;
    RT_INTEGER             START;
    RT_INTEGER             START_COUNT;
    RT_INTEGER             SCHEDULING_COUNT;
    RT_INTEGER             OUTPUT;
    RT_INTEGER             OUTPUT_COUNT;
    RT_BOOLEAN             DS_UPDATE;
    RT_BOOLEAN             EDGE_TRIGGER;
};
static struct TCB_TYPE    TCB [NTASKS+1];

/* Work area side indices for subsystems. */
static RT_INTEGER         SSWORKSIDE [NTASKS+1];
static RT_INTEGER         SSREADSIDE;

unsigned short           SCHEDULER_STATE;

#define Queue_Task(NTSK) \
    READY_COUNT++; \
    READY_QUEUE[READY_COUNT] = NTSK; \
    DISPATCH[NTSK] = TRUE; \
    TASK_STATE[NTSK] = RUNNING

#define Signal_An_Error(NTSK) \
    if( ERROR_FLAG[NTSK] == OK ) Error( NTSK, TIME_OVERFLOW ); \
    else Error( NTSK, ERROR_FLAG[NTSK] ); \

```

```
void Update_Outputs( RT_INTEGER NTSK )
{
    SSREADSIDE = SSWORKSIDE[NTSK];
    SSWORKSIDE[NTSK] = 1 - SSREADSIDE;
    switch(NTSK) {

        default:
            break;
    }
    return;
}

void System_Extin_Copy() {
    sys_extin.top_1 = ExtIn[0];
}

void System_Extout_Copy() {
    ExtOut[0] = subsys_1_out.apple_out;
    ExtOut[1] = subsys_1_out.orange_out;
}

void Update_DS_With_Externals() {

}

void Init_Scheduler()
{
    RT_INTEGER NTSK;
    for( NTSK=1; NTSK<=NTASKS; NTSK++ ) {
        TCB[NTSK].TASK_TYPE      = TASK_TYPE[NTSK];
        TCB[NTSK].ENABLED        = FALSE;
        TCB[NTSK].START          = START_COUNT[NTSK];
        TCB[NTSK].START_COUNT    = START_COUNT[NTSK];
        TCB[NTSK].SCHEDULING_COUNT = SCHEDULING_COUNT[NTSK];
        TCB[NTSK].OUTPUT         = OUTPUT_COUNT[NTSK];
        TCB[NTSK].OUTPUT_COUNT   = OUTPUT_COUNT[NTSK];
        TCB[NTSK].EDGE_TRIGGER   = FALSE;
        TASK_STATE[NTSK]         = INITIAL_TASK_STATE[NTSK];
        DISPATCH[NTSK]           = FALSE;
        ERROR_FLAG[NTSK]         = 0;
        SUBSYS_INIT[NTSK]        = TRUE;
        if (TASK_TYPE[NTSK]==TRIGGERED_ATR || TASK_TYPE[NTSK]==TRIGGERED_SAF ||
            TASK_TYPE[NTSK]==TRIGGERED_ASYNC){
            SSWORKSIDE[NTSK] = 0;
            TCB[NTSK].DS_UPDATE = TRUE;
        } else {
            SSWORKSIDE[NTSK] = 1;
            TCB[NTSK].DS_UPDATE = FALSE;
        }
    }
    DISPATCH_COUNT      = 0;
    CURRENT_PRIORITY    = NTASKS+1;
    READY_COUNT         = 0;
    READY_QUEUE[0]     = 0;
    READY_QUEUE[1]     = 0;
    SSWORKSIDE[0]      = 0;
}
```

```

ERROR_FLAG[0]      = 0;
DISPATCH[0]       = FALSE;
SUBSYS_INIT[0]     = FALSE;
SCHEDULER_STATUS  = OK;
TIME_COUNT        = -1;
}

void SCHEDULER()
{
    register  RT_INTEGER    NTSK;
    register  RT_INTEGER    I;
             RT_INTEGER    ITSK;

    TIME_COUNT = TIME_COUNT + 1;

    /** System Input **/

    SCHEDULER_STATUS = External_Input();

    System_Extin_Copy();

    if( SCHEDULER_STATUS != OK ){
        return;
    }

    /** Clear Ready Queue **/

    READY_COUNT      = 0;
    READY_QUEUE[1] = 0;

    /** Task Scheduling **/

    for( NTSK=NTASKS; NTSK>=1; NTSK-- ){

        switch( TASK_STATE[NTSK] ){
            case IDLE :

                switch( TCB[NTSK].TASK_TYPE ){
                    case CONTINUOUS :
                    case PERIODIC :
                        if( TCB[NTSK].START == 0 ){
                            Queue_Task(NTSK);
                            Update_Outputs(NTSK);
                            TCB[NTSK].START = TCB[NTSK].SCHEDULING_COUNT;
                        } else {
                            TCB[NTSK].START = TCB[NTSK].START - 1;
                        }
                    }
                break;

            case ENABLED_PERIODIC :
                if( !TCB[NTSK].ENABLED ){
                    TASK_STATE[NTSK] = BLOCKED;
                } else if( TCB[NTSK].START == 0 ){
                    Queue_Task(NTSK);
                    Update_Outputs(NTSK);
                }
            }
        }
    }
}

```

```
        TCB[NTSK].START = TCB[NTSK].SCHEDULING_COUNT;
    }else{
        TCB[NTSK].START = TCB[NTSK].START - 1;
    }
    break;

case TRIGGERED_ASYNC :
    if( TCB[NTSK].OUTPUT == 0 ){
        Update_Outputs(NTSK);
        TASK_STATE[NTSK] = BLOCKED;
        if( TCB[NTSK].START == 0 ){
            Queue_Task(NTSK);
            TCB[NTSK].START = 1;
        }
    }
    break;

case TRIGGERED_ANT :
    if( TCB[NTSK].START == 0 ){
        Queue_Task(NTSK);
        Update_Outputs(NTSK);
        TCB[NTSK].START = 1;
    }
    break;

case TRIGGERED_ATR :
    if( TCB[NTSK].OUTPUT == 0 ){
        Update_Outputs(NTSK);
        TASK_STATE[NTSK] = BLOCKED;
        if( TCB[NTSK].START == 0 ){
            Queue_Task(NTSK);
            TCB[NTSK].OUTPUT = TCB[NTSK].OUTPUT_COUNT;
            TCB[NTSK].START = 1;
        }
    } else {
        TCB[NTSK].OUTPUT = TCB[NTSK].OUTPUT - 1;
    }
    break;

case TRIGGERED_SAF :
    if( TCB[NTSK].OUTPUT == 0 ){
        Update_Outputs(NTSK);
        TASK_STATE[NTSK] = BLOCKED;
        if( TCB[NTSK].START == 0 ){
            queue_task(NTSK);
            TCB[NTSK].START = 1;
        }
    }
    break;
}
break;
```

```
case RUNNING :
```

```

switch( TCB[NTSK].TASK_TYPE ){
  case CONTINUOUS :
  case PERIODIC :
    if( TCB[NTSK].START > 0 ){
      TCB[NTSK].START = TCB[NTSK].START - 1;
    } else {
      Signal_An_Error(NTSK);
      return;
    }
    break;

  case ENABLED_PERIODIC :
    if( TCB[NTSK].START > 0 ){
      TCB[NTSK].START = TCB[NTSK].START - 1;
    } else {
      Signal_An_Error(NTSK);
      return;
    }
    break;

  case TRIGGERED_ASYNC :
    if( ERROR_FLAG[NTSK] != 0 ){
      Signal_An_Error(NTSK);
      return;
    }
    break;

  case TRIGGERED_ANT :
    if( TCB[NTSK].START == 0 ){
      Signal_An_Error(NTSK);
      return;
    }
    break;

  case TRIGGERED_ATR :
    if(( TCB[NTSK].OUTPUT > 0 ) && (TCB[NTSK].START > 0)) {
      TCB[NTSK].OUTPUT = TCB[NTSK].OUTPUT - 1;
    } else {
      Signal_An_Error(NTSK);
      return;
    }
    break;

  case TRIGGERED_SAF :
    if( ERROR_FLAG[NTSK] != 0 ){
      Signal_An_Error(NTSK);
      return;
    }
    break;
}
break;

case BLOCKED :

  switch( TCB[NTSK].TASK_TYPE ){
    case ENABLED_PERIODIC :

```

```
        if( TCB[NTSK].ENABLED ){
            Queue_Task(NTSK);
            Update_Outputs(NTSK);
            TCB[NTSK].START = TCB[NTSK].SCHEDULING_COUNT;
        }
        break;

    case TRIGGERED_ASYNC :
        if( TCB[NTSK].START == 0 ){
            Queue_Task(NTSK);
            TCB[NTSK].START = 1;
        }
        break;

    case TRIGGERED_ATR :
        if( TCB[NTSK].START == 0 ){
            Queue_Task(NTSK);
            TCB[NTSK].OUTPUT = TCB[NTSK].OUTPUT_COUNT;
            TCB[NTSK].START = 1;
        }
        break;

    case TRIGGERED_SAF :
        if( TCB[NTSK].START == 0 ){
            Queue_Task(NTSK);
            TCB[NTSK].START = 1;
        }
        break;
    }
    break;
}
}

/**/ System Output /**/

Update_DS_With_Externals();

System_Extout_Copy();

SCHEDULER_STATUS = External_Output ();

/**/ Task Input Sample and Hold /**/

for( I=READY_COUNT; I>=1; I-- ){
    TSK = READY_QUEUE[I];
    switch (TSK){
        case 1:

            break;
        default:
            break;
    }
}

/**/ Signal End of Critical Section /**/
```

```

if( READY_COUNT > 0 ){
    if( READY_QUEUE[1] > DISPATCH_COUNT ){
        DISPATCH_COUNT = READY_QUEUE[1];
    }
    ITSK = READY_QUEUE[READY_COUNT];
    SCHEDULER_STATE = PREEMPTABLE;
}else{
    SCHEDULER_STATE = PREEMPTABLE;
    return;
}

/** Task Dispatching **/

while( ITSK < CURRENT_PRIORITY  && ITSK <= DISPATCH_COUNT ){
    Disable;
    if( DISPATCH[ITSK] ){
        LEVEL++;
        PRIORITY[LEVEL] = CURRENT_PRIORITY;
        CURRENT_PRIORITY = ITSK;
        DISPATCH[ITSK] = FALSE;
        Enable;
        switch (ITSK){
            case 1 :
                subsys_1(&sys_extin, &subsys_1_out);
                break;

            default : break;
        }
        Disable;
        if( ERROR_FLAG[ITSK] == OK ){
            TASK_STATE[ITSK] = IDLE;
        }
        CURRENT_PRIORITY = PRIORITY[LEVEL];
        LEVEL--;
    }
    Enable;
    ITSK++;
}
if( ITSK > DISPATCH_COUNT ){
    DISPATCH_COUNT = 0;
}
}

/*-----*
*-- MAIN --*
*-----*/

main()
{
    /** Initialize Scheduler **/

    Init_Scheduler();

    /* Initialize Application Data */

    Init_Application_Data();
}

```

```
    /** User Initialization */  
  
    Implementation_Initialize (&ExtIn[0], NUMIN, &ExtOut[0], NUMOUT,  
                             SCHEDULER_FREQ);  
  
    /** Start Scheduler */  
    SCHEDULER_STATUS = Background ();  
  
    /** User Termination */  
  
    Implementation_Terminate();  
  
    exit(0);  
}
```



NOTE: This example is quite long, but it illustrates one of the key advantages of AutoCode. This entire C file is generated from one top-level block. The length of this file helps to contrast the benefits of generating code from a model versus hand-coding. The logic of the code will be a reflection of the logic of the model.

Index

Symbols

`$env_var` 19
`%env_var%` 19

A

`ac_timing` option 29
`acmake` 28, 58
Ada code 176
`ada_intgr.tpl` 113
`ada_rt.tpl` 113
`ada_sdk.tpl` 176
Advanced dialog 21
algebraic loops 18, 113
algorithmic procedures 12
`allscope` option 161
application program 59
`arraymin` option 161
asynchronous subsystems 74, 78
 background procedure 79, 82
 interrupt procedure 79, 82
 procedures 78
 start-up procedure 79
 triggered 81
AutoCode
 automatic code generation 6, 8
 blocks 143

BlockScript 140
 global variable blocks 148
 IfThenElse 150
 local variable blocks 149
 sequencing 154
 Standard Procedure SuperBlocks 148
 UCB 143
 UserCode Blocks 143
 variable blocks 148
 While 154
command options 16, 17
configuration options 139, 140
customizing process 139
generated code applications 25
generated reusable procedures 12
invoking 5
model limitations 18
model restrictions 18
options
 see individual option names. 161
options file 169
real-time application 9, 11
sequence for using 5
simulation 5
Tools Menu pulldown 18
Xmath command, autocode 18, 21
autocode command 161
 examples 16
 options 16
autostar command 15, 161

- example command for Ada 117
- example command for C 117
- examples 17
- options 17

autostar.opt 169

aux clock 132

B

background function 11, 60

backmap option 161

block sequencing 154

blocked state 64, 67, 71

BlockScript 11, 139, 140

bubble diagram 63

build

- full 28, 58
- incremental 28, 58

C

C code 176

C++ code 176

c_core.tpl 176

c_sdk.tp 176

c_sdk_m.tpl 176

c_sdkcore.tpl 176

categories of discrete systems 99

code generation 18

- automatic 5
- continuous systems 111
 - generated code sample - Ada 123
 - generated code sample - C 117
 - hints 127
 - how to generate 113
 - implicit frequency 115
 - limitations 113
- customize 5
- discrete systems 99
 - example Ada code 106
 - example C code 102
 - optimized code 101

- procedural code 101
- vectorized code 100

from OS

- for continuous systems 116
- for discrete systems 17

from within SystemBuild

- for discrete systems 16

from Xmath

- for continuous systems 114, 176
- for discrete systems 16

hybrid systems

- how to generate 113

code-comment tokens

- limitations 146
- using 145

compiling and linking 27

computational thread 60

config option 162

configuration file 169

configuration options 140

continuous code generation

- integrators 112
- limitations 113

continuous subsystem 53, 66, 67

continuous systems 66

- code generation
 - from OS 116
 - from Xmath 114, 176
- generated code sample - Ada 123
- generated code sample - C 117
- generating code 113
- hints 127
- implicit frequency 115
- integrator 66, 111, 112

continuous-time model 5

cpp_core.tpl 177

cpp_sdk.tpl 176

cpp_sdk_m.tpl 177

CPU task utilization 59

critical section (scheduler) 60, 69

csi 176

csi - standalone 115

csi option 162

Ctrl-G 20

customizing AutoCode 139

D

data parameterization 142
 discrete systems
 categories 99
 discrete time points 29
 discrete-time
 controller SuperBlocks 5
 dispatch list 63, 67
 dispatcher 10, 63, 64, 69, 83
 division by zero 162
 -DMSWIN32 194
 doublebuf option 162
 double-buffered outputs 70
 -DSOLARIS 194

E

elapsed time counter 69
 enable signal 60, 66, 71, 73
 enabled periodic subsystem as a state machine 71
 enabled periodic subsystems 61, 68, 71
 enabled subsystems 67
 epinfo option 162
 epsilon option 162
 epsilon_r 162
 errcheck option 162
 errors
 customizing overflow handling 97
 scheduler 95
 subsystem overflow 95
 examples
 autocode command 16
 autostar command 17
 generated code (continuous) - Ada 123
 generated code (continuous) - C 117
 extended time [te, ye] 29

F

file option 162
 files

ada_intgr.tpl 113
 ada_rt.tpl 113
 c_intgr.tpl 113
 c_sim.tpl 113
 sa_utils.<hll> 26
 utilities
 standalone 26
 finite state machine 63, 69
 first sample
 See skew
 fixed-point demo 32
fixed-point.dat 32
 fmarker option 163
 foverflow option 163
 free-running periodic subsystem 60, 61, 66, 67, 70
 as a state machine 70
 full build 28, 58

G

gencode.bat 130
 generated application 3
 compiling 27, 32, 39, 40, 48
 components 10
 implementing 7
 nature of 9
 generated code
 applications 25
 comparing with sim results 29
 compile and link 5
 keywords 29
 validate 6
 generated code applications 25
 generated program
 passing control 63
 generating code 18
 customizing 21
 generating real-time code 15
 glbvarblkopt option 163

H

- hardware in-the-loop testing 7
- help system 12
- high-level language
 - Ada 3
 - C 3
 - code 5, 15
- hints
 - continuous systems 127
- hybrid system 66
 - generating code 113

I

- I/O routines 11
- ialg** 176
- ialg option 163
- IALG options 23
- ID (subsystems) 68
- idle state 66, 67, 70
- IfThenElse block 150
- implicit frequency 115
- increasing SIMNT memory size 134
- incremental build 28, 58
- indent option 163
- initmerge option 164
- input names 48
- integrating generated code 31
- integrator 66, 111, 112
 - first order Runge-Kutta 112
 - fourth order Runge-Kutta 112
 - Kutta-Merson 112
 - second order Runge-Kutta 112
 - user-supplied 112
- interpmap option 164
- interrupt handler 11, 60
- interrupts 61
- ipath option 164

K

- krstyle option 164

L

- label names 48
- language option 164
- latched outputs 70, 94
- least common multiple rate 92
- limitations 18
- linesz option 164
- linking 27, 32, 39, 40, 48
- locvarblkopt option 164
- loopmin option 164

M

- Macro Procedure block 143
- main.cpp** 193
- major cycle 71
- makefile generation 8
- makefile.cmdline** 130
- manager/scheduler 9, 63
- mapfile option 164
- mapping
 - subsystems to processors 173
 - command line 174
 - pmap option 174
- MATRIX_x
 - AutoCode 4
 - product family 3
- minimum scheduler cycle 64
- Minmax Display tool 34
- minor cycle 64, 68, 71, 91
- minsf** 176
- minsf option 165
- model
 - testing 7
- model simulation
 - running applications 29
 - time vectors 29

N

namelen option 165
 negative-going edge (trigger) 74
 nobusmap_b token 165
 nodiscon option 165
 noerr option 165
 nogscope option 165
 noinfo option 165
 nomap option 165
 norestart option 165
 nosmooth option 165
 -**nouy** 189
 nouy option 165
 numproc option 166

O

offset
 (see skew)
 online help 12
 optimization 132
 optimized code
 types of optimization 101
 options file 169
 options option 166
 organization
 manual 1
 OS command options
 -csi 116
 -minsf 116
 output posting
 ANT 74, 77
 ASYNC 75
 ATR 74, 77
 SAF 77
 overflow (scheduler) 95
 overflow (subsystem) 95
 overflow (timing) 83

P

parameterization 5, 142
 variables 139
 parname option 166
 periodic task subsystems 59
 repetition rate 59
 periodicity 93
 pmap option
 example 174
 positive-going edge (trigger) 74
 pre-emption 83
 prio option
 example 172
 priomap option 166
 priorities, task 171
 procs_only option 166
 propconst option 166
 pseudo-rate scheduler 91, 92, 94

R

rapid prototyping 2, 5, 7, 30
 rate-monotonic
 algorithm 59
 scheduling 59
 ready queue 66, 67, 83
 real-time
 application 11
 application program 59
 code 3
 file 15, 17
 generating code 5, 15, 32
 simulation 31
 re-entrant dispatcher 69
 related publications 13
 repetition rate 91
 resettable integrator 144
 restrictions 18
 reusable procedures
 AutoCode-generated 12
 reuse option 166
 roundfloat option 166
 rtf option 15, 17, 167

rtos option 167
rtosfile option 167
running 32, 39, 40, 48
state 70

S

sa_defn_a 198
SAF (as-soon-as-finished) trigger 77
sample and hold 60, 66, 69, 83
sample rate 59, 60
sampling rate 64, 67, 91
scheduler 9, 11, 59, 63, 64
 critical section 60, 69
 dispatch list 67
 dispatcher 69, 83
 re-entrant 69
 elapsed time counter 69
 errors 95
 example timing diagram 91
 examples 84
 major cycle 71
 minimum cycle 64
 minor cycle 64, 68, 71, 91
 periodicity 93
 pre-emption 83
 pseudo-rate 91, 92, 94
 rate-monotonic 59
 ready queue 66, 67, 83
 repetition rate 91
 sampling rate 91
 scheduler overflow 95
 skew 93
 subsystem overflow 95
 causes 97
 timing overflow 66
 timing overflow (subsystem) 83
 timing requirement 91
scheduler operation 65
scheduler option 167
sd option 167
sequencing block 154
setsbdefault command 29
sim...{initmode} command 70

simulation
 options 29
 real-time 31
skew 60, 93
 example 93
 setting
 skew option 173
skew option
 example 173
skewmap option 167
smcallout option 167
software constructs 147
 example model 155
 generated code 155, 201
standalone simulation 26, 28
 set up environment variables 28
standalone utility file 5, 11, 26
 sa_utils.<hll> 26
Standard Procedure SuperBlocks 148
startpmap option 168
state machine 63
state transition diagram (STD) 69, 70
subsystemmap option 168
subsystem
 processor subsystem map
 command 174
subsystems 10, 60
 ATR 68
 constraints 83
 continuous 66, 67
 controlling execution 171
 dispatch list 67
 dispatching 83
 dispatching and pre-emption example 84
 dispatching operation with a pseudo-rate
 scheduler 92
 enabled periodic 61, 67, 68, 71
 as state machine 71
 execution queue 80
 free-running periodic 60, 61, 66, 67, 70
 as state machine 70
 least common multiple 92
 mapping options 171
 minor cycle 68
 overflow 95

- overflow causes 97
- periodicity 93
- pre-emptible 61
- processor subsystem map 173
- pseudo-rate 92, 94
- ready queue 66, 67, 83
- running under simulation 71
- sampling rate 67, 91
- scheduler examples 84
- scheduling 71
- setting priorities
 - automatic 171
 - using the prio option 171
- skew 93
 - example 93
 - setting 173
- state 64
 - blocked 64, 67, 71
 - idle 66, 67, 70
 - running 70
- task ID
 - NTASKS 68
- timing diagram 73
- timing overflow 66, 83
- timing requirement 67, 91
- triggered 61, 66
 - (ASAF or SAF) 75
 - as state machines 76, 77
 - output posting (SAF) 75
- triggered ANT 68
- triggered ASAF 68
- triggered asynchronous 74
- super_cruise.data** 133
- SuperBlock
 - top-level 30
- system 16
- SystemBuild 5

T

- target processor 7, 29
- target type
 - I80486 129
 - PPC604 129
 - SIMNT 129
- target-specific utilities 26
- task 60
- task priorities 171
- te, ye extended time 29
- template 140
 - command parameters 21
 - programming language (see tpl) 140
 - tpl program 21, 139
- template program 9, 11
- template programming language
 - (see tpl)
- testing a model 7
- time vectors 29
- timer interrupt handler 11
- timing
 - overflow 66, 83
 - requirement 67, 91
- timing diagram 73
- timing properties 94
- timing window 72
- top-level SuperBlock 30
- Tornado 2 2, 129
- tpl 140
 - files
 - ada_intgr.tpl 113
 - ada_rt.tpl 113
 - c_intgr.tpl 113
 - c_sim.tpl 113
 - program 139
- tpl programming language 21
- tpldac option 168
- tplsrc option 168
- trigger 60
- triggered as-soon-as-finished subsystems 68
- triggered asynchronous 68
- triggered asynchronous subsystems 74
- triggered at-next-trigger subsystems 68
- triggered at-timing-requirement subsystems 68
- triggered subsystems 61, 66
- triggered subsystems as state machines 76, 77
- triggered task subsystems 59
 - timing requirement 59
- typecheck option 168

U

UCB [11, 143](#)
ucbparams option [168](#)
UserCode Block [143](#)
(see UCB)
usrData.h [129](#)
utilities
 target-specific [26](#)

V

variable blocks [148](#)
 global [148](#)
 local [149](#)
vars option [169](#)
vbcallout option [169](#)
vectorized code
 variations [100](#)
vectormode option [169](#)
VxWorks [129](#)
 template [2, 129](#)
 usage notes [136](#)

W

wheeldriver.c [177, 186](#)
wheeldriver.cpp [177, 194](#)
wheelib.cat [177, 194](#)

X

Xmath
 generating code from [16](#)
 variables [142](#)

Z

ZeroCrossing blocks [144](#)